

A Grid and Cloud-based framework for high throughput bioinformatics

Keith Flanagan

*Submitted for the degree of Doctor of
Philosophy in the School of Computing
Science, Newcastle University*

October 2009

Acknowledgements

I would particularly like to thank my supervisors Professor Anil Wipat and Dr Matthew Pocock for their continuous support and advice.

I am grateful to all the support staff at the School of Computing Science for their thoughts and insights. In particular, I would like to thank Jim White and Chris Ritson for providing me with a large amount of additional temporary disk space on the computing science cluster nodes for experimentation purposes.

I would also like to thank the Microbase early-adopters, Sirintra Nakjang and Alice Illiasova, whose assistance and feedback was invaluable in debugging and testing.

I am also grateful to the members of the writing group, whose valuable feedback provided numerous grammatical improvements and caught many typographical errors.

Finally, I would like to thank my friends and family for supporting me throughout the PhD process.

Declaration

I declare that this thesis is my own work. No part of this thesis has previously been submitted for a degree or any other qualification in this or another University.

Keith Flanagan

December 2009

Abstract

Recent advances in genome sequencing technologies have unleashed a flood of new data. As a result, the computational analysis of bioinformatics data sets has been rapidly moving from a lab-based desktop computer environment to exhaustive analyses performed by large dedicated computing resources.

Traditionally, large computational problems have been performed on dedicated clusters of high performance machines that are typically local to, and owned by, a particular institution. The current trend in Grid computing has seen institutions pooling their computational resources in order to offload excess computational work to remote locations during busy periods. In the last year or so, commercial Cloud computing initiatives have matured enough to offer a viable remote source of reliable computational power. Collections of idle desktop computers have also been used as a source of computational power in the form of ‘volunteer Grids’.

The field of bioinformatics is highly dynamic, with new or updated versions of software tools and databases continually being developed. Several different tools and datasets must often be combined into a coherent, automated workflow or pipeline. While existing solutions are available for constructing workflows, there is a clear need for long-lived analyses consisting of many interconnected steps to be able to migrate among Grid and cloud computational resources dynamically.

This project involved research into the principles underlying the design and architecture of flexible, high-throughput bioinformatics processes. Following extensive research into requirements gathering, a novel Grid-based platform, Microbase, has been implemented that is based on service-oriented architectures and peer-to-peer data transfer technology. This platform has been shown to be amenable to utilising a wide range of hardware from commodity desktop computers, to high-performance cloud infrastructure. The system has been shown to drastically reduce the bandwidth requirements of bioinformatics data distribution, and therefore reduces both the financial and computational costs associated with cloud computing. The system is inherently modular in nature, comprising a service based notification system, a data storage system scheduler and a job manager. In keeping with e-

Science principles, each module can operate in physical isolation from each other, distributed within an intranet or Internet. Moreover, since each module is loosely coupled via Web services, modules have the potential to be used in combination with external service oriented components or in isolation as part of another system.

In order to demonstrate the utility of such an open source system to the bioinformatics community, a pipeline of inter-connected bioinformatics applications was developed using the Microbase system to form a high throughput application for the comparative and visual analysis of microbial genomes. This application, Automated Genome Analyser (AGA) has been developed to operate without user interaction. AGA exposes its results via Web-services which can be used by further analytical stages within Microbase, by external computational resources via a Web service interface or which can be queried by users via an interactive genome browser.

In addition to providing the necessary infrastructure for scalable Grid applications, a modular development framework has been provided, which simplifies the process of writing Grid applications. Microbase has been adopted by a number of projects ranging from comparative genomics to synthetic biology simulations.

Contents

1	Introduction	1
1.1	Data explosion in Bioinformatics	1
1.2	Scalability	2
1.3	Motivation	5
1.4	Project aims and objectives	7
1.5	Thesis structure	7
2	Background	9
2.1	Distributed Systems	10
2.1.1	Architectures	11
2.1.1.1	Client-server architectures	11
2.2	High-throughput computing	17
2.2.1	Programming models and scalable parallel computing	18
2.2.2	High-throughput computing platforms	19
2.2.2.1	Shared memory parallel computing	20
2.2.2.2	Distributed parallel computing	22
2.2.2.3	Distributed high-throughput computing	23
2.2.2.4	Distributed Computing	24
2.2.3	Summary	27
2.3	Data transfer protocols	27
2.3.1	Peer to peer, global-scale file transfer protocols and file systems	28
2.3.1.1	BitTorrent	29
2.3.2	Summary	30
2.4	Technologies underlying Grid systems	31
2.4.1	Web services	31
2.4.2	Workflows and pipelines	33
2.4.3	Notification-based orchestration	33
2.5	Grid architectures	35

2.5.1	Introduction	35
2.5.2	High performance grids	37
2.5.3	Commodity grids	38
2.5.4	P2P architectures in Grid organisation and communications	39
2.5.4.1	Peer to Peer approaches to resource matching	39
2.5.4.2	Mobile agents in Grids	41
2.5.4.3	Ensuring fairness in a P2P Grid	42
2.5.5	Cloud computing	42
2.5.6	Data management in high-throughput systems	43
2.6	High-throughput computation in e-Science and bioinformatics	44
2.7	Summary	47
3	Microbase	49
3.1	Introduction	49
3.2	Motivation	49
3.3	System-level requirements	51
3.3.1	Environment-specific considerations	52
3.3.2	Scalability requirements	53
3.3.3	Data handling requirements	54
3.3.4	Maintenance and extensibility requirements	54
3.3.5	Application support and workflow structuring	56
3.3.6	User requirements	58
3.3.6.1	Developer requirements	58
3.3.6.2	System administrator requirements	59
3.4	Architecture Overview	60
3.4.1	Facilitating flexible and extensible analysis pipelines	62
3.5	Supporting technologies	67
4	Notification system	69
4.1	Introduction	69
4.2	Motivation	69
4.3	Requirements	75
4.4	Architecture	76
4.4.1	Handling persistent messages	76
4.4.2	Handling broadcast messages	79
4.5	Implementation	81
4.6	Conclusion	83

5	Resource system	85
5.1	Introduction	85
5.2	Motivation	86
5.2.1	Data identification and storage	86
5.2.2	Data distribution	86
5.2.3	File version control	88
5.2.4	File querying	89
5.2.5	Pipeline extensibility	89
5.2.6	Developer usability	90
5.3	Requirements summary	92
5.3.1	Terminology	93
5.4	Architecture	93
5.4.1	Bulk data transport protocol	94
5.4.2	Resource client API	95
5.4.2.1	Azureus-Microbase integration	97
5.4.3	Torrent registry	99
5.4.4	Resource archiving	101
5.4.5	Downloading a resource	101
5.4.6	Publishing a resource	102
5.5	Discussion	103
6	Responders	107
6.1	Introduction	107
6.2	Motivation	108
6.2.1	Bridging Microbase and domain applications	108
6.2.2	Responder pipelining, extendibility and developer convenience	110
6.3	Requirements	110
6.3.1	Responder structure	112
6.4	Developer support for responders in Microbase	114
6.4.1	Responder initialisation	116
6.4.2	Handling notification events	118
6.4.3	Executing command line applications	122
6.5	Maven project layout	126
6.5.1	Responder project layout and interdependencies	128
6.5.2	Runtime role of Maven artifact information	130
6.6	Conclusions	131

7	Job management and enactment	134
7.1	Introduction	134
7.2	Motivation	135
7.3	Requirements	137
7.4	Architecture	138
7.4.1	Failure handling	141
7.4.2	Logging	142
7.4.3	File versioning	143
7.4.4	Overseeing computational work	144
7.4.4.1	Process of enacting a task	144
7.4.5	Job enactment	144
7.5	Compute client	148
7.6	Job execution by compute clients	149
7.7	Performance analysis	151
7.7.1	Introduction	151
7.7.2	Data collection and analysis	154
7.7.3	Timing results	158
7.7.4	Benchmarking methodology	160
7.8	Results	160
7.8.1	Performance benchmarks	160
7.8.2	Administration toolkit	164
7.9	Conclusions	164
8	Automated Genome Analyser	169
8.1	Introduction	169
8.2	Motivation	170
8.3	Architecture	170
8.3.1	AGA responders	172
8.3.2	AGA Viewer	180
8.4	Results	180
8.4.1	System configuration	180
8.4.1.1	BLAST-P NR responder using Amazon EC2 and Newcastle nodes	183
8.4.1.2	BLAST-P Pairwise responder using Amazon EC2	187
8.4.2	Benchmarking an entire pipeline of responders	191
8.5	Conclusions	193
8.5.1	Responder development experience and data flow	193
8.5.2	Future work	194

9	Discussion and conclusions	195
9.1	The Microbase System	195
9.1.1	Architecture choices	195
9.1.2	Scalability	197
9.1.3	Responder development framework	198
9.1.4	Comparisons with other frameworks	199
9.1.4.1	Programming models	202
9.2	Use cases	203
9.2.1	AGA	204
9.2.2	Mucosa project	205
9.2.3	Parallel metaSHARK	206
9.2.4	AptaMEMS-ID	208
9.2.5	iGem 2009	208
9.3	Evaluation	209
9.3.1	System efficiency and job design considerations	209
9.3.2	Service and data security in a Microbase system	210
9.3.3	Achievements	211
9.4	Future work	212
A	How to write a responder	216
A.1	Introduction	216
A.2	Microbase	217
A.2.1	Requirements	217
A.3	Quick-start virtual machine image	218
A.4	Responder architecture	220
A.5	Writing a responder	221
A.5.1	Root project directory	222
A.5.2	Compute job sub-project	223
A.5.2.1	BLAST	223
A.5.2.2	Java component	225
A.5.2.3	Implementing the Java component of a job	226
A.5.2.4	Packaging platform-native applications	230
A.5.2.5	Final job implementation directory	235
A.5.3	Event handler sub-project	235
A.5.3.1	Implementing the event handler	237
A.5.3.2	Modifying services.xml	241
A.6	Installation / Deployment	242
A.7	Testing	249

Abbreviations

AGA	Automated Genome Analyser
API	Application Programming Interface
BLAST	Basic Local Alignment Search Tool
BOINC	Berkeley Open Infrastructure for Network Computing
CDS	Coding Sequence
CIFS	Common Internet File System
CORBA	Common Object Request Broker Architecture
COTS	Common Off The Shelf
DCOM	Distributed Component Object Model
DHT	Distributed Hash Table
DSM	Distributed Shared Memory
FPGA	Field Programmable Gate Array
FIFO	First in, first out
FTP	File Transfer Protocol
GPU	Graphic Processor Unit
GSI	Grid Security Infrastructure
GUI	Graphical User Interface
GWT	Google Web Toolkit
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
ID	identifier
I/O	Input/Output
IPC	Inter-Process Communication
JAR	Java Archive
jar	Java Archive
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
LAN	Local Area Network
LSID	Life Science Identifier

MIMD Multiple Instruction Multiple Datastream
MISD Multiple Instruction Single Datastream
MPI Message Passing Interface
MPP Massively Parallel Processing
NFS Network File System
NUMA Non-Uniform Memory Access
OGSA Open Grid Services Architecture
P2P peer-to-peer
PTP point-to-point
POJO Plain Ordinary Java Object
POP Post Office Protocol
QoS Quality of Service
RDBMS Relational Database Management System
RPC Remote Procedure Call
RMI Remote Method Invocation
SCP Secure Copy
SIMD Single Instruction Multiple Datastream
SPMD Single Program Multiple Datastream
SISD Single Instruction Single Datastream
SGE Sun Grid Engine
SMB Server Message Block
SMP Symmetric Multi-processor
SNP single nucleotide polymorphism
SOA Service Oriented Architecture
SOAP Simple Object Access Protocol
SMTP Simple Mail Transfer Protocol
SQL Structured Query Language
SSE Streaming SIMD Extensions
SSH Secure Shell
SSL Secure Sockets Layer
STDOUT standard out

STDERR standard error

TTL Time To Live

UID unique identifier

VM Virtual Machine

XML Extensible Markup Language

WAN Wide Area Network

war Web Application Archive

WSDL Web Services Description Language

VPN Virtual Private Network

GRAM Grid Resource Allocation and Management

RFT Reliable File Transfer

RLS Replica Location Service

URL Uniform Resource Locator

WMS Workspace Management Service

Chapter 1

Introduction

1.1 Data explosion in Bioinformatics

Bioinformatics involves the application of computing science and mathematical techniques to help understand biological data. The first protein sequence, that of the B-chain of insulin, was determined during the 1950s by Sanger et al. [270]. This achievement was followed by Fier's group [162] in 1972, who were the first to determine the nucleotide sequence of a single gene. Advances in sequencing techniques during the 1970s [267, 210, 269] permitted the complete genome sequencing of several small sequences, such as individual genes of bacteriophages [93, 268]. Since then, the throughput of DNA sequencing has seen rapid increases due to the refinement [11], automation [212] and parallelisation of the process. A major achievement was the sequencing of the first genome of a free-living organism, *Haemophilus influenzae* Rd, in 1995 [96]. The Sanger method has seen several refinements resulting in greater efficiency; read lengths have approximately doubled in the past 10 years. Large-scale industrialisation of the Sanger method has also taken place, with several large sequencing centres now operating hundreds of sequencing machines [138]. New sequencing methods have also been developed [207, 138, 259, 264].

Improvements in technology, coupled with large-scale deployment of sequencing hardware has seen efficiencies of scale reduce the cost per genome sequence. Sequencing entire bacterial genomes is now almost routine. Despite the cost reductions achieved to date, there is still a long way to go before the long-term goal of the '\$1000 human genome sequence' is realised [276, 264, 311]. Nevertheless, genome databases are being populated with hundreds of bacterial sequences at an ever-increasing rate [32].

The GenBank [27] and EMBL [175] sequence databases were established during the early 1980s as

publicly available data repositories into which new DNA or protein sequences could be deposited. Following the release of the first two complete bacterial genome sequences in 1995 [96], there has been an explosion in the number of complete genome sequences that have been made publicly available. The major sequence database repositories, GenBank, EMBL and DDBJ [287]¹, have all shown similar continued exponential growth rates², mirroring the rate at which genome sequencing projects continue to produce new data [191].

As sequencing technologies were leading to an increase in the output of new sequence data, computers with greater memory capacity and computational power were becoming available. With the increased availability of genome sequences and computing power, considerable effort has been focused on developing software tools for automated sequence processing, including functional analyses, feature annotation, and comparison techniques, as well as visualisation utilities. Many software tools have been developed to aid with information storage and processing, and there is an active research area in developing new tools [203, 197]. Analysis tools are varied in scope and scale, ranging from scripts operating over flat-files running on a single lab-based PC, to large massively parallelised annotation pipelines [97]. Bioinformatics tools perform a wide range of analysis tasks. Tools such as GLIMMER [71], Genewise [34], InterPro [225] assist with sequence annotation tasks. Sequence alignment algorithms such as Needleman-Wunsch [231], Smith-Waterman [280], and utilities such as Blast [6] and MUMmer [176] allow the automated discovery of sequence similarities to be measured and single nucleotide polymorphisms (SNPs) to be identified. These tools aid the construction of phylogenetic trees and provide evidence for evolutionary processes such as gene duplications, deletions, mutation events and gene translations including lateral gene transfers. Other software packages provide graphical interpretations of primary sequence data or the secondary data produced by various analyses [98, 290, 48, 158].

1.2 Scalability

Without scalable systems and systematic approaches for the analysis of bioinformatics data, exhaustive studies of bioinformatics data sets are intractable [167, 91, 277]. The problem of performing large-scale analyses in bioinformatics stems from two root causes: scalability and complexity. Scalability issues arise from the exponential growth rate of primary data sets, the number of bioinformatics tools that need to be executed over those data sets, and the high computational and data storage costs associated with generating and maintaining secondary data sets — often $O(n^2)$ or worse per analysis

¹<http://www.ddbj.nig.ac.jp>

²<ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt>

tool, in many cases. For instance, executing exhaustive pairwise Blast [6] searches often requires specialised software and dedicated hardware in order to complete within a reasonable timeframe, since a single CPU would take many years to complete an exhaustive analysis [278, 288]. Many approaches have been proposed to address scalability problems, ranging from algorithm-specific dedicated hardware Field Programmable Gate Array (FPGA) [329, 140], to massively parallel use of generic computing hardware [278, 288, 139].

The second problem, complexity, arises from with the logistics of assembling multiple computationally-intensive analysis tools to run as a pipeline. As the number of tools that are required to execute in a high-throughput, automated fashion increases, co-ordination of structured data flows between processes becomes essential. The output of one program may need to be supplied to another as input data [31]. Therefore, in addition to providing scalable execution and data management, software platforms are required that are flexible enough to support and maintain sets of bioinformatics tools organised into workflows [202].

To address these issues, numerous high-throughput annotation pipelines have been developed that are capable of executing a range of bioinformatics applications on large, high-end dedicated computer clusters consisting of hundreds or thousands of nodes [147, 97, 31, 313].

Bioinformatics researchers have a long tradition of sharing data with their peers via the Internet. Initially, cross-project data sharing was achieved predominantly via static or dynamic Web sites. These sites were primarily designed for human interaction, and therefore pose difficulties for effective automated data retrieval. Data extraction from standard web sites often requires the use of ‘screen-scraping’ techniques, and is therefore ‘fragile’, being heavily dependent on the on-screen structure [67]. However, as the number and use of online data resources has increased, it has become increasingly necessary for machines to interact with, and transfer information among, remote resources. With the emergence of Service Oriented Architectures (SOAs) and their supporting technologies, it has now become commonplace to expose bioinformatics analysis applications or results databases as Web services [232, 323, 275]. Web services allow machines to interact with remotely-exposed data and analysis tools programatically, via well-defined service interfaces. Automation toolkits such as Taverna [236] and Kepler [5] enable the construction of complex workflows that utilise and co-ordinate multiple remotely-hosted services to achieve a particular goal. Workflow enactment permits data to flow from one service to another in a completely automated fashion. On completion of the workflow, result data is returned to the biologist. Workflow automation has been shown to save biologists a large amount time, by removing manual ‘copy and paste’ operations that would otherwise be required to move data between sites [152].

Grid technologies are rapidly gaining popularity for assistance with bioinformatics data processing [139, 65]. Although there are multiple definitions as to what constitutes ‘Grid computing’ [108], Grid technologies typically involve large numbers of distributed heterogeneous computational resources that are spread across several geographical locations. Computational Grids are a means for researchers to obtain and share large amounts of computational power and data storage either within their own institution, or across institutional and geographical boundaries. Some definitions of ‘Grid technology’ focus on large-scale data storage and interoperability, while others are more concerned with the use of dedicated, high-performance compute clusters. At the other end of the spectrum, several Grid projects, known as ‘desktop Grids’ implement ‘cycle-stealing’ techniques to acquire a large total amount of computational power through the use of multiple desktop-grade machines, each with modest hardware specifications. It is recognised that there is a need for exhaustive analyses of biological data, that the continual production of new primary data requires automatically-updating secondary data sets, and that achieving this functionality requires something like high-throughput Grid technology in order to be feasible [313].

A major research challenge in bioinformatics is the identification and removal of bottlenecks in the shift towards parallelisation and high-throughput approaches to data processing [65]. Of particular interest is the utilisation of recent developments in Grid and Cloud computing. These technologies have enormous potential in terms of computational power, but there are numerous challenges in leveraging this power. For example, problems may be faced in mapping computational problems in bioinformatics to new infrastructures and models for distributed computing that are becoming available through various Grid and Cloud computing initiatives. Such infrastructures are inherently heterogeneous in terms of hardware, and are more likely to be accessed via the Internet requiring applications to cope with high latencies and constrained bandwidth. Monetary charges may also be incurred for the use of a hardware resource.

The suitability of SOAs and peer-to-peer (P2P) infrastructures were considered for providing the co-ordination and data transfer operations that a Grid platforms require. Of particular interest is how Grid functionality is made accessible to application developers without exposing the underlying implementation complexities. Also of interest is how existing analysis applications were integrated into a high-throughput distributed system. This project therefore also focuses on the maintainability and extensibility issues of a platform that must be accessible to application developers, while at the same time being scalable and reliable enough to manage a Grid composed of non-dedicated, administratively-restricted, heterogeneous worker nodes. Developing applications in such an environment has been described in the literature as being ‘extremely difficult’ [172].

Bioinformatics analyses are very often composed of multiple applications arranged into a workflow or processing pipeline [236]. Constructing and operating such analysis workflows that operate within a Grid environment is a difficult task that poses a significant challenge for application developers [185].

In addition to the need to execute multiple analysis tools, large amounts of primary and secondary analysis data must be managed efficiently in order to construct a consistent, integrated data set. Amassing large repositories of sequence information raises a number of post-processing research challenges. For example, how to process and manage large amounts of sequence and annotation data; how to effectively integrate secondary data sets; how to expose data to third parties, and allow them to efficiently query data repositories.

1.3 Motivation

A large organisation might have several hundred, or thousands of desktop computers. It is likely that at least some of the time, a large proportion of these will be idle, with no active users [227, 318]. There is therefore a potentially vast amount of computing power not being utilised. Distributed systems have the potential to be expandable to extremely large proportions. For instance, some distributed search systems with an underlying P2P implementation are able to span millions of Internet-connected computers, utilising entirely distributed and self-maintaining data structures [37, 223]. However, if ‘desktop Grids’ are to even approach this level of scalability, several challenges must be addressed.

Distributed computation platforms based on loose collections of desktop computers have different properties to closely-coupled clusters of dedicated servers. These differences pose several challenges when attempting to efficiently utilise remote computational power. Although the combined raw CPU power of a distributed system composed of commodity hardware may equal or even exceed a dedicated compute cluster, the actual throughput that can be achieved in terms of utilisation of individual CPUs may be much lower and depends heavily on the types of jobs that are to be run. An individual nodes’ local disk is also typically smaller and slower than server equivalents and is not necessarily backed up regularly, leading to long-term data storage reliability problems. Network connections between loosely-coupled systems typically have much less bandwidth available than dedicated compute clusters, resulting in slower file transfers and higher latencies. Furthermore, general purpose desktop computers do not necessarily have domain-specific software installed, necessitating more network transfers and software installation overheads than are required for pre-loaded nodes of a dedicated

compute cluster.

Another problem facing Grids composed of desktop computers are unpredictable user interruptions — a user may reclaim the computer from the Grid at any moment. Therefore, it must be possible to restart computational work or migrate it to another node, preferably without losing large amounts of already completed work. Several frameworks have been developed that address several of the issues involved with the utilisation of commodity hardware. For instance, Condor [192], SGE³ and BOINC [9] are all widely used in many areas of research [25, 234, 237, 277].

Computational tasks that perform large amounts of isolated CPU-intensive work and transmit only small amounts of data across the network are ideal for use within a distributed compute cluster, since the impact of slow or high-latency network connections are minimised. Unfortunately in bioinformatics this is not always the case and data distribution is a major problem where large data sets must be transferred to many worker nodes. For instance, although multiple `Blast` analyses can be run independently, the `Blast` databases required for each alignment may run into many tens or hundreds of megabytes. Transferring such amounts of data to hundreds of nodes is a logistical challenge, requiring infrastructure capable of large-scale file distribution. Even a small cluster of machines might overwhelm a central server, or the network connection it relies on. Recently, a number of P2P solutions have been proposed with respect to data transfer and service organisation within grid-based systems to address these scalability issues [89, 126].

Access to worker nodes by Grid applications can be made more difficult by institutional or administrative reasons including: file-system permissions, ownership of the machines, and standardisation around a common Grid middle-ware infrastructure. These types of restrictions may limit the potential of the kinds of applications that may be executed on worker nodes. Large organisations also typically have a wide range of machines, running multiple operating systems and with different hardware architectures and capabilities. Such heterogeneous environments introduce difficulties when attempting to run third-party platform-specific executables, particularly when analysis tools are not necessarily pre-installed, since platform differences must be resolved at run-time.

Many of the tools that bioinformaticians use take somewhere in the region of several minutes to an several hours to execute on current typical desktop computer hardware with a typical data item, such as a genome sequence. Bioinformatics tools are typically amenable to process-level parallelisation by executing multiple instances over different data sets on different CPUs or different computers. Applications will therefore typically run in isolation, requiring little or no Inter-Process Communication (IPC). Therefore most of the data transfer requirements occur at the initialisation

³<http://www.sun.com/software/sge/>

and termination phases of execution, although sporadic access to external Web services or databases may also be required. Many existing analysis tools are ideally suited to running within a distributed environment composed of ‘small’ machines available for short to medium periods of time. However, it is unlikely that these applications are aware of frameworks such as Condor [192], and so will not make use of advanced features such as job checkpointing [193].

1.4 Project aims and objectives

This project aimed to research principles underlying the design and architecture of flexible, high-throughput bioinformatics processes and to demonstrate the application of these principles by developing a software implementation.

Objectives

To achieve these aims it was necessary to meet a number of objectives:

1. To establish the motivation and system requirements for Grid system capable of executing long lived bioinformatics analyses in a dynamic execution environment.
2. To develop a Grid based notification system that allowed the coordination of processes within a distributed computing environment.
3. To develop a system to efficiently manage programmatic and data resources within this environment.
4. To develop a user-friendly mechanism for packaging legacy bioinformatics applications to be executed on the Grid.
5. To develop a job management system to oversee job execution and completion, ensuring system robustness and maximise computational efficiency.
6. To apply the system to the development of a resource for comparative genome analysis in order to demonstrate its utility.

1.5 Thesis structure

This thesis is divided into the following parts:

- Chapter 2 provides the necessary background information and literature review of previous work related to this thesis.
- Chapter 3 introduces the motivations, system requirements and high-level architecture of a Grid system capable of executing long-lived bioinformatics analyses in a dynamic execution environment of commodity hardware.
- Chapters 4, 5 and 7 describe the major core components of the Microbase system, providing detailed descriptions how the architectures of these subsystems satisfy the high level requirements presented in Chapter 3. In particular, the combination of the approaches described in these chapters form a novel infrastructure
- Chapter 6 presents a development framework and a design pattern for implementing Grid-aware applications, or wrappers for existing applications. This chapter builds on the work presented in earlier chapters by describing an abstract interface to the Grid system that hides many of the complexities of a distributed computing environment. This Chapter also presents a software ‘design pattern’ aimed at bioinformatics developers, which if followed, permits highly scalable and flexible analysis pipelines to be constructed.
- Chapter 7 describes a distributed job enactment environment capable of executing applications constructed using the design pattern introduced in Chapter 6. By taking advantage of service-oriented architectures, the event system presented in Chapter 4, and the distributed file transfer system detailed in Chapter 5, the enactment system has been shown to manage jobs globally across a number of data centres.
- Chapter 8 describes the construction and implementation of a bioinformatics analysis pipeline that utilises Microbase to analyse bacterial sequences. Performance results of several pipeline enactments with different hardware configurations are presented.
- Chapter 9 presents an overall discussion and evaluation of the aims and objects. It describes the achievements of the project, including details of projects that have used the framework and design pattern to solve real-world bioinformaticscomputational problems. An outline of potential future work is also provided.

Chapter 2

Background

The first part of this chapter discusses approaches to high performance computing and relevant distributed systems technologies including Service Oriented Architectures (SOAs), workflow-based systems, distributed file transfer methods and Grid computing. A summary of current bioinformatics analysis pipelines is then presented, followed by an introduction to various bioinformatics data sources and analysis tools that were used for this project.

Bioinformatics concerns analysis of biological data using computers. Developments in computing hardware over the past few decades have effectively advanced computing power at an exponential rate¹. However, the growth of bioinformatics data sets and the development of new analysis techniques show similar and in some cases even higher growth patterns [142, 27]. Bioinformatics has therefore become reliant on distributed systems for the sharing and publication of data, and on parallel computation for enabling the tractable analysis of large amounts of data [41, 82].

One of the main aims of bioinformatics is to reduce the human workload involved in analysing biological data by utilising automated methods and large numbers of machines to perform the same or equivalent types of analyses as humans. Machine-based analyses are not always as accurate as those performed by their human counterparts. However, it is often the case that the ability to process orders of magnitude more data at a reduced accuracy is more beneficial than smaller amounts of highly accurate information. Bioinformatics encompasses many computing science and software engineering principles. For instance, distributed computing, data management, workflows, customised data storage and analysis algorithms, and e-Science platforms all play their part in enabling scaleable bioinformatics applications.

¹Moore's Law: <http://www.intel.com/technology/mooreslaw/>, accessed September 2009.

2.1 Distributed Systems

The phrase ‘distributed systems’ covers a wide array of applications and architectures. Any system involving multiple computers connected periodically or permanently to one another via a connection (such as an Ethernet network) can be considered distributed. The study of distributed systems is concerned with how the responsibility for completing a conceptual unit of work can be spread among machines and mapped to multiple cooperating operating system processes. Distributed computing covers a broad range of systems: file and data transfer; e-mail, Web and database client-server applications; computationally-oriented distributed systems; remote invocation of services such as Remote Procedure Call ([RPC](#)), Secure Shell ([SSH](#)), and Web services; and Grids. This section provides an overview of the broad range of protocols and architectures that have been developed to facilitate the development of distributed applications.

Distributed systems have several intrinsic properties which arise from the way hardware is physically and logically arranged, as well as from interactions among software processes. These properties include:

- Inherently concurrent in nature: each node is an independent computer with at least one CPU [[295](#), p. 2].
- Inter-Process Communication ([IPC](#)) suffers from increased latency due to message protocol processing overheads and as the physical distance data transferred over is increased [[295](#), p. 7].
- Communication latency has implications for time-stamping events: every node maintains its own clock; there is no inherent concept of ‘global time’ in a distributed system [[295](#), p. 11].
- Reliability: depending upon the architecture employed by a particular distributed system implementation, a distributed approach may either provide greater reliability, or reduced reliability as the number of nodes is increased.
- Heterogeneity: distributed systems may potentially be composed of a broad range of hardware and operating system platforms, ranging from embedded devices to super computers.

The concurrent nature of a distributed system, coupled with unpredictable communication delays pose significant problems when coordinating multiple distributed processes. The absence of a global clock for synchronising distributed nodes means that although a node knows its own state, it cannot know the state of the other nodes at *exactly* the same time; only a partial ordering of events can be

achieved [179, 50]. This uncertainty has important implications which must be considered when implementing functionality such as database transactions, process synchronisation, and deadlock handling within distributed systems.

Distributed applications may either be more or less reliable than centralised applications, depending on how failures are handled. A single machine or network connection failure, if handled badly, can result in the failure of the entire system. Therefore, component failure needs to be handled effectively when designing reliable distributed applications in order to present a seamless service to the end user and to ease system maintenance for administrators [295, p. 4-7]. [157] defines a set of guidelines that specify how a distributed system should respond to environmental changes, such as hardware failures and configuration updates. It suggests that distributed applications should employ various types of abstraction to mask the negative properties of process distribution as much as possible from end users. For instance, ISO suggests failure transparency through the use of transactions, check-pointing and hot fail-over replication [157]; location transparency through DNS; and migration transparency portable programs and staging of data prior to switching servers.

2.1.1 Architectures

Nodes participating in a distributed system have a logical arrangement determined by the patterns of connectivity between processes executing on different machines. Physical locations of nodes and hardware interconnections between nodes may differ from the logical layout. Together, the choice of the logical and physical arrangement of distributed nodes determine the properties of the whole system in terms of efficiency, scalability, and resilience to component failures. Several categories of well-known architectures are discussed later in this section.

Although the choice of logical architecture for a system is largely independent of the physical implementation, some logical architectures are better suited to certain types of physical configuration. For instance, closely-situated nodes with high-speed connections are likely to have lower communication latencies and higher bandwidth capabilities than geographically distant nodes communicating via the Internet. Therefore, a distributed application requiring a set of tightly-coupled processes is likely to perform better when deployed to a set of machines that are located in the same facility.

2.1.1.1 Client-server architectures

There are two conceptual components in a client-server architecture. A `server` is a software component that can expose a range of `services` such as access to data, computational power or

brokering facilities via a network connection to one or more clients. A service may be provided by more than one server component, operating on a distributed set of computers. A `client` is a software process that consumes the provided service. Client processes need not be connected via a permanent network link, unlike server processes that must be constantly available in order to service requests. Clients typically run on different physical hardware to the `server` processes they communicate with [63, p. 8]. In many distributed systems, client instances greatly outnumber server instances.

Client-server architectures are highly prevalent in networked computer systems [83, pp. 3-5]. Applications range from network file-system protocols and database services operating across a Local Area Network (`LAN`), to email, Web, and Web services operating across the Internet. The term ‘client-server architecture’ covers a range of sub-architectures that consist of client and server components in different arrangements. These include 2-tier, 3-tier, and n -tier variants.²

Multi-tier architectures are a variant on the basic client-server approach. A software component that plays the role of a `server` to `client` components may itself be a `client` to a different `server` process. For instance, a common example of a 3-tier system is the relation between web browser, web server and database components. A web server plays the role of a server when providing HTML pages to browsers. The web server plays the role of a client if the content for the HTML pages must be retrieved from a database.

Another variation is the ratio of computational work performed by a server-side process and the ratio of computational work performed by the client process. Applications that utilise local resources of the client computer, such as disk storage space or large amounts of CPU power are known as `thick clients`. Post Office Protocol (`POP`) mail clients and some types of distributed computing clients fit into this category [10].

Applications that perform a minimal amount of computational work on the client computer are termed `thin clients` [63, p. 40]. Thin clients are useful when the bulk of the processing must occur elsewhere for practical, security, or convenience reasons. A thin client might display the results of a remotely-running software application. Systems such as the X11 windowing system and VNC make use of a thin layer to display the content of a remotely running application. Such systems enable data- or CPU-intensive applications to be run remotely on a highly-specified server, but be controlled by one or more relatively cheap client computers.

²Client/Server: Past, Present and Future, George Schussel, 1996, <http://www.dciexpo.com/geos/dbsejava.htm> (accessed October 2008)

Peer to peer architectures

In a client-server system, servers are typically permanently connected to a network in order to be ready for incoming requests from clients. In a peer-to-peer (P2P) environment constant availability is often infeasible, especially in cases where the a P2P network is composed of peers using unreliable connections, such as home computers. The emphasis in a P2P system is on ensuring that *enough* peers are providing the required service all of the time, rather than ensuring high availability for any individual peer [28].

There are several conflicting definitions [14] of what constitutes a P2P system. At one extreme, highly structured, centralised systems which utilise the resources exposed by Internet-connected ‘peers’ have been described as P2P systems [298, p. 29] even though there is no direct communication between peers. An example of this would be the Seti@Home [154] project. At the other extreme, unstructured and decentralised ‘server-less’ protocols such as Gnutella [104] are considered by purists to be ‘true’ P2P systems. There are a number of intermediate levels of distribution, such as partially centralised systems and ‘brokered’ systems where peer discovery is centralised, but data transfers are decentralised [298, p. 29].

In a P2P system, distinction between client and server is blurred since nodes typically play the role of both. Therefore, each service consumer node (client) may also be a provider of the same service to other nodes. Peers in a system provide a service directly to other peers, without a need for an intermediate dedicated server. Typically, the emphasis in a P2P system is the symmetry between peers, and their equality in the system [298, pp. 23-24].

Another property exhibited by some P2P systems is the ability to function without a centralised architecture. P2P systems often construct and manage their own overlay network, a logical network that operates above the physical network layer [298, pp. 35-36]. The overlay network determines which peers communicate with other peers, and therefore how messages are routed within the system.

Operating in a decentralised fashion presents several challenges to the reliability of a system including: locating existing peers in the network; guaranteeing a search or messaging operation reaches all intended peers; placing newly-joined peers in the overlay network; and handling the removal of peers. To overcome these issues, a P2P system operating without a rigid centralised structure must support the self-organisation and self-maintenance of the overlay network. Maintenance operations typically include ensuring that peers within the network are suitably well-connected so that a node removal does not cause the formation of a separate sub-graph, but not so overly connected that the

bandwidth overhead of maintaining accurate peer lists overburdens nodes [260].

P2P architectures are usually more difficult to implement than client-server approaches for a variety of technical and organisational reasons including navigating firewalls and maintaining overlay networks [298, pp. 31-36][28, pp. 10-11]. However, where it is appropriate for a system to use a P2P architecture, advantages often include greater scalability and dynamic load balancing since network bandwidth or computing power can be contributed by individual nodes. Common uses of P2P systems include decentralised searching, messaging, and data transfer where there are many clients, many data items or both.

Pastry [260] is a generic P2P object location system. It is a completely decentralised, self-organising overlay network designed for use in large-scale P2P systems (hundreds of thousands of nodes). Pastry provides the infrastructure on which distributed applications can be built.

Each node in a Pastry network is assigned a unique (numeric) identifier. Each node maintains a list of other nodes. Messages are passed between nodes until arrival at the intended destination node. When a particular node is asked to deliver a message to a target node, it checks its list of known nodes to see if the destination node is one of them. If it is, the message can be delivered. If not, the node will attempt to forward the message to a known node that is numerically closest to the target destination. The authors say that using this approach, message routing in Pastry typically requires $O(\log N)$ routing steps, where N is the number of network nodes.

Pastry attempts to minimise the number of network hops to deliver messages to particular nodes by making use of network locality information. Nodes that are local to a particular network prefer to communicate with other local nodes.

Pastry has been used as the routing layer in several P2P applications, including a distributed file system, PAST [76], and a distributed message-passing system [49].

Gnutella Gnutella [104] is a P2P protocol supporting distributed search, retrieval and publication of file-based resources among networked hosts. The Gnutella network provides a P2P overlay network infrastructure that enables participating peers (termed ‘servants’) to discover other servants hosting resources of interest.

On initial start-up, a peer does not have any information about the Gnutella network, other than a small set of well-known hosts termed ‘host caches’. A host cache may be contacted in order to obtain an initial subset of Gnutella peers. The list of ‘neighbouring’ peers is then kept up-to-date via peer-to-peer interactions, without the need for subsequent contact with one of the central host caches.

A query for a file resource may arrive at a peer from one of its neighbouring peers. On receipt of a query, the peer will first attempt to satisfy the request itself. If that is not possible, for instance if the requested resource is not present locally, then the query will be forwarded to the remaining set of immediate neighbours. Each neighbour will then repeat the process until either a suitable peer is found, or all peers have been queried.

To participate in the network, a peer must run a Gnutella client (such as Limewire [198]). The client uses the Gnutella protocol to perform distributed discovery, but also includes a HTTP client and server. Once a remote peer of interest has been discovered, a HTTP transfer is initiated directly between the two peers. Since the P2P portion of the Gnutella network is used for resolving peers rather than large data transfers, it does not become overwhelmed with content traffic. The network is therefore highly scalable.

Skype Skype³ is a global VoIP telephony network that is part client-server and part P2P. A central server is used to store personal user details, to perform authentication, and to ensure global nickname uniqueness. A P2P network is used to allow scalable distributed user searching, and to facilitate communication between peers behind restrictive firewalls. The Skype network is self-organising, where nodes can choose to be ‘standard’ nodes or ‘super’ nodes, depending on their environment. Standard nodes may be promoted to super nodes if they have adequate CPU power and network bandwidth. Super nodes act as hubs for standard nodes and may route traffic if two communicating standard nodes are behind firewalls [22].

A range of different architectures can be considered to be P2P to some extent. P2P systems have been defined as being centralised, partially centralised, and decentralised [14]. Self-organising systems such as Skype take advantage of a partially centralised architecture to facilitate communication between less-well connected nodes. Partial centralisation can help to reduce the time required to search for a resource, since fewer network hops are required. However, this strategy comes at the cost of increased network and computational load for ‘hub’ nodes. Hub failures may have a negative impact on many ‘ordinary’ nodes.

Completely decentralised architectures are not reliant on any centralised infrastructure. They are therefore inherently scalable in terms of the number of nodes that can be supported by a system. However, the lack of any overlay network structure requires queries to be propagated throughout the entire network, i.e., by message flooding. Care must be taken to ensure that loops of interconnected nodes are not formed that would result in perpetual message forwarding. One approach is to introduce

³<http://www.skype.com>, accessed November 2008

a Time To Live (**TTL**) for each message sent through the system. A **TTL** is a maximal limit on the distance (in terms of network hops) that a message is allowed to travel before it is dropped by a node. However, as pointed out in [14], while this solves the flooding problem, it introduces a ‘message horizon’ that potentially prevents a node from ever receiving a message if it is ‘too far’ from the originating node.

Strictly speaking, no large scale **P2P** is *truly* decentralised, since the bootstrap process always requires an initial set of peer addresses to be acquired from a (set of) well-known location(s). In the context of a **LAN** it would be possible for nodes to discover each other in a decentralised fashion through message broadcasts. However, this is not possible with Internet-scale networks. Once running, many **P2P** systems are capable of self-organising and performing self-maintenance to the overlay network, and require no centralised infrastructure.

For the purposes of this thesis, **P2P** is defined to be a system in which participating peers perform at least *some* communication directly with each other. Therefore, we regard a **P2P** system as one which may either be an entirely decentralised system, or a system which has some degree of structure in the form of centralised servers of ‘hubs’. It does not matter whether these centralised hubs are architected ‘by design’, or whether they are a result of a dynamic reconfiguration of the overlay network.

Mobile agents

Mobile agents are programs that are designed to migrate to remote computers, either manually by command of a system administrator or by means of an automated process. [156] defines a mobile agent as consisting of: agent program code; a thread of execution, and associated execution stack; a unit of data. All constituent parts accompany the agent as it moves between physical locations. The data part is mutable, and reflects changes in the ongoing computation. Mobile programs have a wide range of uses including: utilising the idle time of remote CPUs [117]; automated load-balancing of server applications; and increasing the fault-tolerance of systems [141]. The ability of a software agent to move to a machine to which it was not initially installed facilitates more efficient use of computational resources. For instance, if a computer with hardware specifications more suited to an agent’s computational task becomes available after initial software deployment, then the software agent can migrate to the new environment dynamically.

In large distributed computation systems it is necessary to handle individual node failures gracefully. One approach to achieving this behaviour with the use of mobile agents is outlined in [117]. A mobile agent is responsible for deploying itself to the remote computer. Frequent progress checkpoints are

made so that if the machine fails or becomes unavailable, the computational task can be migrated to a different computer. On migration, work can resume from the last checkpoint, minimising the ‘wasted’ work time. Dynamic installation support for domain-specific software also eases the burden on system administrators.

In bioinformatics, mobile agents have been used for a number of purposes including data mining [274], genomic annotation [70], tool and data integration [61]. Agent toolkits such as BioAgent [215] have demonstrated the ability for the modular nature of agents to permit extensibility of an application through the addition of new agents to the system.

2.2 High-throughput computing

The demand for computational power in data-intensive research has always outstripped supply. Performance increases in the latest generations of computer hardware can always be consumed by running more complex analyses not possible with the previous generation of hardware, executing applications with larger data sets, or re-running existing analyses at higher resolutions. In contrast to Amdahl’s law [8], Gustafson’s law [136] states that given a suitably large computational problem, it should be possible to parallelise the problem to fit the number of available processors. Gustafson’s law is a good match for exhaustive computational analyses in bioinformatics since the continual production of new data and analysis techniques provide sufficiently large quantities of work for parallel processing to greatly improve the rate at which analyses can be performed.

For a given generation of computer hardware, many computational problems are too large for a single computer to manage, whether the limitation is due to inadequate CPU power, memory, or permanent disk storage. While increases in individual component speeds have been rapid and continuous, the scale of some computational tasks are extremely large. Even with the fastest CPUs available today, completion could take many days, months or even years when executed on a single machine [184].

Therefore, it is necessary to consider the use of multiple compute resources with many threads of execution operating concurrently, with each thread working on a small part of the overall problem. A compute task that takes several months to process on a single computer could be completed in a much shorter space of time if it could be broken into smaller, more manageable chunks and handled by multiple computers. While it is generally much harder to write and test parallel applications, this disadvantage is greatly outweighed by the speed-up that can be achieved by adding more processing units. Parallel processing can be achieved either by building larger computers containing more CPUs (multi-processing), or by utilising a set of smaller, interconnected computers (distributed processing).

There are typically three competing factors that influence the implementation of a parallel system: the target hardware architecture, the software architecture, and the amenability of a particular computational problem to parallelisation. To some extent, the properties of the underlying hardware dictate the appropriate software approach, although various abstraction techniques permit different software paradigms to be used on regardless of the underlying hardware, sometimes at the cost of efficiency. A computational problem may be parallelised in such a way that there is an obvious choice of hardware architecture to execute it. Alternatively, a faster than serial, but sub-optimal parallel implementation may be required if existing hardware must be used.

2.2.1 Programming models and scalable parallel computing

The major challenge in writing a parallel application is how a large computational task can be split into units that can be efficiently processed by multiple processing units. It is essential that sequential consistency is maintained [180], that is, the parallel version of a program produces the same computational result as the sequential version. In general, computational problems fit into three categories: those that are embarrassingly parallel, operations that may be parallelised to some extent, and operations that it is not possible to parallelise without changing the result of the computation. Meanwhile, computer operations can exploit parallelism at several levels: the use of fine-grained specialised CPU instructions; multi-threaded programs; and multi-process distributed applications. There are several well-known strategies that can be used to break a large task into more manageable blocks. The most suitable task-splitting strategy to use for a particular application is influenced by the structure of the data, the type of processing required, and the hardware available to execute the computation. Choosing an unsuitable strategy usually has adverse effects on efficiency and execution time, rather than computational correctness.

Parallel processing can be achieved by exploiting multiple instruction sequences, multiple data streams, or a combination. Flynn's taxonomy [99] conveniently provides four categories for classifying data processing operations:

- Single Instruction Single Datastream (**SISD**): a sequential set of operations applied to a single data stream
- Multiple Instruction Single Datastream (**MISD**): pipeline processing of a single data stream
- Single Instruction Multiple Datastream (**SIMD**): application of the same operation to multiple data streams

- Multiple Instruction Multiple Datastream (**MIMD**): application of multiple operations (pipelines) to multiple data streams

SIMD involves applying the same processing step to multiple instances of data. Examples of **SIMD** approaches to parallelism can commonly be found in the processing instructions of CPUs intended for multimedia operations [189], such as Streaming SIMD Extensions (**SSE**) instructions⁴, where identical operations must be applied to large numbers of data items. In contrast, the **MIMD** approach involves a cooperating set of threads working on the same problem, with each thread potentially having its own independent instruction sequence. In the context of a hardware architectures, **MIMD** tends to indicate more complexity in the threads operating over a data set than **SIMD**. While the terms **SIMD** and **MIMD** are usually used to describe low-level hardware architectures or software implementations, a cluster of computers executing different instances of the same program over different data sets can be thought of as a very coarse-grained **SIMD** architecture. A more appropriate description for this type of system is Single Program Multiple Datastream (**SPMD**). **SPMD** implies independent processes, rather than tightly-coupled processes executing in a lock-step fashion. Individual computers running as part of an **SPMD** cluster may also exploit low-level machine-local parallelism using either **SIMD** or **MIMD** techniques, or a combination of the two if each program instance is running on a multi-CPU machine.

2.2.2 High-throughput computing platforms

Historically, designs of parallel computing hardware implementations have been highly specialised, heavily dictating the way in which software was written. A number of architectures have been proposed including vector processing and systolic arrays. In many cases, the programming languages have been tightly-coupled to the computer hardware. In other cases, the form of parallel processing used (as discussed above) is heavily influenced by the underlying hardware capabilities. For instance, vector processors such as the Cray were ideally suited for computational tasks that are easily vectorised and involve large amounts of data, such as computational fluid dynamics, physics and weather forecasting. These machines were less well suited for situations involving more complex CPU instructions or small amounts of data due to their architectures being highly optimised for vector processing.

Processors in modern High Performance Computing (**HPC**) hardware are typically commodity, scalar or super-scalar processors. While these processors are best suited for **MIMD**-type operations, they

⁴<http://www.intel.com/technology/architecture-silicon/sse4-instructions/index.htm>, accessed November 2008

also implement **SIMD**-style instructions that can be used for vector processing. The attention of modern **HPC** architectures is focused more towards general purpose parallelisation of higher level thread or application-level parallelism, rather than instruction-level **SIMD** operations.

- Shared memory: a single computer with multiple CPUs sharing a common memory unit.
- Non-Uniform Memory Access (**NUMA**) shared memory: a single computer with multiple CPUs. Each CPU has its own block of memory.
- Distributed shared memory: multiple network-connected sequential or parallel computers, each with their own memories. The combined memories may be presented to the programmer as a single, large virtual unit of memory.
- Distributed, non-shared memory distributed processing: multiple network-connected sequential or parallel computers operate either cooperatively or individually. Messages may be passed between cooperating machines, but each computer maintains its own individual address space.

A number of factors dictate the suitability of a parallel processing architecture for a given computational problem. These include: the frequency at which threads need to communicate, the frequency at which threads need to synchronise with each other, and the amount of data that needs to be transferred between threads.

2.2.2.1 Shared memory parallel computing

The term ‘shared memory’ refers both to a software development paradigm, and a set of hardware architectures. Typically, ‘shared memory’ refers to the programming model of multiple threads sharing the same memory address space for communication purposes. Although some hardware architectures are ‘true’ shared memory implementations, where several CPUs share the same physical units of memory, large modern systems actually implement a virtual shared memory in order to facilitate a convenient programming model.

Shared memory parallel processing involves running multiple cooperating, concurrent, threads of execution, often in a low-latency environment such as a single, large computer with multiple CPUs or a cluster of computers with fast interconnects such as gigabit Ethernet or Infiniband [17, 137, 160]. The various threads typically work on different parts of the same in-memory problem in a tightly-coupled fashion. Communication and data transfer between threads is facilitated by accessing a pool of memory shared between the available CPUs, while flow control and synchronisation is facilitated

by standard concurrent programming constructs such as semaphores, monitors, barriers and critical sections [145, 242]. Parallel processing techniques are suitable for many types of computational problems involving large-scale numerical analyses such as matrix calculations [321, p. 301-303], solving linear equations [321, p. 313], parallel genetic algorithms [170] and large scale data mining [336].

Multi-core CPUs and Symmetric Multi-processor (SMP) hardware are modern examples of shared memory parallel machines. Dual and quad-core CPUs in particular are now commodity items available to desktop users.

Domain-specific hardware is another class of device that utilises a shared memory architecture. Specialised hardware devices are designed to perform a particular, specific task very quickly. These devices are typically SIMD hardware implementations of a domain-specific application or algorithm although some may be suited to more general purpose computation. For instance in bioinformatics, the Smith-Waterman [280] and Basic Local Alignment Search Tool (BLAST) [6] algorithms have complete or partial Field Programmable Gate Array (FPGA) implementations [329, 330, 186, 140]. Although these devices are only capable of running specific algorithms, the increase in performance is substantial — sometimes one or even two orders of magnitude over a typical desktop CPU. While achieving much higher speeds, Harris et al. [140] claim that their FPGA implementation still manages to be 99% as sensitive as the standard software implementation.

A more recent development currently being investigated is the use of Graphic Processor Unit (GPU)s to speed up SIMD-type operations. The latest graphics processors are becoming increasingly programmable. Graphics processors contain multiple processing pipelines that are able to process streaming floating point data extremely efficiently. Computationally intensive domain-specific algorithms from many fields of research have been adapted to execute on GPUs including bioinformatics [196, 178], particle simulations [331] and cryptography [334]. GPU have also been used more generically as mathematics co-processors [120, 39]. Some implementations have been shown to be considerably faster than equivalent implementations using general-purpose CPUs. As graphics cards become more powerful and the amount of on-board memory becomes greater, they become more attractive as co-processor units for suitable tasks. One of the factors limiting their uptake has been the vendor and even card-specific programming required, as well as the need to express problems in terms of graphics constructs. Work is currently underway to develop more general languages that allow access to the hardware-accelerated functions that these cards provide [40, 296, 60].

In [326], Wirawan et al. have successfully executed several sequence alignment algorithms on commodity games console hardware equipped with a Cell [159] processor. The Cell architecture shares

strong vector processing characteristics with graphics processors and therefore allows similar speed ups for suitably-written software. Future exploitation of Cell-like architectures for scientific applications looks likely if computational problems can be reformulated to take advantage of vector processing techniques [327].

As the number of CPUs within a machine is scaled up, contention between multiple CPUs and memory modules becomes greater. While bus and memory contention can be mitigated to some extent with the use of more complex hardware [148, 79] and careful software design [257], eventually adding more CPUs becomes detrimental to performance [149].

2.2.2.2 Distributed parallel computing

As contention between numerous components of large single parallel computers became too problematic, clusters composed of multiple, network-connected computers were seen as an alternative. Distributed parallel computing addresses the issue of bus contention between internal components, enabling much larger systems to be constructed; each node is an individual, independent computer running independent processes. Distributed shared-memory computer systems are relatively cheap to construct, especially if they are composed of standard high density rack-mounted blade servers or Common Off The Shelf (COTS) desktop PC hardware. There are two major distributed parallel computing paradigms: distributed shared memory and message passing. Distributed shared memory allows distributed processes to address the same virtual memory space, permitting synchronisation and data transfer operations. The message passing approach provides each process with its own independent memory space, requiring messages to be sent between them when data transfers or synchronisation is required.

Distributed shared-memory computing provides an illusion of a single shared-memory computer to the application programmer, when in fact threads and memory contents may be spread among a cluster of computers. Several libraries have been created to ease development of parallel applications that must execute over a large number of discrete nodes [292, 262, 293, 75]. These frameworks hide the details of the actual hardware behind a layer of abstraction, presenting the application programmer with environments containing virtualised components, and access to resources such as Distributed Shared Memory (DSM) [187, 47]. Using the DSM model, parallel programs can be built in an abstract fashion, without requiring knowledge regarding whether they are running on a single large multi-computer, or distributed across multiple nodes of a cluster. With distributed parallel machines, delays caused by contention for the bus between CPUs and memory is replaced by contention and

latency of the network connection between the nodes, as well as additional CPU overhead involved in processing network messages.

Accessing the contents of memory stored on a remote node is much slower than accessing local memory. This effect is termed Non-Uniform Memory Access (NUMA). Although multi-processor machines with several memories also suffer from NUMA, the effect is much more pronounced when using distributed clusters of machines, due to the greater latencies involved. Since the DSM is presented to the application as a single virtual address space, the application is not necessarily aware of which memory segments are located locally, and which are located on remote machines. If memory accesses are frequent and spread over a large portion of the address space some types of application may suffer performance problems [87], although other studies have shown that networking overhead is not a bottleneck for all applications [30].

The effects of NUMA can be mitigated to some extent via intelligent page placement techniques [36]. Various software and hardware-based replication techniques have been proposed to cache frequently-used data locally to a node, while maintaining consistency in the case of multiple write operations by distributed processes [243, 169, 77, 64]. Other techniques involve pre-fetching data before it is required, and the use of multiple threads to ensure that CPU utilisation is high even when one or more threads are blocked, awaiting an Input/Output (I/O) operation [224].

2.2.2.3 Distributed high-throughput computing

Non-shared memory distributed computing typically involves loosely-coupled distributed processes that periodically communicate with a supervisor process, or synchronise with each other. IPC and synchronisation may be achieved in several ways, including:

- Client-server: multiple client worker nodes contact a central server to acquire work and to perform synchronisation processes.
- Peer to peer: distributed process communicate and co-ordinate with each other directly.
- Database-centric: a variation on the client-server approach, where a transactions within a centralised database are used to co-ordinate processes. The database is used as a kind of transactional shared memory.

The highly-specialised nature of early parallel machines made it difficult to port programs among the many vendor-specific distributed computing platforms. Application Programming Interfaces (APIs)

such as MPI [101] were developed in order to facilitate code portability between the parallel hardware produced by different vendors.

While the parallel processing approaches discussed in the previous sections provide application programmers with a convenient development environment, the paradigm is not without its disadvantages. Performance can suffer if compute nodes need to transfer either large amounts of information over a network connection, or send large numbers of IPC messages [291]. Parallel virtual machines are also susceptible to node failures. Since the system is effectively running multiple parts of a single program, it is possible that unreliable nodes can cause the entire system fail if no error checking or redundancy is in place. There have been several proposed approaches to increasing the reliability of parallel systems, including redundant threads and checkpointing [282].

2.2.2.4 Distributed Computing

Like parallel programs, distributed computing also involves splitting a large compute task into smaller more manageable units. However, instead of multiple threads running within a single machine (or distributed virtual machine), each unit of work executes as a separate process within physically and logically distinct hardware. Distributed processes may communicate with each other, but IPC tends to be for process synchronisation purposes, rather than large or frequent data transfers since there is no “shared memory” model. In the main, distributed processes tend work in isolation from one another in order to avoid overheads due to network latency or bandwidth contention [192, 154, 9].

Distributed programs can be written either with or without explicit knowledge of their existence within a distributed environment. There are multiple frameworks that enable application programmers to write their software specifically to take advantage of distributed operations such as IPC, object marshalling and data transfers as well as thread synchronisation using RPC-style methods: Common Object Request Broker Architecture (CORBA) [309], Distributed Component Object Model (DCOM) [261], Java Remote Method Invocation (RMI) [217]. On the other hand, it is also possible to run ordinary, serial programs within a distributed environment with no use of specialised APIs. Multiple copies of serial programs can be run with different data sets across large numbers of compute nodes, i.e., program-level parallelism (SPMD). In this case, synchronisation between processes and data collection is generally performed at the end of process execution by a distributed environment framework.

Distributed applications can be run either in homogeneous clusters, built from high-performance compute nodes, or across a set of heterogeneous (in terms of hardware and/or operating system)

nodes, or even a combination of the two.

Typically a large organisation such as a company or university will have many hundreds or thousands of desktop computers. These machines are located either in personal offices, or larger numbers are provided in communal cluster rooms. It has long been known [227] that numerous workstations will spend much of their time idle, particularly in the evenings and weekends - but also during ordinary working hours. As personal desktop workstations become more powerful, it becomes increasingly desirable to utilise their computational capabilities during their idle time [12].

Although distributed computing using otherwise-idle processing capabilities is more challenging than using a parallel multi-processor server, there are significant economical advantages to doing so. Powerful server equipment is expensive to acquire and often difficult to maintain. In addition, the sheer number of available desktop computers installed on a typical university campus — numbering in the thousands — provides a pool of raw computing power to rival that of a high-end dedicated server cluster. The typical rolling hardware-refresh cycle for a university is in the region of 3-4 years. When desktop computer equipment is upgraded, distributed compute processes will also benefit, with no additional cost.

Distributed computing has been common in multiple forms for many years, and used in many different disciplines. As such there are a wide variety of approaches to utilising remote compute resources ranging from remote shell execution, to batch processing systems [128, 58] and more advanced frameworks [192, 218, 105, 133] that deal with load balancing, and process checkpointing. One of the main problems facing the basic remote-execution approach to distributed computation is reliability. As the number of computers in the processing pool increases, the chance of a hardware failure, or that a user terminates the running processes increases. Using the remote-execution or batch-processing approaches is difficult for large-scale or long-running jobs under these circumstances, and using a framework like Condor [192] might be a more sensible option in these cases.

Each type of system still has its uses, however. For example, even though the newer, feature-rich systems such as Condor are clearly more advanced than a simple shell script or batch processing system, they are also more complex to set up and use. In the case of Condor, there is a requirement for installing and configuring job submission and queue services prior to running computational jobs. For small-scale prototyping or one-off processing, a shell script or batch processor may be easier or more convenient for the end-user, and would still provide acceptable performance and reliability.

With the rise in popularity of the Internet during the mid-1990's, it was realised that there was a potentially massive amount of compute resource available to be tapped in the form of personal com-

puters sitting in homes throughout the world. Seti@Home [154] was the first large-scale distributed system to utilise so-called “public computing” or “volunteer computing”. It consisted of a central set of servers charged with the tasks of: storing recorded telescope data; splitting recorded signal data into 250kb chunks termed “work units”, distributing work units to computers across the Internet; and storing/collating the results of detected “interesting” signal spikes for further analysis. The project soon acquired enough computers donating processing time — over one million — that the project received computational power twice that of the fastest supercomputer available at the time [9]. This demonstrated that volunteer computing could potentially play a role in assisting scientific projects short of computing power. The size of the work-units sent to Internet-connected machines mattered greatly, due to the slow and expensive (dial-up) Internet connections predominantly used by home users at the time. The success of the project hinged on the fact that it took much longer (hours) to process a work-unit than it did to download the raw data, and upload results (minutes).

Following the success of the Seti@Home project, several other public distributed computing projects were developed for various domains [273, 74, 208]. Each new project coded their analysis algorithms directly into their own compute client. If people wanted to participate in multiple projects they were required to download multiple client programs, each with its own binaries, configuration, and platform requirements.

The Berkeley Open Infrastructure for Network Computing (BOINC) [9] project was proposed as a solution to this problem. BOINC is a small generic client application designed to support multiple distributed compute projects. After installing the BOINC client, the user must register themselves with the distributed compute project(s) they wish to donate their CPU time to. The BOINC client then downloads the required executable and data files from a particular project’s central server. The use of a single client allows the user control over how CPU time is prioritised between different computing projects. In the event that a particular project has no work currently to process, the BOINC client allows another registered project to process instead.

In practice, it is possible for high-throughput systems to utilise a hybrid approach combining the advantages of parallel and distributed models. With the recent advent of multi-threaded and multi-core CPUs, most workstations are capable of running two or more tasks concurrently. To take advantage of this, either multiple single-threaded jobs can be sent to each workstation, or individual compute jobs can take advantage of parallel processing techniques.

2.2.3 Summary

Properties of parallel processing platforms			
Architecture	Scalability (CPUs)	Task suitability	Limitations
Shared memory (SMP)	10s	Tightly-coupled threads Frequent IPC	Memory contention Bus contention
NUMA shared memory (Massively Parallel Processing (MPP))	100s	Tightly-coupled threads Frequent synchronisation	Bus contention
Distributed shared memory (MPP)	1000s- 10000s	Loosely-coupled threads Frequent synchronisation	Network latency Network bandwidth
Distributed, non-shared memory (clusters)	10000s- 100000s	Largely independent tasks Infrequent or no IPC Infrequent synchronisation	Network bandwidth Data distribution

As of 2008, the majority (80%) of the top 500 supercomputers are cluster-based, rather than MPP [301].

2.3 Data transfer protocols

Data transfers between networked computers can be achieved in different ways, both in terms of conceptual differences as well as different architectural and implementation approaches. Although any communications between distributed machines can be regarded as ‘data transfer’. Here, the phrase ‘data transfer’ is used to describe communication between machines for the purpose of information exchange, as opposed to communications for synchronisation purposes or for initiating remote execution via RPC-like protocols. In a typical scenario, data may be located on one computer, but must be processed on another. In order for the second computer to be able to process the data, the data must be accessible to it.

Client-server transfer protocols consist of a server process and a client process, usually running on different machines. Client and server implementations are protocol-specific. The File Transfer Protocol (FTP) [29] [245] protocol is one of the earliest and most used file transfer mechanisms within intranets and on the Internet. Other examples of widely-used client-server file transfer protocols include Hypertext Transfer Protocol (HTTP) [92], WebDav [123] and Secure Copy (SCP), a file transfer protocol that uses encryption provided by SSH [335].

Client-server transfer protocols are inherently centralised, and as such server processes can suffer from scalability problems when under load from large numbers of client processes. Standard load-balancing or protocol-specific caching techniques can be employed to improve the ability of a system to perform better under heavy loads by making the same content available in multiple locations [263].

In contrast to file transfer protocols, network file systems support standard file system operations over a network to remotely stored data. Network file systems such as Common Internet File System (CIFS) and Network File System (NFS) have a number of advantages compared to file transfer protocols [219, 181]. By placing network transfer operations behind a file system view, most existing applications will work seamlessly without modification. Random access to files content is also supported, permitting applications to start reading from the middle of files as opposed to acquiring the entire file, as is necessary with some file transfer protocols. Updates to remote files are also possible without having to transfer the entire file. As with other client-server systems, server-based network file systems suffer from reduced performance when many client processes perform I/O operations [13]. Server replication, client-side caching and more intelligent client requests have been proposed to address this limitation [213, 20, 78].

2.3.1 Peer to peer, global-scale file transfer protocols and file systems

A plethora of distributed transfer protocols [57, 104] and file systems have emerged over the past decade that distribute the responsibilities traditionally associated with server processes across peers participating in the system [281, 174, 76, 319, 88]. These distributed systems have minimal or no central server requirements and therefore improve scalability by supporting much larger numbers of nodes [13, 319].

PAST [76] is a P2P storage system intended to utilise the collective disk capacities of Internet-connected nodes in a self-organising fashion. PAST supports automatic replication of files for increased reliability and distribution performance. Nodes participating in a PAST network are not required to be ‘high availability’ dedicated machines, and may leave the network at any time with no adverse effect on file availability. The authors claim that storage utilisation can approach 100% even though no central control system exists.

As distributed systems become larger, it becomes more difficult to enforce synchronised actions across all nodes, especially if nodes are unreliable or are managed by different administrative entities. As a result, to avoid restrictive contracts that would necessitate ‘agreement protocols’ among nodes, PAST implements weaker semantics on file-system operations than are routinely expected from local or LAN-based file-systems. Important properties of the PAST system include:

- Immutability of published data: once a file identifier has been used, it cannot be re-used for consistency reasons.

- PAST supports ‘reclaiming’ rather than ‘deletion’ of files: removal of data from the system reclaims disk space from nodes, but does not necessarily remove the content from all nodes. That is, once data has been published, there is no guarantee that it can be unpublished.

2.3.1.1 BitTorrent

BitTorrent [57] is a popular P2P file distribution system widely used for the efficient distribution of large software packages across the Internet including Linux distributions and video game patches. Peers that are downloading the same file co-operate with each other by transferring parts of the file amongst themselves, rather than relying on a central server to upload files to all the peers.

Files are initially made available by publishing a `torrent` file. A `torrent` contains a hash of the file content, as well as other data such as file names, lengths, and the Uniform Resource Locator (URL) of a tracker [57]. `Torrent` files are lightweight pointers to the actual content, and can therefore be disseminated via a standard HTTP server, or other means, such as email.

A `seeder` is a peer that has an entire copy of a file and is currently `seeding` (uploading) to other peers. A `leecher` is a peer which currently has an incomplete copy of the file. While leechers are attempting to acquire the entire file, they also upload the portions of the file they already have to other peers. A `tracker` is a server that keeps track of available files and peers. Peers periodically contact the tracker to exchange information about other peers that are currently seeding or leeching the file. Contacting the server in this way is termed `scraping`⁵. Peers may periodically `scrape` the server in order to determine a) while downloading, whether it is worth sending a request to receive details of new peers, and b) which files to actively seed based on the current number of seeds reported by the tracker.

The original BitTorrent protocol was a hybrid-decentralised [14] P2P architecture, requiring a centralised tracking system to facilitate P2P bulk data transfers. Several improvements to the BitTorrent protocol have since been made to decentralise the ‘directory service’ role played by torrent tracker servers. Several BitTorrent clients have Distributed Hash Table (DHT) implementations that aid the discovery of file content. A DHT stores a subset of key/value pairs on each node participating in the network [223]. DHTs have been used to store peer information, forming a de-centralised distributed tracking system [249]. Removing the need for a centralised tracker improves scalability and reliability by removing a single point of failure and contention.

⁵<http://azureuswiki.com/index.php/Scrape>, accessed 2009/04/20.

2.3.2 Summary

P2P systems are ideal for the mass distribution of large data files. The total system-wide bandwidth of **P2P** systems can provide is typically far greater than the bandwidth of a set of central distribution servers. For instance, when a new version of a large popular software package is released, there is likely to be a large demand during the first few hours or days following the release. This demand may far exceed the ability of a set of centralised servers to service all requests simultaneously. Distributing these files via a **P2P** system such as BitTorrent can significantly reduce the bandwidth requirement for the distributor, while at the same time increasing the speed that consumers can download the requested file.

Properties of client-server protocols such as **FTP**:

- Low latency: it is possible to find items of interest quickly.
- Transfer rate is at best inversely proportional to the total number of resources being simultaneously transferred for a given amount of bandwidth to a server. Therefore for a computer requiring several files, it is usually more efficient to serially transfer files since parallel requests from multiple nodes will reduce performance.

Properties of BitTorrent:

- Higher latency than **FTP** or **HTTP** transfers, since there is a need to contact a tracker in order to resolve peers hosting files of interest.
- System-wide transfer rates are higher than central-server approaches when large numbers of concurrent inter-node transfers are taking place.
- Potentially the combined bandwidth of all nodes participating in a particular transfer can be utilised. As more peers obtain chunks of the file, less stress is placed on the initial seeder. The torrent protocol is designed to notice which peers it can get pieces from the fastest - automatically balancing network load.
- Therefore transferring multiple resources concurrently is probably more efficient than serially transferring files from a single server.

2.4 Technologies underlying Grid systems

Grid computing is a form of distributed computing. This section introduces Grid systems in general, and the technologies that are commonly used to implement Grid systems. The following section will describe specific Grid implementations and their intended operating environments.

Traditionally, distributed computing has focused primarily on achieving the best use of compute resources, dedicated clusters or otherwise. The phrase “Grid computing” is intended to convey the use of pervasive distributed computing resources by software applications, in much the same way as electrical appliances utilise a power grid. A grid is composed of a heterogeneous collection of distributed “devices” that provide some kind of service to other members of the system. The “Grid services” that participating devices expose might include: access to compute hardware, data storage, or data querying. In fact, components do not necessarily have to be computers in the traditional sense. For instance, remote sensor networks have been used to provide live data streams into compute grids [240, 151]. In this case, the project’s sensors have minimal computational power and so would not be considered to be part of a distributed computation system in the traditional sense. However, they are considered Grid components because they are data sources that are exposed via Grid-services, and therefore may communicate with other Grid components. The overarching aim of “the Grid” concept is to enable different types of components to work together as a set of “black boxes”, each exposing functionality via a publicly accessible set of services that can be utilised by other services [111].

Grids have many, often conflicting definitions [143, 106, 272]. This ambiguity arises partly because Grids have been deployed to many different research domains each with their own requirements [25, 100, 206, 234, 237, 277], and partly because Grid deployment environments and technologies vary considerably. A ‘Grid’ system does not conform to a specific mould. Rather, the heterogenous, collaborative and cross-institution aspects of the system are emphasised [253, 106]. Grid systems range from small systems composed of clusters of workstations, to large cross-continent collaborations of super computers [172]. Participants of a Grid do not necessarily need to be computation devices in the traditional sense. For instance, sensor nodes with small amounts of computational power may be participants of a Grid along with high-powered computer clusters [151].

2.4.1 Web services

Web services [131] are a means of exposing data or computational resources to members of a distributed system. They form a client-server distributed system; multiple clients can request data or

computation exposed by a server. Unlike traditional distributed systems technologies such as RMI [217] and CORBA [309], Web services can be used to more easily facilitate communication between Internet-connected sites where firewalls may restrict traffic flow. Web services can be thought of as ‘document-based’ computing rather than RPC or distributed objects [310]. Rather than calling a remote method, or accessing a remote ‘object’, Simple Object Access Protocol (SOAP) messages are transferred between clients and servers. The Web services specification does not specify a particular message transport mechanism, and therefore Web service implementations must rely on existing protocols for message delivery. HTTP is a commonly used delivery protocol that conveniently allows Web services to be implemented as web applications, hosted using existing web application container infrastructure. The use of HTTP facilitates cross-site communication since HTTP is commonly allowed through firewalls. However, other standard transport protocols such as Simple Mail Transfer Protocol (SMTP) could be used to deliver Web service messages if necessary [165].

Extensible Markup Language (XML)-based SOAP messages are passed between clients and servers. The use of XML for both service descriptions and message content makes Web services language- and platform- and transport-neutral. While parsing XML data structures requires greater message processing overheads than binary RMI-type messages [165], they are straightforward to parse. Therefore clients may be written in almost any language, including many scripting languages⁶.

Despite being an order of magnitude slower than RMI [165], Web services are rapidly gaining popularity in the field of bioinformatics where required data sets are distributed across multiple sites. Bioinformatics analyses often require access to and integrate data distributed over several sites. Web services permit this data to be exposed in a programmatically-accessible way, while workflows facilitate the automation of data integration and analysis tasks [286]. While Web services are becoming increasingly popular for data query and transport operations, integrating data from different sources with different semantics is still an open area of active research [283]. Ontologies permit a community to share formal definitions of data items, enabling a shared interpretation across projects [132, 16, 23]. Research in this area has focused on using ontological definitions and logical reasoning technologies in order to integrate data from different sources with varying semantics. Ontologies and semantic reasoning have also been used as a means of locating Web services of interest based on descriptions of their inputs and outputs [322].

As well as data querying operations, Web services can also be used to expose data processing services. Such services may be custom-built, or may expose existing command line tools [275].

Web services can be used in either a synchronous or asynchronous fashion. Asynchronous operation

⁶<http://www.soaplite.com/>, accessed 2009/04/27

can be useful if the Web service request will take a significant amount of time to complete, either because the request is resource-intensive, or because the request needs to join a queue. For example, a request unique identifier (UID) may be returned by an initial call to the service that can be used by a client to poll the ‘completeness’ of the requested task [38].

2.4.2 Workflows and pipelines

In bioinformatics, many analysis tools interact with plain text files of varying formats. One or more files are taken as inputs, and one or more output files are produced after processing. It is often the case that several different programs need to be chained together as part of a larger project. At each link in the chain, the output of one program needs to be fed as an input to the next. This may involve some parsing or other manipulation of the data to convert it into the format expected by the next tool in the chain [65, 214, 235]. While batch systems and frameworks such as Condor provide access to distributed CPU power, their operating models do not capture the processing often required at the intermediate steps between the executions of batches of different types of jobs. To build large analysis toolsets with distributed tools requires frameworks requires appropriate infrastructure [107].

Workflow and pipelining tools such as Taverna [236], GridFlow [44] and OpenKnowledge [68] have been developed to automate the process of obtaining data exposed through Web services, avoiding the requirement for researchers to manually ‘cut and paste’ content between sites [65].

2.4.3 Notification-based orchestration

Notification systems are essentially event-driven distributed systems [19]. Their event-driven properties are analogous to modern Graphical User Interface (GUI) programming toolkits. For example, one or more graphical components may ‘subscribe’ to the events generated by ‘publisher’ component. On activation — a mouse click, keyboard input — the event is propagated to the subscriber components.

Publisher-subscriber notification systems have long been used as a means of process co-ordination and IPC between distributed processes [33] and particularly in real-time CORBA middlewares [254]. Notification systems consist of publishers, subscribers, messages, topics, and a delivery mechanism. Publishers and subscribers are typically independent distributed processes.

Notification systems offer a number of advantages over direct communication between distributed processes:

- There is no requirement for both communicating processes to be available at the same time. If one or more subscribers are unavailable at the time of message publication, then the subscribers can retrieve awaiting messages at a later time.
- Publishing and subscribing processes need not be aware of each-other's existence. De-coupled message delivery allows flexibility in terms of dynamic subscriber registrations and de-registrations, and also in term of subscriber location updates.

Subscribers may either use a push- or pull-based model for message collection from a notification system. In the 'push' model, the notification system infrastructure actively attempts to deliver messages to a known subscriber location, such as a Web service endpoint. When using the 'pull' model, subscribers periodically poll the system for new message, much like an email client. Both push and pull models have advantages and disadvantages, and the best one to use in a particular situation depends largely on the requirements of the system being developed. For instance, polling required by the 'pull' unnecessarily wastes resources such as CPU cycles and network bandwidth when there are no messages awaiting delivery. The polled resource might become overloaded if there are multiple polling components, or the rate of polling is too frequent. In contrast, push-based systems are particularly suited to a set of asynchronous processes, and are perhaps more efficient since communication between components occurs only when necessary. However, push-based systems require that the location of the recipients be at known locations. The push-based approach is therefore better suited to situations involving static subscribers, such as those hosted on server hardware, whereas the pull-based approach is better suited for mobile agents.

Notification systems can be used in Grid systems to orchestrate services by facilitating distributed transactions, initiating bulk data transfers, requesting computation from a remote resource, or for informing remote systems of a completed action [171, 18]. When used as a trigger for large data transfers, notification systems have been likened to the control connection of the FTP protocol [285].

Client-server-based notification systems can suffer from performance problems, particularly if large messages or large numbers of messages must be sent to a numerous subscribers. Notification systems with a P2P architecture can alleviate certain scalability aspects of content delivery [164, 190], but other aspects such as assuring message delivery, and assuring message ordering are more difficult. Ensuring message logging is also more difficult in P2P systems due to the 'peer horizon', at which peers are no longer visible [14, 15, 199].

2.5 Grid architectures

2.5.1 Introduction

The concept of a “Grid” is different to different user groups, partly because the concepts and technologies have evolved over time and partly because Grid-based systems are used in so many different domains, each with their own set of requirements. Some domains are more compute-centric, while others are more data-centric. For instance, a “computational grid” might be defined as:

“A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities.” [111]

Grids used for computational modelling, such as processing data from high-energy physics projects fit into the above category [100]. On the other hand, some domains place more emphasis on the data stored within a distributed system. Astronomical sky surveys, for example, produce vast amounts of image data that must be efficiently stored and queried [234]. In addition to computationally- and data-intensive applications, Grid-based systems have been proposed for other applications including monitoring remote sensor networks, such as flood prediction [151]. In these latter cases, Grid infrastructure is used for its data sharing and notification properties; data can be exposed by sensor devices in a standardised way to the rest of the system, potentially using the Internet as a communications medium rather than custom cabling or wireless systems.

Implicit to all Grid definitions is the idea of a (potentially large) set of loosely-coupled nodes participating in a distributed system, providing each other with various Grid services, potentially spanning multiple geographically distant sites [129]. It has been pointed out that Grids should provide more than just access to large amounts of compute power. The modern concept of a Grid system should incorporate data management, security, and interoperability in addition to computational resources [272].

The components of a Grid system might be spread across different institutions, located at geographically distant sites. Although nodes within a Grid environment could communicate via any protocol (proprietary or otherwise), there has been a tendency for Grid-based systems to converge on the use of standard, open protocols [161, 251] such as [FTP](#), and Web services (Web Services Description Language ([WSDL](#)), [SOAP](#)). Standardisation on suitable communication protocols is required for effective data and resource sharing. Due to the potentially global nature of a Grid system, there are

practical issues to consider such as the traversal of data through corporate firewalls which, though extremely challenging, is starting to be addressed [118]. Data transfer methods traditionally used by distributed computation systems, such as shared file-systems are not necessarily suitable for Grid infrastructures.

Web service oriented architectures address several of the communications requirements of cross-site grids and many Grid frameworks have employed Web services to provide Grid-services. The distinction between “Grid services” and “Web services” has become somewhat blurred over the years. The technical definition of a Web service defines it as a stateless entity - requests are considered as distinct entities with no relationships between them. Grid-services on the other hand, often need to maintain state information between requests, for example so that multiple requests from the same client can form a session. Grid services are essentially Web services with extended functionality — in particular state-full servers and event notification support [102]. In practice, both Grid and Web services use the same technology, to the point where they might be considered synonymous in most contexts [18, 41].

It has been suggested that in order for computing Grids to scale in the same way as utility grids such as electrical grids, different levels of infrastructure to deal with global, local and site-specific requirements should be developed. This is analogous to the national, regional and local infrastructures seen in power grids [54]. Grids should be capable of managing fluctuations in supply and demand, such as those caused by time zone differences [54]. While distributed computation systems such as Condor and Sun Grid Engine (SGE) are certainly types of Grid technologies, they are primarily job-scheduling systems. For the purposes of this thesis, they are considered to be essential components of a Grid-based system, but not a complete Grid by themselves. For the purposes of this thesis, a Grid-based system is defined as encompassing the following:

- Application-level workflow and pipeline management utilising SOAs.
- Data management facilities including: archival, distribution, and browse-able access.
- Job scheduling and enactment at a computer cluster or institution level.

Both client-server and decentralised architectures have been employed in development of Grid systems. Projects using these architectures have utilised a wide range of hardware, ranging from high-performance dedicated hardware, to clusters of ‘volunteer’ desktop computers. Some Grid systems have been developed with a single application in mind, while others are more generic in nature. The suitability of a Grid middleware platform for a particular application domain depends to a large extent on the properties of the computational work to be done.

2.5.2 High performance grids

Globus [110] is a widely-used Web services-based Grid framework. It allows compute resources at multiple remote locations to be used collaboratively to achieve a domain-specific goal. Geographically and administratively distinct Grid resources can be combined into a virtual organisation. For instance, federation of large data resources at multiple locations may be required for data-integration purposes, or powerful computational resources may be exposed to facilitate large-scale data analysis that would otherwise be unfeasible at a single institution. Globus addresses many of the needs commonly required by large-scale Grid applications, such as resource discovery, task scheduling, data transfer and data security. By providing commonly-required functionality as a series of open source libraries, it promotes re-usability and increased quality/reliability. In fact, Globus has become a de-facto standard for collaborative Grid projects involving the resources of large institutions.

Globus Grid services are described by the Open Grid Services Architecture (OGSA) specification [107, 109]. An implementation of this specification is the Globus Toolkit [105]. It provides a series of components, each fronted by a Web service, for providing functionality common to Grid applications. For instance, ‘execution management’ facilities are provided by several modules: the Grid Resource Allocation and Management (GRAM) is responsible for configuration, staging, execution, and monitoring of remote executables; the Workspace Management Service (WMS) allows the use of virtual machines to execute pre-configured tasks in isolation of other processes. Data management in a Globus system is provided by several complimentary modules: GridFTP is the underlying data transport mechanism; Reliable File Transfer (RFT) is a layer on top of GridFTP providing reliable delivery; Replica Location Service (RLS) provides a decentralised mechanism for locating file replicates.

Although co-ordination and state querying operations are facilitated by Web services, bulk data transfers are performed by the GridFTP protocol. This protocol has been shown to support very high point-to-point throughput when suitable equipment is available [4]. The RFT module augments GridFTP with failure handling mechanisms and automated retry attempts, required when large numbers of files must be transferred between sites without continual monitoring by an operator.

Many Grid service providers specialising in high-performance computing provide Globus installations on dedicated hardware, as well as other Grid middlewares such as Condor and batch processing systems [24, 121].

2.5.3 Commodity grids

Commodity Grids are system composed of [COTS](#) hardware, typically large numbers of desktop computers. These Grid systems may be dedicated to a particular projects processing needs, or may make opportunistic use of the idle time of desktop computers. Alternatively, computational time may be donated to the system by individuals with no formal connection to a project; such volunteer Grids may be composed of many thousands of home computers [[312](#), [289](#)].

Although Globus has gained widespread acceptance, becoming a de-facto standard in the scientific community, some have argued that its client-server approach and administrative upkeep requirements make it sub-optimal for desktop Grids [[45](#)]. Where large, dedicated computational resources are available, such as the LHC Computing Grid [[100](#)], the service-oriented architecture is a logical choice. However, Grids composed of multiple desktop computers have different properties.

Desktop computers typically have less processing capability, smaller amounts of RAM and disk capacity, as well as slower network connections than dedicated high-performance servers. However, if a Grid computation can be divided into small enough parts able to execute on less-capable machines and each part is relatively independent, then desktop or volunteer Grids can offer a potentially vast amount of computational resources extremely economically [[308](#), [12](#), [127](#)].

Distributed processing systems such as Condor [[192](#)] and various batch processing systems [[294](#), [53](#), [128](#)] capable of utilising the idle CPU time of desktop computers have been available for several decades. However, their relative complexity, system administration requirements and potential dependence on locally-available shared file-systems has limited their scope to large organisations with the resources to manage deployment and maintenance of large numbers of nodes. In the mid-90s the Seti@Home [[154](#)] project was one of the first, and arguably one of the most successful ‘volunteer computing’ projects. Its ease of use enabled ordinary computer users to participate in a large distributed computation project, simply by downloading a small client program that connected to a set of central servers. Communication between the client and the server was purely via [HTTP](#), facilitating its deployment to fire-walled machines behind proxy servers. Large computational tasks were split into chunks called ‘work units’. Each participating computer could download a work unit, process it, and upload the results. Due to the low speed of commonly-available Internet connections available at the time (dial-up), bandwidth had to be managed carefully. While each work-unit was roughly 400kB, the output size was typically much smaller. Although Seti@Home was a custom-built, single-purpose project, it demonstrated that large-scale analyses could harness the power of individual personal computers connected via the Internet. Over the next few years, several other

single-purpose distributed applications were released [279, 74].

As volunteer computing increased in popularity, it became clear that single-purpose compute clients were too limited. If a user wanted to participate in multiple projects, they would need to download a separate program for each project. Aside from inconveniencing the user, each separate project duplicated programming effort, such as job management, error handling, and downloading and uploading files. There was therefore a need for a generic client that could run manage several distributed computation projects [9]. There are several advantages to such clients: a single API for developers; common functionality is shared between projects; if one project suffers a server failure then other projects can keep the worker nodes busy by sending additional work units.

2.5.4 P2P architectures in Grid organisation and communications

It has been suggested that P2P architectures and the goals of the Grid overlap in many ways [183]. P2P architectures have been put forward as a potential solution to client-server scalability problems faced by increasing numbers of nodes in Grid systems. Many P2P approaches have been suggested both for system organisation and resource location [306] as well as for scalability in data transfers [315]. A number of distributed computing and Grid projects have recently begun to exploit P2P techniques. These are described in this section.

2.5.4.1 Peer to Peer approaches to resource matching

An important ability for a desktop Grid system to have is to be able to match hardware and software requirements of Grid tasks to appropriate worker nodes. Condor does this via its ‘class-ads’ system, whereby desktop computers are registered with a central server. The Condor task scheduler then matches submitted job requirements to worker nodes with appropriate hardware specifications.

A decentralised approach for matching computational tasks to worker nodes in a ‘P2P Grid’ [307] has been proposed. The authors argue that this approach removes the need for a centralised server, permitting very large Grids to be constructed in an ad-hoc manner from Internet-connected PCs. Grid resources termed ‘producers’, such as individual desktop PCs attached to the Internet, describe their locally-available resources via an XML description. This includes hardware information including CPU architecture, and operating system type. The resource description also specifies ‘logical’ resources, such as available data files and software present on the machine. These ‘producer’ descriptions can then be matched to similarly-structured ‘consumer’ descriptions that specify the requirements of computational tasks. A participant of the P2P Grid requiring a set of hosts (providers) to

perform computationally-intensive work can query for suitable machines in a decentralised manner. The P2P Grid system is based on Gnutella [37]. Therefore, no centralised server or resource registry is required. The authors point out that a system such as P2P Grid is suitable for environments where the capabilities and available resources of participating host machines may change dynamically, such as the addition or removal of USB storage devices.

Gridkit [43] is a component-based Grid framework. Gridkit is designed for deployment to dynamic environments, where devices may not be continually connected, rather than the more static environments suited to frameworks such as Globus. The authors point out that the resources allocated to a Globus-based application are not dynamically configurable and that it is not possible to alter application behaviours, including extensibility at runtime; that is, server restarts are required.

Gridkit appears to be geared towards real-time data collection from sensor networks[151]. As such, it supports a wide range of Quality of Service (QoS) options and data collection capabilities, including streaming information from remote sources. A small memory footprint (small WS-stack) and efficient routing algorithms allow Gridkit to execute within embedded devices with limited network bandwidth. QoS requirements of a realtime system are met by the attachment of QoS specifications to tasks, where a ‘task’ is defined by Gridkit to be a single unit of work. QoS specification descriptions allow application domain-specific QoS terms to be defined in an ontology, allowing customised requirements to be specified, such as ‘minimum frames per second’, ‘minimum latency’. The authors claim that this fine-grained approach to application processing requirements gives a much greater amount of control than other frameworks such as Globus, where ‘task’ requirements are specified in terms of physical attributes such as numbers of CPUs, or amount of RAM.

Applications in a Gridkit system are constructed as a set of potentially distributed interconnected components. Each component is responsible for the implementation of a particular type of functionality, such as providing access to a database, providing a buffering capability, or performing some computation on a data stream: compression, transcoding and so on. A Gridkit ‘task’ is the composition of multiple components. Inter-component communication is provided by Gridkit in several different forms: request-reply, multicast and notification-style publish-subscriber models..

Once an application has been specified in terms of components, a set of ‘virtual clusters’ can be constructed that represent mappings of components to suitable sets of physical hardware. For instance, a virtual cluster can range from a group of processes located on a single machine, to a distributed set of processes running on multiple machines. A virtual cluster appropriate to the needs of the application can be constructed by examining the annotations that specify QoS requirements of the application. For example, a requirement that states that the communication between two components should be

of very low latency may mean that the optimum physical layout is to place both component processes on the same machine, or on machines that are topologically ‘close’.

2.5.4.2 Mobile agents in Grids

A Grid system intended to utilise Internet-connected PCs via mobile agents is outlined by Fukuda et al. [117]. The authors point out that executing applications on computers under remote ownership poses several problems including: reliability, trustworthiness and limited connectivity of available computational resources. A solution based on mobile agents is proposed, providing the necessary middle-ware to allow user jobs to navigate between computers as necessary in an automated fashion. The infrastructure of the system consists of:

- A ‘moderator’ service that maintains user registration information.
- A web-based interface allows users to submit jobs to the system, allowing necessary job files to be uploaded, and system requirement information to be provided.
- A mobile agent, an instance of which is created for each job and is responsible for managing the job during its life-cycle.
- A Java user job wrapper responsible for mediating communications between the agent and the application layers.
- A user job (application), to be written in Java, C or C++.

The mobile agent responsible for job execution will query a registry of available computers, termed ‘targets’, in order to find the best match in terms of system requirements. If no matching machine can be found then other ‘moderator’ services, interconnected via the Globus Metacomputing Directory Service, can be queried for suitable hosts. Once a suitable target machine has been found, the agent initiates a transfer of the required data and executable files and a ‘child’ agent is started on the remote machine. If more than one machine is required, for instance, for parallel computation, then multiple child processes can be started on separate target computers. File transfers are handled via HTTP; each desktop computer runs an instance of the Apache Web server [112].

Snapshots of program execution are required in order to migrate the application to another target machine, in case the first machine suffers a failure or otherwise becomes unavailable. Such snapshots need to be made periodically, to be backed up to other target machines. ANTLR [239] and JavaCC

[3] are used to pre-process C/C++ or Java user applications at compilation time in order to add the necessary check-pointing hooks.

IPC between multiple target computers working on the same computational task is handled by the Java-based wrapper. Since target computers may be operating behind firewalls, direct communication between them may be impossible. Instead, each target computer is assigned a unique identifier. Communications between concurrent processes that would normally be routed directly to the appropriate worker nodes through an API such as Message Passing Interface (MPI) must first be tunnelled through the HTTP connection back to the agent overseeing the process executions. The agent then forwards the message(s) onwards to the intended recipients.

2.5.4.3 Ensuring fairness in a P2P Grid

One of the potential problems of “public” grid computing is the reliance on the good-will of volunteers to provide computing power, and the compute projects to be not too “greedy” in terms of the CPU time they consume. CompuP2P [135] is a distributed computation architecture that aims to increase reliability over client-server approaches such as Seti@Home. CompuP2P introduces the concept of “buyer” and “seller” nodes for resource providers and consumers. Resources to be shared might include CPU power, or disk storage space. Seti@Home has a centralised architecture, where only the server is permitted to schedule jobs for processing and only the peers perform computational work. CompuP2P on the other hand, allows any node to submit requests and/or perform computation. Rather than rely on the good-will of participants in networks such as Seti@Home and BOINC, CompuP2P implements ideas based on microeconomics to encourage participation in the network. Participating peers can “sell” resources such as CPU time to other peers in need. Eventually, other peers are able to reciprocate by making their available resources for a price. Peer-based negotiation takes place to determine the best “price” and therefore which “seller” a buyer can obtain resources from. This kind of scheme should work well when participants require roughly equal amounts of resource from the system over time. It requires that “selfish” nodes eventually contribute back to the system in order to gain “currency” required to purchase more resource. On a public network, this may well be the case.

2.5.5 Cloud computing

Cloud computing, sometimes also referred to as utility computing, is still in its infancy, and as such there are many conflicting definitions as to what it is [86]. Typically, a provider with a large amount

of compute resource sells units of computation to consumers. Users pay for units of computational resources they use, such as CPU hours, units of disk storage, and units of network traffic. Cloud computing offers the ability to dynamically expand an organisation's compute power, for example, to handle large numbers of requests at peak times [201].

There are various forms of Cloud computing, ranging from highly-specific applications such as email and productivity applications [2], to intermediate-level building blocks for application developers [51], to low-level virtual machine instances into which users can install any application they choose [1].

Cloud computing is considered by some to be an evolution of Grid computing [168]. Grids require the formation of virtual organisations formed from a group of institutions willing to share computational resources. However, simply combining the existing computing resources from a group of institutions is likely to result in a heterogenous environment, making it difficult to develop and deploy applications. Fears over security also mean that system administrators may be unwilling to make configuration changes that would make a Grid more accessible to its users and developers.

By contrast, Cloud computing aims to provide the end user greater flexibility and control over the computing infrastructure they rent. To a large extent, this greater control is facilitated by allowing users access to secured virtual environments, rather than allowing them free reign on the physical hardware [168].

Clouds offers a number of advantages to hosting services locally, not least the convenience of not having to manage server equipment in a local data center. Cloud providers will typically have spare capacity to compensate for component failures, or even entire machine failures. Providers may have data centers in several geographical locations to further reduce the risk of a single point of failure [1]. Despite its apparent advantages, there is some concern that using vendor-specific Cloud-based applications and utilities will result in being locked into particular providers [182].

2.5.6 Data management in high-throughput systems

Since rapid data transfer is critical to high CPU utilisation distributed computation systems, some data transfer protocols and file systems have been optimised for this field [339, 4, 46].

Efficient transfer of data is essential for smooth running of a Grid system in order prevent CPUs from becoming idle while awaiting file transfers. Large amounts of data (gigabytes to terabytes) from several distributed resources might need to be transferred to many nodes for processing. In the context of the Grid, data might need to be transferred between geographically distant sites using a Wide Area

Network ([WAN](#)) or public network, such as the Internet. Therefore care must be taken to ensure that expensive network connections are not unnecessarily saturated, and that sensitive information is properly encrypted.

The Globus project has developed GridFTP [4] which incorporates several extensions and improvements to the FTP standard with Grid environments in mind [110]. For instance, transfers may be mediated by a 3rd party, in addition to the FTP standard client-server transfers. The implementation also supports easier traversal of data through firewalls compared to standard FTP, and provides better security via Grid Security Infrastructure ([GSI](#)) and Kerberos. GridFTP also provides more efficient large-scale data transfers by allowing data to be striped across multiple servers (analogous to RAID 0), and supporting partial transfers of data. Large Grid service providers such as TerraGrid typically use GridFTP, SCP, HTTP, and other point-to-point means of communication [222].

Recent work has investigated the use of [P2P](#) data transfers within Grid systems as a means of improving efficiency and reducing the time machines spend waiting for data [316, 124]. Fedak et al. [89] point out that while the popularity of ‘desktop Grids’ has massively increased in recent years, there has only been a little work addressing data-intensive research applications running in such environments. They point out that the existing popular ‘desktop Grids’ such as BOINC are client-server based, and therefore may suffer from scalability problems. The BitDew system described in [89] provides a development framework that allows the developer to specify how many copies of a data item should be made available simultaneously, the useful lifetime of a data item, as well as which protocol should be used to transfer data between nodes. The BitDew system has been demonstrated by executing the bioinformatics application BLAST on up to 250 nodes. Both [FTP](#) and BitTorrent transfers were used in two sets of experiments with a 2.68GB BLAST database needing to be transferred to each worker node. Fedak et al. clearly highlight the advantage of the BitTorrent transport system in this case, where a large file must be transferred to each worker node; while the [FTP](#) transfer test shows a linear decrease in transfer times as more nodes are added, the BitTorrent test shows a constant transfer time as the number of nodes are increased.

2.6 High-throughput computation in e-Science and bioinformatics

One of the central aims of bioinformatics is to extract new knowledge from the data amassed in the various public data repositories.

MyGrid [284] is an e-Science project centred around enabling scientists (and in particular, bioinformaticians) to more easily construct workflows utilising distributed data resources and compute

services. MyGrid consists of data repositories, services enabling access to data, agents providing computational services, and a workflow description language and enactor.

One of the central components of MyGrid is Taverna [236], a workflow enactment and management system. Taverna consists of an enactment engine for running workflows described in the SCUFL language. It also provides a GUI for constructing, enacting and browsing the results of workflows. Taverna provides an extensible result browsing architecture that provides generic views for plain text or XML data, but also allows custom data visualisations to be added. Examples include image and PDF viewers, and 3-dimensional protein structure viewers.

As discussed in the previous Section 2.2, there are several levels at which parallelism can be applied. The decision of which level of granularity to apply parallel processing techniques has important consequences for the ease of implementation and the types of hardware best suited to program execution and therefore impacts the ultimate scalability of a system.

As the size of bioinformatics sequence databases continue to increase at phenomenal rates, full-scale analyses become more difficult to perform in a reasonable time on a single machine. Fortunately, many of the computational tasks in bioinformatics fit into the ‘embarrassingly parallel’ category. One example is the sequence similarity search tool, BLAST [6]. The BLAST tool is already fast compared to its predecessors and for this reason it is hugely popular and well-studied. The standard BLAST implementation can make use of several processors for certain steps of the algorithm, which ultimately speeds up its search when compared to executing on a single CPU. The BLAST algorithm has also been implemented in a distributed parallel fashion, using the MPI interface [252]. This approach enables a larger number of CPUs to participate in a sequence search, but the system is ultimately limited by IPC overheads. Another implementation of BLAST utilises a Grid-based framework to execute multiple, independent instances of the program on different data sets [119]. This approach permits ‘embarrassingly parallel’ scalability over thousands of nodes arranged in geographically distant clusters.

Executing BLAST on a large amount of sequence data has an embarrassingly parallel solution, since the data set can either be split at the genome level for small sequences, or larger sequences can be split into smaller sections. The various implementations of BLAST highlight the trade-offs for targeting SMP-type hardware, distributed parallel approaches, and Grid approaches, respectively.

Also of interest is how a computational problem that is not easily parallelised, or only parallelisable to a small extent at a low (thread) level, becomes embarrassingly parallel at a high (program instance) level when there is a requirement to perform exhaustive searches over all available data.

An added benefit of coarse-grained (program-level) parallelism is the ability to execute programs on a large scale that cannot be easily modified, either because the source code is not available, or would be difficult to parallelise. Large distributed systems have been constructed and shown to work well with this kind of parallelism. In such systems, parallelism within individual program instances becomes a less important issue. Instead, the focus of this thesis is on the distribution and staging of data to remote compute resources, facilitating the coordination of multiple, large computational tasks using a variety of bioinformatics analysis tools, and enabling the extension and updating of such systems in the provision of an exhaustive data resource for biologists.

BioPipe Biopipe [147] is a Perl-based framework for constructing processing pipelines. Its purpose is to allow analysis pipelines to be developed from re-usable components. The control mechanisms are database-centric, while computationally intensive tasks are sent to a cluster of computational nodes for parallel processing. Biopipe pipelines are described in an XML document that specifies data sources, program executables, and execution ordering. Pipelines are constructed modularly from several building blocks: `Input/Output (I/O) components` for handling data transport and parsing requirements, `analysis components` for overseeing data processing, and `application wrappers` that bridge Biopipe with existing analysis applications.

Data transfer and parsing operations decoupled from processing operations. A Biopipe `I/O component` can handle data transfers from several types of data source, including Relational Database Management Systems (RDBMSs), flat files, and HTTP transfers. Bioperl is employed to facilitate an abstraction layer between particular data formats and analysis components. Biopipe `analysis components` use in-memory Bioperl objects for processing, or conversion into an appropriate format for use with third party applications. Such applications are supported through the use of `wrappers` that provide the appropriate interfacing logic. Wrappers have been written for many popular bioinformatics applications such as BLAST [6], CLUSTAL W [304] and Genscan [42].

Pipeline enactment is performed by Biopipe's job management system, which shares much in common with the Ensembl pipeline [246]. `Job units` are modular components that make use of one or more Biopipe `I/O handler components` for the data acquisition, parsing, and result output operations. Biopipe supports job distribution to worker nodes via the batch execution systems LSF [58] and PBS [128].

PEDANT PEDANT[115, 313] is an automated genome annotation system providing an exhaustive analyses of all publicly available genome sequences. Protein similarity information is provided via

Blast and the SIMAP project. Exhaustive InterProScan analyses are also provided. Notable features of PEDANT include the pipeline approach, the use of a Grid architecture for computationally-intensive programs, and Web service query interfaces to result data sets. Walter et al. point out that it is becoming increasingly infeasible to re-compute data sets from scratch for every update — i.e., for every newly released sequence. As such, they utilise a pre-computed data similarity data set, SIMAP, which they claim increases performance by between 5 and 60 times. An interesting property of PEDANT is that there are no ‘releases’ as such; incremental updates are added as new genome sequences are published. Presently PEDANT performs only single sequence annotations, although there are plans to provide comparative analysis of genomes in the future.

2.7 Summary

Bioinformatics is a cross-disciplinary science, drawing on knowledge and expertise from several domains. The rate at which new data is being generated is increasing exponentially. Currently, complex workflows have been constructed by bioinformaticians to run advanced sets of analyses over fairly small data sets [236]. Alternatively, large-scale data processing centres run pre-defined annotation pipelines over large amounts of data [97]. Ideally, these two worlds could be combined. This requires large amounts of computational power combined with a flexible analysis pipeline development and enactment environment.

Many large organisations have a substantial numbers of desktop computers. It has been shown that much of the time, these computers are under-utilised. Even at peak periods there are often a large number of idle machines in public clusters. Utilising idle workstations for computationally-intensive work has a number of advantages. There have been several works that have shown that utilising commodity hardware is often extremely worthwhile, both in terms of cost-effectiveness of reusing existing infrastructure more efficiently, and in terms of useful quantities of analysis work being performed.

This thesis is concerned with how to develop and enact flexible, complex bioinformatics analysis pipelines within a distributed computing environment composed of a mixture of dedicated compute hardware and commodity desktop computer hardware. Furthermore, approaches to exposing this functionality to bioinformaticians — who are experts at developing or interpreting the output of advanced analysis algorithms, but who are not necessarily experts in distributed computing — are also investigated.

This chapter has discussed relevant previous works and technologies for this field of research. The

next chapter discusses the motivations and requirements of a Grid system capable of addressing the project aims introduced in [Chapter 1](#).

Chapter 3

Microbase

3.1 Introduction

This chapter introduces the motivations and system-wide requirements for a distributed computation framework, Microbase, suited to performing long-running bioinformatics analyses. Subsequent chapters discuss each component of Microbase, and how these components contribute to the overall system requirements.

3.2 Motivation

Genome sequence data is becoming available at an ever increasing rate, with the number of active genome sequencing projects increasing exponentially [191]. Publicly available sequence databases such as GenBank double in size roughly every two years [27]. Parallelism is one of the most obvious ways to speed up the processing of large amounts of computational work. As CPU clock speed increase limits are reached, desktop PCs are becoming multi-thread capable as processor manufacturers shift towards dual- and multi-core processors. What once was the domain of high-end server hardware, is now becoming increasingly available in commodity hardware. Therefore, it seems inevitable that exploiting parallelism is essential for future increases in performance, even within individual computers [238, 144]. However, while the ability to run many threads concurrently has the potential to vastly speed up the overall processing ability of a computer, the new capabilities cannot be used under all circumstances. Older single-threaded programs will not see instant speed improvements as was the case with previous CPU improvements such as clock speed increments, or additional cache memory. There are two ways to overcome this limitation: re-write applications

to take advantage of parallelism; or run more than one application, or more than one instance of an application concurrently. Although there are big differences between distributed compute systems and standard desktop PC hardware, the underlying problem remains the same. Unless applications are aware of their environment, or the operating system has enough processes to schedule simultaneously, then no advantage of parallel hardware will be seen; the additional CPU cores will be under-utilised. In other words, parallel-processing capable machines are not inherently *faster* at performing a single task, but they are capable of running several such tasks simultaneously, thus improving overall throughput. Parallel architectures are ideal for solving multiple problems at a much faster rate than a sequential processor would allow. They are also suitable for large-scale problems if the computational work can be sub-divided into smaller units of work.

Large computational tasks in bioinformatics can often be split into more manageable chunks, suitable for execution on ordinary desktop machines rather than requiring the use of large dedicated compute clusters. However, the implementation usually depends on how domain-specific problems can be divided. Apart from the obvious application-level split, logical choices include splitting the computation into genome- or protein-sized chunks. For instance, the all-vs-all alignment of a large set of sequences¹ has been calculated to be intractable on a single computer, requiring in excess of 1500 CPU years[119]. Instead, the mammoth task can be divided by performing multiple pair-wise alignments on subsets of sequences. As long as the hits from each alignment are combined, the final result set should be the same as that produced by a single long-running task on a single machine. The suitability of the data to fit smaller computers and be distributable among them makes utilisation of general-purpose desktop computers attractive. These otherwise-idle machines can be put to good use, maximising their investment and lessening the need to buy additional expensive server-room equipment. Additionally, when the next hardware update cycle arrives, compute power available for distributed job processing will be increased for no additional cost.

With mass-market dual- and quad-core x86 processors available from both Intel and AMD, it is only a matter of time before multi-core CPUs become dominant in desktop PCs. This provides the opportunity for desktop Grid systems to execute multiple single-threaded distributed compute jobs on each desktop PC, or allow the applications that support multi-processor machines (such as BLAST or InterProScan [337, 225]) to complete their tasks more efficiently. In effect, the best of both the distributed and purely parallel worlds can be combined: isolated processes can run over multiple worker nodes with minimal Inter-Process Communication (IPC), while at the same time, multi-threaded parallel programs can utilise commodity multi-processor machines, forming a part-

¹The nucleotide sequence 'NT' database contains entries from the GenBank, EMBL, and DDBJ databases, available here: <ftp://ftp.ncbi.nih.gov/blast/db/>

distributed and part multi-processing solution.

On a typical university campus, or large corporate environment, there are likely to be several thousand desktop PCs. The sheer number of PCs effectively guarantees that a proportion of them will be completely idle (i.e., no logged-in users) most of the time. Even during peak hours, a large number of machines are available. At Newcastle University there are approximately 2400 computers participating in the Condor pool. One of the major motivations for developing Microbase is to take advantage of increasingly powerful, but often under-utilised desktop PCs for CPU-intensive bioinformatics applications. There is therefore a requirement to handle a wide variety of hardware capabilities and configurations. Dedicated compute resources (i.e., large cluster machines) may form part of the available compute power, but a large proportion of a typical Microbase installation's computational power is intended to come from 'cycle-scavenging' idle time of ordinary desktop computers. One of the most important aspects to consider is that the primary purpose of these desktop computers is not for running Grid applications. They are 'volunteer' computers that make their resources available to the system when they are not being used. The Grid compute client will run with the lowest priority and may be interrupted by at any time by a more important task, such as a user log-in. The compute client may be suspended, or even entirely removed without prior warning.

Although there is a vast potential of processing capacity available from commodity desktop worker nodes, several challenges must be overcome in order to utilise them reliably.

- Desktop nodes may join or be removed from the system at any moment, without warning. It must be possible to migrate work to alternative locations.
- Desktop nodes do not necessarily have narrow-interest domain-specific software installed.
- The logistical issues of the distribution of large files to remote computers needs to be considered.
- Although system administrators must initially permit the use of a Grid compute client, it is not feasible to expect them to install and maintain required domain-specific applications. A low administrative overhead is therefore required.

3.3 System-level requirements

The Microbase system requirements cover a broad range of categories including: data handling; responsiveness; modularity; accessibility; environmental; reliability; and usability. The following

sections introduce the project-level requirements of a Grid system intended to support bioinformatics analysis pipelines.

3.3.1 Environment-specific considerations

The Newcastle University Condor installation currently consists of over 2000 nodes forming a heterogeneous set of Windows and Linux machines with varying hardware capabilities. A 96-CPU (64-bit Linux) dedicated cluster is also available. Apart from the ability to submit Condor jobs, and Secure Shell (SSH) access to the Linux machines, we have no control or special privileges over the machines; they are part of a centrally-managed system for which we have no administrator access. Some temporary file space is accessible for the duration of a job execution. This file space is purged at the end of a Condor session, so cannot be relied upon for persistent storage. Since most of the available worker nodes are general-purpose desktop PCs, no assumptions can be made regarding the availability of domain-specific software packages. Worker nodes are also susceptible to being removed from the Condor pool at any moment, resulting in the termination of any active job(s) on that node. The Windows nodes are configured to run Condor jobs only when no user is currently logged in, so as not to impact on the user experience. Therefore, pool-disconnection events occur quite frequently and unpredictably for a given PC. It is likely that more PCs will spend a longer duration connected to the Condor pool outside of normal working hours [294]. Given the large number of machines, there is also the possibility that hardware failures may regularly incapacitate small numbers of machines. Microbase must be able to work within the constraints of the centrally-managed network at Newcastle. These constraints require:

1. Platform heterogeneity: the system must be able to cope with dynamically changing ratios of operating system or CPU architecture availability by adapting the number of scheduled tasks to suit the current platform availability distribution.
2. Job migration: Microbase must have the ability to migrate computational work to alternative worker nodes when active node(s) become unresponsive or unavailable.
3. Handling job execution failures: Job execution failures resulting from environmental properties are expected to occur relatively frequently. A distinction should be made between a job execution failure that occurs as the result of the execution environment, and a job execution failure that is the result of a job implementation fault. In general, jobs should be retried in the case of a failure, and a job re-execution should only reduce the overall efficiency of the system.

Failures should not impact the validity or accuracy of the results obtained from a computation, and should not result in ‘duplicate’ data items.

In contrast, server machines available to Microbase can be considered to be much more reliable than worker nodes. This class of machine typically has a large amount of high-performance disk capacity. Administrative access is also available on server hardware. It is expected that high-availability centralised services and databases can be deployed to server hardware, rather than desktop nodes.

- Microbase must support execution of domain-specific software on a variety of platforms, ranging from 32-bit Linux and Windows-based desktops to 64-bit dedicated compute clusters.
- Automated installation of domain-specific applications to remote worker nodes must be supported. This requirement is needed to meet the needs of running in a non-dedicated environment, as well as reducing the demands on system administrators.

3.3.2 Scalability requirements

Scalability requirements are driven entirely by the environment Microbase is expected to be deployed to. Each worker node present within an installation puts additional load on server architecture. Large numbers of worker nodes will necessitate scalable server processes so that the load can be distributed over a number of servers. Server load comes from a variety of sources, including: requests from worker nodes for work, data transfers to and from worker nodes, and management of large SQL result repositories. The responsibility for maintaining a scalable system is jointly shared between the Microbase infrastructure and the applications that run within it. Microbase is responsible for providing scalable and extensible infrastructure for domain applications to execute within, including:

- Efficient file transfers between nodes is required. Distribution of data files to multiple locations simultaneously is expensive in terms of server network and disk bandwidth. Due to the operating environment Microbase is required to function within, multiple worker nodes will be required to transfer the same files many times. In order to support these large scale file transfer operations, Microbase is required to provide efficient distribution of files in order to minimise transfer bottlenecks that might otherwise undermine the efficiency of the system.
- Low overhead software installations are required. Due to the transient nature of worker nodes, repeated temporary software installations will incur additional network and disk load on server resources.

Domain applications must also take some responsibility for the scalability of the system as a whole. Microbase can guarantee scalability as long as domain applications ensure that shared-resource bottlenecks are not introduced, such as frequent access or complex queries to shared SQL databases.

3.3.3 Data handling requirements

Microbase must handle a number of data management issues relating to the detection of new data files, the management of data flows between applications, and the permanent storage of generated data files. Some data management issues, particularly those to do with temporary, intermediate data files also intersect with maintainability and extensibility requirements introduced next, in section [3.3.4](#).

Since primary bioinformatics data sources are continually being updated, it is necessary to keep secondary data sets up-to-date by acquiring new data and performing new computational work. Depending on the requirements of the applications involved in a pipeline, and the overall goals of the pipeline itself, secondary data sets must be processed in one or both of the following ways:

- Files produced by analysis applications must be permanently archived. This involves practical issues, such as retrieval from ‘unreliable’ worker nodes,
- Depending on application requirements, result data may also need to be stored in a structured data storage system, such as an SQL database.

3.3.4 Maintenance and extensibility requirements

There is a requirement for Microbase to support long-running analyses. Over a period of time, new analysis tools, or new versions of existing analysis tools are periodically released and an analysis pipeline may need to be updated to include these new versions. In the case of major software version updates, for instance if the implementation is substantially changed, or the output file format of a program changes, then the new software version may need to be run in parallel with the old version. This increases the workload since both versions now need to be executed, but is useful if processes further downstream in the pipeline expect data in one or other formats. It is also useful if a comparison between the results produced by the different software versions is required.

It is also possible that entirely new applications will need to be added to an already-installed system. Rather than re-installing and re-generating all data from scratch, it would be preferable to simply add the new application to wherever it needs to be placed within the analysis pipeline. The new

application may have to “catch up” to the current state of the system by processing existing primary data sets, but other applications should be unaffected by the addition of a new application.

The following functionality is essential for long-term maintainability, where programs as well as data are added to the system incrementally:

- The data sets generated by new tools should compliment existing data sets, without requiring existing data to be re-computed.
- It should be possible to add new analysis tools to any point of an existing pipeline, allowing data to flow from one tool to the next.
- Newly added applications must be able to ‘catch up’ with the current system state; i.e., they must have the opportunity to process all existing input data before being required to process new information.
- Version control must be implemented for data files and program executables in order to maintain consistency.

Also essential for long-lived applications is the ability to determine when an application fails, and in what circumstances a failure occurs. Crashes and bugs may be specific to a particular hardware and operating system combination, software package, or even an individual computer. It is also of interest to system administrators to gather hardware usage statistics, in order to determine cluster utilisation and efficiency information.

It is essential to maintain detailed logs of every action performed by the system. Provenance trails are essential so that the impact of a single data item at the top of a pipeline can be traced and assessed as it propagates throughout the entire system. This information is useful for debugging failing jobs as well as for inspecting the general data flows through the system [66]. In addition to the storage of event graphs, all output data files should be stored. Even if an analysis is repeated, the old files should still be accessible. This ensures that system events can be associated with result data, which can be associated with a particular version of a data file and version of a program executable. As new versions of programs or new versions of data items are added to the system, they become the system defaults for new executions or query retrievals, but they do not entirely replace previous system configurations. This ensures that an accurate provenance trail is maintained.

Scalability requirements discussed in the previous section require a Microbase installation to be able to scale as workloads increase. The implications for a maintainable system are as follows:

- It must be possible to add new instances of server components to an existing system with the minimal reconfiguration.
- Likewise, it must be possible to replace (i.e., migrate) currently installed server components to different hardware, again with minimal reconfiguration.

The maintenance and extensibility requirements discussed in this section, combined with the previously-discussed application support requirements point to the need for modularity. Although groups of applications may have data flow dependencies between them, there should be no inherent ‘integration’ between different applications. Modularity should enable additional applications to be inserted at any point within a pipeline, without adversely affecting others. Therefore, Microbase must support the addition of domain-specific functionality through independently-packaged modules.

3.3.5 Application support and workflow structuring

It is anticipated that the majority of applications that must be run as part of a typical bioinformatics pipeline are existing analysis programs that are either single-threaded or exploit local machine-level parallelism. Such programs are often command-line driven and are non-interactive, making them amenable to automation. However, they will not necessarily be aware of distributed computing or Grid infrastructures and services, or even remote file transfer protocols. Microbase is therefore responsible for ensuring that an appropriate execution environment is constructed on worker nodes for applications that masks the complexities of a Grid environment. The applications themselves should be oblivious to the fact that they are running within a distributed environment. Taking into account the environmental conditions (Section 3.3.1), software installations will need to be performed every time a worker node joins the pool of available worker nodes. There is therefore a need for this process to be efficient so that the available CPU time of worker nodes is maximised.

In addition to the logistical requirements of staging data files and the practicalities of executing applications on remote nodes, there are high-level scheduling and data management issues that must be addressed. Firstly, when an analysis application is run by hand, the human operator is responsible for specifying command line switches and data file paths to be used that are appropriate for both the application and the type of input data. For instance, it might make sense to for a particular bioinformatics application to work with prokaryotic, but not eukaryotic data sets. When an application is executed in an automated fashion, the decision regarding appropriate data content and command line formation must also be automated. The responsibility for the formation of appropriate command lines lies with the software agent executing the application and ultimately the developer. While the

content of the data files and actual command line options are inherently application-specific, the process of environment construction and data staging for any application is the same. Therefore, there is a requirement for Microbase to manage these generic functions in order to ease the burden on the pipeline developer.

Secondly, since analysis applications are not aware that they are executing as part of a larger workflow or pipeline, it is necessary for a management process to supervise high-level operations. The tasks a supervisor application would need to undertake include: reacting to new input data becoming available; scheduling instances of an analysis application to execute with the appropriate data; concatenating result data from completed distributed executions; managing structured data stores, such as pipeline-specific SQL databases; and announcing to the rest of the system when an analysis task has been completed. These ‘supervisor’ applications would need to be written by the pipeline developer, since they are inherently application-specific.

While it should not be necessary to modify existing applications to run within a Microbase system, the pipeline developer will be required to write a ‘wrapper’ around analysis tools in order to supervise their operation and to co-ordinate with other elements of the pipeline. The process of constructing command lines is inherently application-specific, there is a requirement for Microbase to ensure that such customisations are possible in a manner which is suitable for the heterogeneous execution environment, and the pipeline developer.

1. It must be possible to re-use existing applications without modification.
2. In order to minimise the load on limited server resources, an efficient data transfer mechanism is required for repeatedly copying files to worker nodes every time they join the Condor pool.
3. Microbase must manage the construction and removal of temporary execution environments on worker nodes, providing the following functionality:
 - (a) software installations,
 - (b) staging of input data files,
 - (c) archival of result output files.
4. It must be convenient for pipeline developers to construct supervisor applications capable of high-level management of analysis tools. Microbase should provide a development framework that allows supervisor applications to achieve the following functionality:

- (a) Perform co-ordination operations with other supervisors: be informed of new data items as they enter the system, such as newly published genome sequence files; to inform other applications when an analysis is complete. Announcements must be possible without knowledge of specific recipients, or even if there are any recipients at all.
- (b) Determine whether a new data item is relevant to a particular analysis tool.
- (c) Determine the amount of computation required for a new data item.
- (d) Distribute units of the computation among available worker nodes.

3.3.6 User requirements

Distributed systems are inherently more difficult and complex to use and maintain than single-machine programs. One of the barriers to the uptake of a system is how difficult the system is to use, administrate and develop for. There are typically three types of users of a Grid system and usability issues impact these groups in different ways:

System administrators are required to install and maintain Microbase components and necessary supporting software, such as application servers. System administrators need to manage the day-to-day tasks involved in the general upkeep of the system, such as resolving networking issues, monitoring compute cluster utilisation, and ensuring enough storage space exists for result data.

Pipeline developers are responsible for constructing analysis pipelines from multiple bioinformatics applications. These Bioinformatics applications may need to be adapted to fit a distributed environment, or their output files reformatted in order to be fed as input to another program. Developers need a working knowledge of Microbase and its public Application Programming Interfaces (APIs) in order to adapt existing bioinformatics tools to a Grid environment, but do not necessarily require in-depth knowledge of every component.

Research users may be biologists or bioinformaticians wishing to browse or query the output of one or more analyses. These users are primarily interested in the results data from applications. They may wish to submit domain-specific queries that integrate over the available data sets.

3.3.6.1 Developer requirements

One of the main motivations for Microbase is to achieve distributed, concurrent processing of multiple instances of existing applications. Such applications are typically designed for desktop use.

Although some applications may exploit small-scale Symmetric Multi-processor ([SMP](#))-style parallelism, they are not likely to be ‘Grid-aware’. In order to run such applications on a large scale the following are required:

- Existing applications should not need to be modified in any way, and it should not be necessary for them to have any knowledge of the Grid environment they are executing within.
- Microbase must provide insulation to these applications from the Grid, including:
 - setting up and tearing down execution environments on distributed nodes;
 - ensuring correct files are distributed to worker nodes.

Pipeline developers are likely to be bioinformaticians with experience of the programs they are using, and knowledge of the results they generate. These developers will typically have some programming knowledge, but will not necessarily be expert in distributed systems or parallel programming. In order to ease the process of porting applications to execute within Microbase, the following functionality must be provided by suitable library support and developer [APIs](#):

- Ability to wrap domain-specific applications in a manner suitable for deployment to a heterogeneous set of worker nodes.
- Enabling a clear, modular separation to be made between server- and worker node-based components, and their respective responsibilities.
- Provision for decoupled communication with other domain-specific applications.

3.3.6.2 System administrator requirements

The following assumptions are made regarding the deployment environment of a Microbase system:

- Administrative access is available for servers such that databases and web application container directories are modifiable.
- *No* administrative access is required to desktop worker nodes, as long as the Microbase compute client can be started by some means, such as via a system boot script, SSH, or Condor.

Microbase should therefore provide:

- The ability to install the core Microbase services, and required domain-specific applications to a system operating under the constraints specified above.
- Changes to the system including the addition of new applications, or the modification of existing applications should be possible without major administrative effort.
- Server-based components should be ‘mobile’ in the sense that a redeployment to a different set of servers should be feasible without major reconfiguration or data regeneration.

3.4 Architecture Overview

In section 3.3 we presented the requirements for a large-scale, generic, distributed compute platform. This section will describe the architecture chosen for Microbase. Grid-based systems, particularly in an e-Science context, are as much concerned with the flexibility and maintainability of a system and matching user requests to shared computational resources over long periods of time, as they are with scavenging every last available CPU cycle [108]. Any system fulfilling these requirements is likely to become large and complex. Architecturally, such systems are often split into modular components based on functionality (see Section 2.5). We have divided Microbase into the following components, each of which provide an aspect of core functionality. Taken together, these components address the system-level requirements discussed in the previous section:

- Notification system - facilitates de-coupled communication between components. Fully described in Chapter 4 on page 69.
- Resource system - a scalable, distributed file store. Fully described in Chapter 5 on page 85.
- Job management system - provides job scheduling and failure management for heterogeneous groups of worker nodes. Described in Chapter 7 on page 134.
- Domain-specific application components (termed `responders`) - these are user-written components that either perform an analysis themselves, or delegate to a pre-existing analysis program. A framework for the development of these components is discussed in Chapter 6 on page 107.

Microbase consists of a set of separate, loosely-coupled services that co-operate together to provide the infrastructure required by Grid-scale applications. This architectural approach facilitates scalability and reliability through the ability to replicate service components over a number of

servers. In an actual deployment of a Microbase system, the various services listed above may be located on a single physical server, or spread across several, potentially geographically distant machines. For scalability reasons, an installation may contain more than one instance of a given component type; for instance, data-intensive applications may benefit from multiple resource system instances in order to service the needs of multiple concurrent data requests in a timely fashion. However, for the purposes of the immediate discussion, all instances of a particular component type can be considered to be part of the same conceptual unit.

A notification-based approach has been adopted for high-level [IPC](#) between Microbase components. Notification systems have been shown to facilitate the interaction and integration of geographically distant Web services (see Section [2.4.3](#)). In Microbase, the notification system is used to co-ordinate the data flow between applications. It permits modular domain application components to register an interest in a particular type of message and thus receive past and present announcements from other application modules. Event-based messaging enables multiple entities in a distributed system to communicate in a loosely-coupled fashion, essential in a dynamic and changeable environment where the participants are not necessarily known until runtime. The Microbase notification system provides a centralised event-driven communication facility for other services, enabling asynchronous service orchestration to occur.

The Microbase resource system is responsible for storing and distributing input and output data files for each registered application. In a distributed environment, data resources may need to be exposed to a large number of worker nodes simultaneously. In addition, worker nodes must be able to find the resources they require. The Microbase resource system consists of a central resource look-up facility and a dynamically-expandable, scaleable file distribution system. The look-up service provides a directory listing, where items may be tagged with user-defined meta-data to allow efficient querying. The distribution system provides scaleable transfers via a peer-to-peer ([P2P](#)) protocol, facilitating the rapid transfer of files to multiple nodes.

The Microbase job manager oversees the running of computationally-intensive tasks and manages their execution environment. A job is a unit of work suitable for execution on a single computer. A Microbase job implementation either performs domain-specific computation itself, or acts as a thin wrapper around an existing command line application. A job scheduler is required to match jobs to machines capable of running them, to queue jobs until a suitable node is available and to keep track of and retry failed jobs. This allows the job requester to be de-coupled from the actual hardware performing the execution. The Microbase job scheduler provides this functionality, and additionally provides transparent distributed data transfers to and from worker nodes via the resource

system and publishes task execution reports through the notification system. Worker nodes execute a Microbase compute client in order to process jobs. This compute client provides the capability to acquire computational work from the job management system, stage data, and dynamically install software. The compute client may run on top of existing distributed platforms such as Condor [192], or may simply be started by a remote shell, such as SSH.

Together, the components described so far form the core of a Microbase installation (see Figure 3.1). They provide the common, generic functionality that a typical analysis pipeline will require: high-level, lightweight communications; an efficient bulk data transport mechanism; and a job scheduling and execution system for CPU-intensive work. Domain-specific pipelines can be constructed to take advantage of the infrastructure provided by the core components with the addition of one or more components termed *responders*. A *responder* (see Figure 3.2) is a self-contained collection of modules that, when taken together, encompass the entire scope of a domain application's existence within a Microbase system. The modules that comprise a responder may have completely orthogonal functions and may work in entirely different environments (see Figure 3.3), but together their common aim is to support the domain-application within the Microbase environment. For example, a standard bioinformatics command line application, its Microbase compatibility layer (wrapper), and associated SQL database, Web service query interface and user interface would all be grouped together within a *responder*. This semi-formal grouping of related domain-application modules is essential for the fulfilment of several of the Microbase system requirements as discussed in the previous section, since it facilitates introspection of project components which then enables automation and modularity in several areas.

3.4.1 Facilitating flexible and extensible analysis pipelines

Responders will be discussed in detail in Chapter 6. Here, it is important to point out that responders have the following properties:

- Communication with the Microbase core components is possible via event messages transferred via the notification system.
- Responders are not inherently aware of other responders within the system.
- Responders can be organised into hierarchies, where several responders can be 'connected' via shared interests in particular types of notification message.

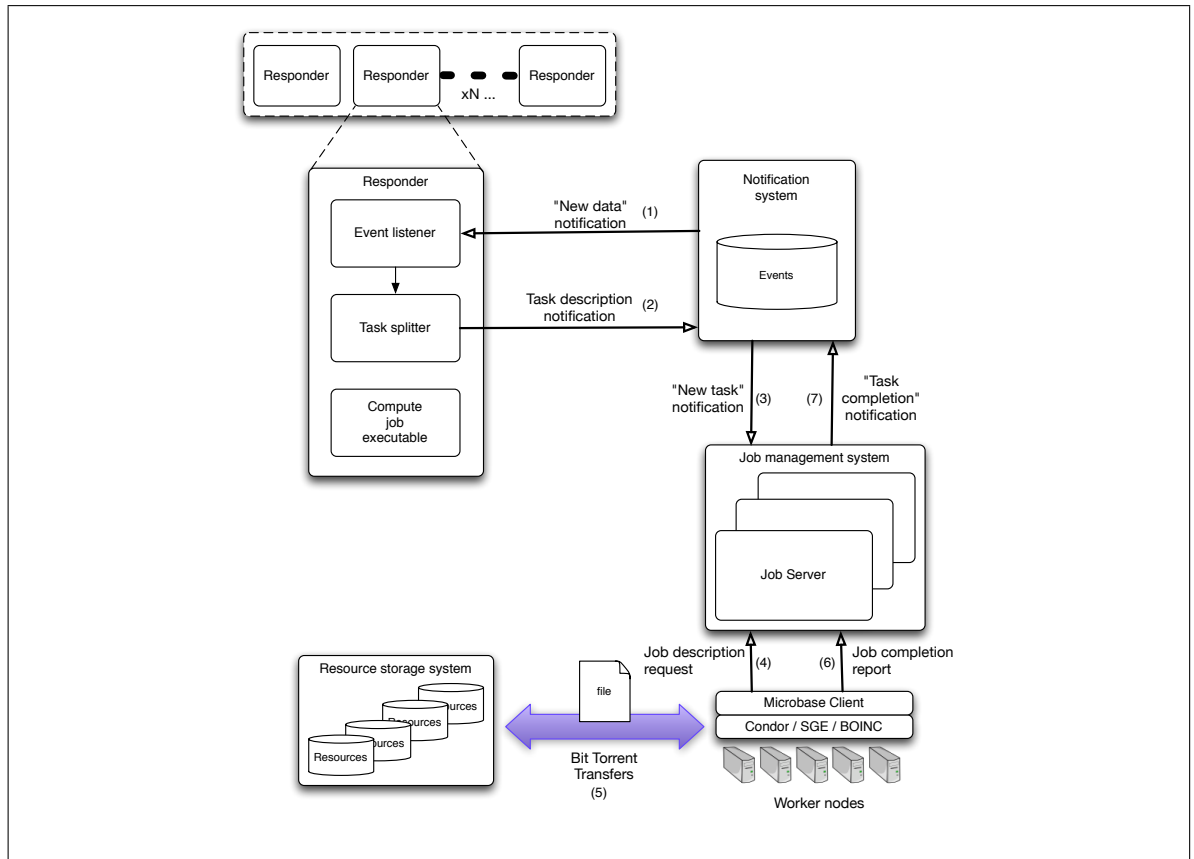


Figure 3.1: Shows the interactions between the major components of a Microbase system. A notification system component routes and stores all high-level messages between the other components of the system. This diagram shows how a request for computational work from a domain-specific analysis application involves the core Microbase components: A ‘new data’ event message (1) is sent to an interested domain application responder. The Web service component of this responder interprets the message to determine if any computational work needs to be performed. An application-specific task splitter then breaks the computational work into units manageable by individual worker nodes. A task description message (2) results, which is forwarded via the notification system to the job management system (3). Here, the requested work is added to a job queue until work is requested by worker nodes (4). Once a worker node has obtained the job description, it downloads and installs the necessary files via a P2P transfer protocol from the resource storage system (5). Files may either be transferred from dedicated resource system servers, or from other worker nodes running similar kinds of jobs. On completion of a unit of work, individual worker nodes publish result files to the resource storage system and job reports are submitted to their allocated job server. Once all jobs relating to the initial responder’s request have been completed, a task completion notification message (7) is sent to the notification system, which is then forwarded on to the responder that originally requested the computational work.

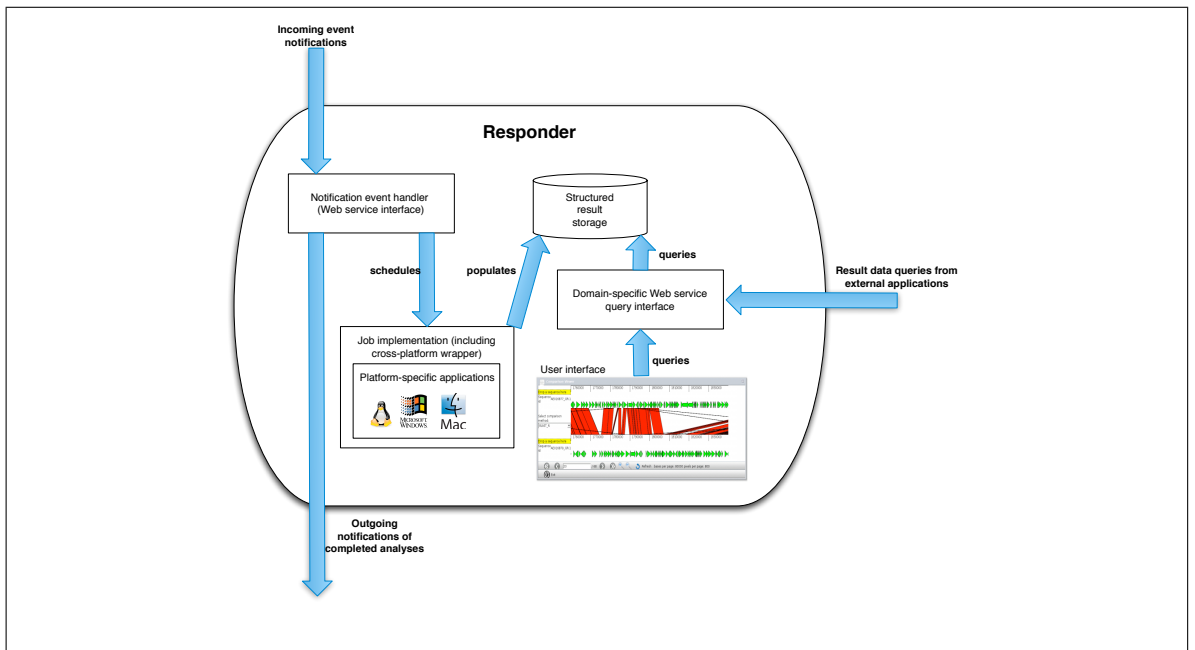


Figure 3.2: A responder is a self-contained collection of modules providing domain-specific functionality. The composition of a typical responder is shown here. A responder minimally needs to contain two components: an event handler and a compute job implementation. The event handler must respond to notifications by deciding how much work needs to be completed to satisfy the event, by scheduling the required work, and to notify other responders when the work has been completed. A job implementation must also be provided that is capable of executing computationally-intensive applications on a number of potential platforms.

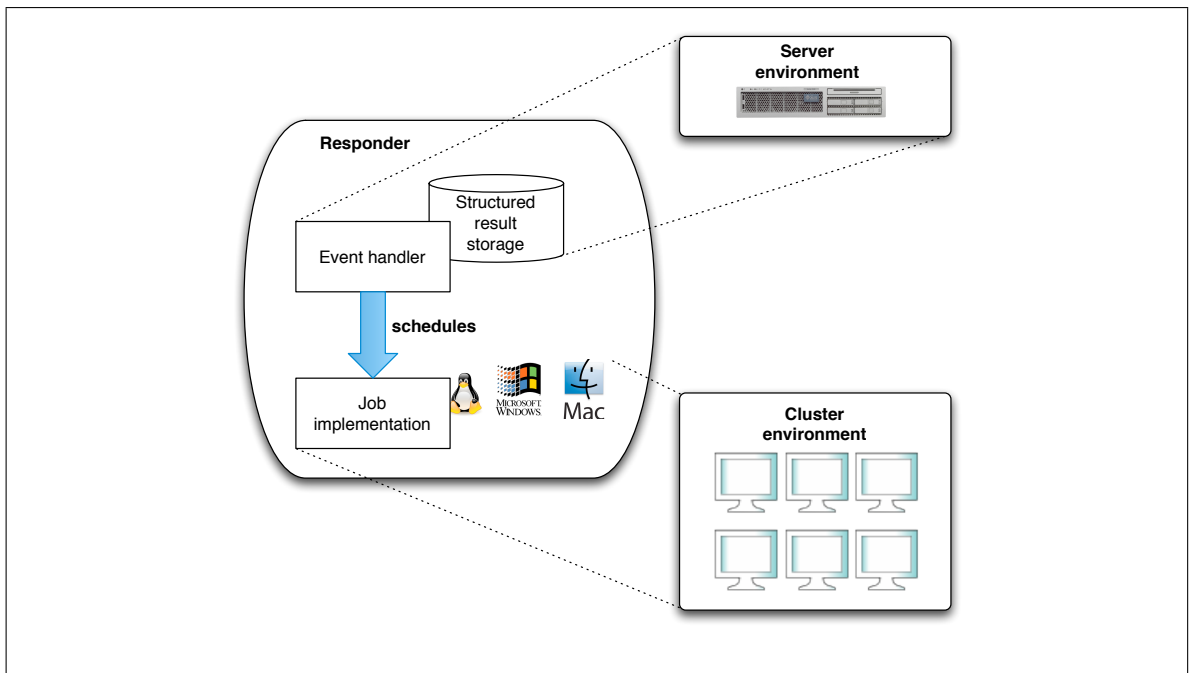


Figure 3.3: Different parts of the responder are deployed to different environments. For instance, control logic and databases are deployed permanently to reliable server-grade hardware, whereas computationally-intensive jobs are deployed on-demand to one or more worker nodes. Job implementations for multiple platforms may be provided.

One of the most important objectives for Microbase is the ability to support flexible and extensible analysis pipelines.

The modular design of a responder, coupled with the functionality provided by the Microbase system together enable flexible and extensible analysis pipelines to be constructed. Several challenges must be overcome in order to satisfy this key Microbase requirement: data preservation and co-ordination. The responsibilities for enabling extensibility are shared between Microbase core responders and domain application responders.

In typical analysis pipelines, intermediate files are discarded as they consume disk space and are of no use to the originally-intended aim of the pipeline. However, to permit pipeline extensions at arbitrary points, *all* intermediate files must be preserved. Figure 3.4 shows an analysis pipeline composed of several programs. Apart from necessary data flow indicated by the arrows, each analysis step is otherwise independent and has no influence on other responders. In Figure 3.4, it is assumed that the goal of the pipeline is to obtain the results of responders 3, 7, and 5. The preceding responders (1, 2 and 4) perform the computation necessary to support that goal, but from the perspective of the pipeline, the preceding responders only contribute indirectly the required result data set.

If the aim of the pipeline shown in Figure 3.4 subsequently changes at a future time, then it is useful to have an archive of the result data from the intermediate steps in order to facilitate pipeline extension without re-computing the intermediate stages. For instance if responder 2 executed a Basic Local Alignment Search Tool (BLAST) analysis and responder 3 filtered ‘interesting’ BLAST hits, then from the perspective of the pipeline, there would be no need to keep ‘non-interesting’ result data from responder 2. In this situation, adding responder A at a future time would be straightforward, whereas adding responder B would require re-computation of the results from responder 2.

Re-executing applications in order to re-generate missing intermediate data is less desirable than storing previous result data. Re-executing applications may be time-consuming, requiring many CPU hours of to complete. Re-generated result files are also not guaranteed to be identical to the original data files if software updates have been applied in the meantime. It is therefore preferable to archive the original data files to guarantee consistency.

Through the use of notification messages, co-ordination of responders can be achieved. As responders are added to a system, they are registered as push subscribers with the notification system. A loosely-coupled pipeline or hierarchy of independent responders then emerges (see Figure 3.4). Different stages of this pipeline are triggered asynchronously as messages trickle through the system. A typical responder has no knowledge of its position within an analysis pipeline; it only has

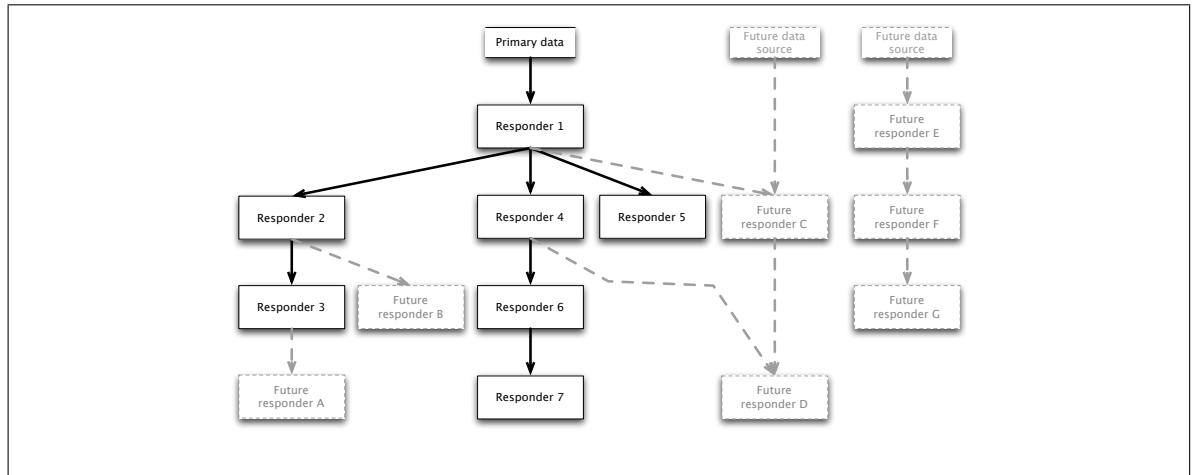


Figure 3.4: A hypothetical analysis pipeline is shown consisting of seven responders, each presumably executing a different type of analysis, and all reacting in response to the preceding responder in the pipeline. The first responder reacts to new data becoming available to the system. Responders shown with solid lines are assumed to be present within the ‘current’ pipeline. Responders with dotted outlines show some of the potential extension points. Arrows indicate data flows between responders.

Shows three different pipeline extension situations that a Microbase system must be able to handle:

- 1) ‘Future responder A’ shows a relatively simple extension of an existing pipeline. The output of ‘Responder 3’ is used as the basis for the input of the new responder.
- 2) ‘Future responder B’ illustrates the importance of preserving data files from intermediate processing steps. The output data of ‘responder 2’ is required in order for ‘future responder B’ to be attached to the system.
- 3) The addition of ‘Future responder C/D’ also require intermediate data files to be stored. This case also demonstrates the ability to introduce entirely new primary data sources.
- 4) Finally, responders E, F, and G show that an entirely independent pipeline of responders could be added to the same Microbase installation. In this case, the second pipeline should have no influence on the first.

knowledge of how to handle the stimuli it is registered to receive.

All of the core services provided by Microbase, except the notification service, are in fact responders themselves. For instance, the job management system reacts to requests for computation, and reports completion events via notification. The resource system also uses the notification system as a means to announce the existence of new file resources. The modular responder-based approach not only enables analysis pipelines to be extended arbitrarily, but the core functionality provided by Microbase itself may also be extended by adding additional responders. The term `core responder` is used to refer to a responder that provides core Microbase functionality, whereas the term `domain responder` refers to a responder that provides application-domain functionality. The distinction is made only to emphasise which responders that are essential and required for every Microbase installation, and which responders provide functionality specific to a particular application domain. There are no technical differences between `core responders` and `domain responders`.

3.5 Supporting technologies

The Microbase architecture is implemented using a set of open-source, standard technologies. The use of existing software components allows systems to be constructed more rapidly by avoiding unnecessary repetition of work. Open-source software purports the ability to customise these existing applications and libraries to suit our requirements, where this is necessary [328]. The use of standards-compliant software facilitates interoperability with other languages, platforms, or research groups. We have selected several technologies to support the development of Microbase, described below.

Web services are an industry standard means of invoking services remotely, or accessing remote data. They have the advantage that they run within a standard servlet container, work well over the Internet (SOAP over HTTP avoids most firewall issues), and are relatively easy to develop [52, 72]. Web Service technology is rapidly maturing and several stable Web service implementation libraries are freely available [73, 300]. Web services play an important role in Microbase. Having a well-defined public-facing interface that hides inner implementation-specific details is desirable for the long-term maintainability of a project, and the users/tools that depend on it. Web Services enable the construction of such APIs relatively easily and are used intensively within Microbase for providing access to remotely-hosted functions and “canned query” interfaces.

Microbase will require the bulk transfer of large amounts of binary data to multiple worker nodes. BitTorrent (see Section 2.3.1.1) utilises a peer-to-peer transport protocol. While several such protocols exist, we chose to use BitTorrent, specifically, the Azureus library [249]. Our chosen implementation of the BitTorrent protocol supports an entirely decentralised tracking system, removing one potential bottleneck in a high-volume environment.

A Microbase-based system will often need to store large amounts of structured data. We have used open-source relational database systems to fulfil this requirement, specifically, Hibernate [299] and PostgreSQL [130]. These design choices for the core Microbase components do not dictate the data storage options for every Microbase component. Although Microbase components have used structured storage mechanisms, third party components are not required or even encouraged to use the storage technologies we have chosen and are free to use their own data storage solution.

Most of the system is implemented in Java [216]. Given the requirements, a cross-platform language such as Java is an obvious choice. Given the heterogeneous environment in which Microbase components must run, the ability to execute applications without recompilation is a clear advantage. In addition, the combination of Tomcat [113] and XFire [73] provides a base for hosting server-side

components.

The Maven build system [114] has been employed to compile the Microbase project. Maven allows the tractable construction and compilation of large projects through a pattern-based design strategy and its comprehensive dependency management. Microbase uses the ability to uniquely identify a Maven project and its dependencies for both compile-time and runtime purposes.

Chapter 4

Notification system

4.1 Introduction

Notification systems are responsible for handling inter-component communications in loosely-coupled distributed systems. The notification system itself is a centralised component at a location known by all other components. System components that communicate via the notification system may be located on the same server, or at different physical locations, potentially dispersed on the Internet.

Entities termed `publishers` are able to send messages to the notification system. Other entities termed `subscribers` can register an interest in particular types of message. The notification system ensures delivery of a message to all interested subscribers. A `message` consists of application-specific content which is not parsed by the notification system and meta-data including a `topic` which is used by the notification system to determine suitable recipients of the message.

In a Microbase system, the notification system provides reliable, ordered delivery of messages required for decoupled components to communicate. Microbase components may play the role of either a `publisher` or `subscriber`, or both (Figure 4.1 on the following page).

4.2 Motivation

Although it would be possible for all Web service components to communicate directly with each other in a point-to-point manner, there are several reasons why this approach is not desirable for all types of inter-responder communication. point-to-point (PTP) communication requires each component to have knowledge of the endpoint(s) of the other components with which it needs to communicate, compromising the modular design requirement (Section 3.3.5 on page 56). If the locations of

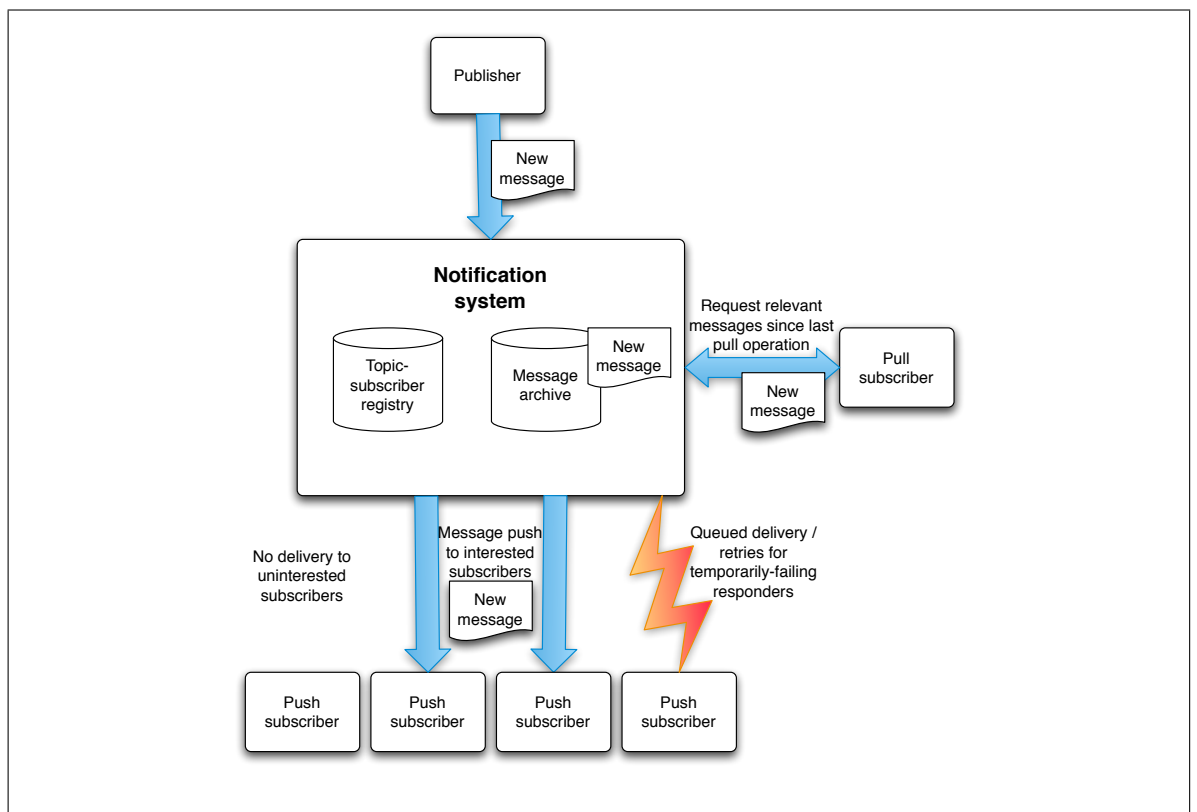


Figure 4.1: Conceptual overview of the notification system. Publishers send messages labelled with a topic. The notification system then determines which subscribers are interested in that message. The message is forwarded appropriately.

these services change — for instance, after a scheduled redeployment or as a result of a server failure — then the endpoints of the hosted services need to be updated, impeding system maintainability. The configuration service (see Section 3.4 on page 60) reduces maintenance issues to some extent through the use of its service type to endpoint registry. However, although endpoints of existing services can be updated, the addition of new components, or removal of existing components would pose a problem. In these cases, each component involved in PTP communication with others would need to manage its own list of “interested” peers, potentially requiring a system-wide reconfiguration. A notification-based approach (for background, see Section 2.4.3 on page 33) to communication solves these issues:

- Provision of a single public API for the publication and reception of messages: an individual component only needs to use one API in order to communicate with any other component. (see modularity and developer user requirements).
- Participants in the system require less configuration maintenance, since they only need to communicate with the notification service directly.
- Publishers only need to ensure correct delivery to the notification system. The notification system handles onward delivery to subscribers.
- The notification service manages topic-subscriber “interest” mappings, thereby allowing new subscribers to be added without updating publisher components.
- The notification system should take responsibility for logging messages from publishers, providing a provenance trail useful for debugging, or inspecting a system’s state (see maintenance requirements in Section 3.3.4 on page 54).
- New subscribers have access to the entire history of previously published messages, allowing them to be brought up-to-date with the current state of the system (see system extensibility requirements in Section 3.3.4 on page 54).

Grid systems are far more exposed to component failure than stand-alone systems. The addition of more software and hardware components to a system increases the risk of observing an individual component failure, either as a result of a software crash, or a hardware problem such as a network outage. If PTP communication between services is analogous to instant-message communication, then notification-based messaging is the Grid equivalent of e-mail. A notification system provides

greater reliability through the temporal decoupling of the message publication process from the message delivery process. For instance, a publisher sending events to a set of subscribers can continue to operate, even if one or more of the subscribers are temporarily unavailable. When the failed subscriber(s) becomes operational again, the ‘missed’ messages will be forwarded appropriately. In contrast, a [PTP](#) mechanism would fail in this situation since it requires both publisher and subscriber to be ‘on-line’ at the same time.

A Microbase system is composed of `core responder` and `domain responder` components. As described in Section [3.3.5](#) on page [56](#), a responder consists of at least one Web service-based server-resident component, and at least one mobile component that may be resident on one or more transient worker nodes. Server-resident components will typically be permanently registered, at largely static locations. Therefore, the preferred delivery method for server-resident components is `push subscription`. Due to the transient and mobile nature of worker nodes, `pull-subscription` is the only possible delivery mechanism.

To support these two situations, the Microbase notification system needs to support two different kinds of message delivery:

1. A post-style directed delivery system, that pushes messages to specific recipients at well-known locations, and
2. a public bulletin-board system where anonymous, transient entities can collect messages relevant to them.

It is not appropriate for *all* inter-component communications to be sent via the notification system. For notification-based communication to be appropriate, the message content should be lightweight and of public interest to other components. The notification system should be used when such messages need to be reliably delivered to decoupled subscribers and made persistent for future use by newly-added subscribers. Communications that involve large quantities of data, or for which low latencies are essential, are typically better served by direct [PTP](#) communication.

In Microbase, there are several categories of communication, only some of which are suited to notification-based messaging. Examples of where communications are suited to notification-based messaging are:

- The propagation of high-level state change information from a `responder` to other interested `responders`.

- The coordination and synchronisation between processes executing within responders . These messages are often implementation specific.

Types of communications that are not suited to notification-based messaging are:

- Queries for structured data stored by remote components.
- Queries for large data objects stored by remote components.

High-level state changes within components, such as ‘new data’ or ‘action complete’ announcements may be of interest to other components. Even if no component currently installed within a system is interested in a particular event type, a newly written or newly installed component may register an interest at some point in the future. These events need to be permanently archived in case a new responder interested in these message types is added to the system at a future time. Events of this type are ideally suited to being handled by the notification system, since message recipients are not necessarily known (and in fact, should *not* be known) by the publisher.

Coordination and process synchronisation messages between decoupled components must also be transferred via the notification system. However, unlike high-level state events that are exchanged by registered Web services, coordination and synchronisation requests typically need to be communicated among ‘anonymous’ entities that are not registered with the system. Examples of anonymous entities include transient worker nodes that join and leave the system unpredictably, or a domain-specific process migrating between worker nodes. Synchronisation operations are generally either time-dependent, highly implementation-specific or both. The necessity for a component to send these types of event may depend on the current state of a system, such as the current unavailability of a particular resource required by that component. Coordination messages are necessary for the system to operate, but there is nothing inherent in their content that is of interest to pipeline-level provenance. Although this type of message must be stored until delivery is complete, there is no requirement to archive them permanently.

Data queries to remote components are almost always implementation-specific; a component requires a specific kind of data from a specialised source that can provide it. For instance, a component requiring a list of unprocessed genome sequences needs to acquire that list from the component responsible for maintaining genome sequence information. There is no reason for other components to be interested in specific queries such as these; while a different component may also be interested in sequence identifiers, its list of ‘unprocessed’ items will most likely contain different items. Routing

domain-data through a Web service-based notification system would not be advisable in any case, since it would become an un-scalable single point of contention (see background Section 2.1.1).

For the reasons described above, data acquisition from remote storage is better handled by point-to-point domain-specific Web service transactions where structured data resources such as SQL databases need to be queried. Bulk data transfers, such as large files, are more appropriately achieved via the Microbase resource system (introduced in Section 3.4 on page 60, and described in detail in Chapter 5 on page 85), with the transfer being initiated by a lightweight point-to-point service call. Therefore, these types of component communication should *not* be routed via the notification system, and are not considered for the remainder of this section. The only types of communication that the notification system is required to handle are lightweight status update notifications, or lightweight requests to decoupled entities.

Point-to-point communication still has its uses, however. In cases where low latency is critical, messages are entirely implementation-specific and of no relevance to other components, or where there are so many subscribers that it would be infeasible to use the notification system. Direct point-to-point communication could be used, for instance, between tightly-coupled Web service components *within* a responder. These communication types are not relevant to the notification system, and will therefore not be considered further.

Although both high-level responder state messages and low-level coordination/synchronisation messages need to be routed through the notification system (for the reasons outlined above), their delivery requirements are very different. A high-level responder event represents a major state change of some kind, such as new data becoming available, or an event representing a successful or failed computation. These events are markers of important milestones and form points in a provenance trail linking distributed events in a causation graph. As such, it is essential that these event notification messages are archived permanently, ordered by time-stamp. It is also important that these messages are reliably delivered to subscribers in the correct order. If a subscriber is not able to receive a message for some reason, then the notification system must reattempt delivery at a later time.

Low-level coordination and synchronisation messages represent requests or notifications of state changes that are necessary for the correct functioning of the system. These messages are a means for components to send ‘housekeeping’ messages to other decoupled components. Examples of this kind of message include announcements of new configuration settings (such as Web service endpoints) and requests to make a resource available. Low-level messages are not directly related to achieving high-level milestones, but they are nevertheless essential for the correct operation of the system. In a typical deployment, low-level messages are sent at a greater rate than persistent,

high-level messages. However, the nature of the intended content of these messages is often time-dependent. The contract between publishers/subscribers of low-level messages and the notification system is therefore different from that of persistent messages. The emphasis is on delivery speed, rather than reliability.

4.3 Requirements

During the development of the notification system, the following component-level requirements were established to support Microbase system-level requirements:

1. Provide a decoupled communication mechanism to be used among a set of `core responders` and `domain responders`. This mechanism supports the modularity and maintainability of system-level requirements (see Section 3.3.4 and Section 3.3.5 on page 56).
2. Provide permanent storage for `responder state change` messages. These messages will act as a log of important milestones over the system's 'lifetime'. Storage of these messages contributes to system extensibility, and therefore satisfies the requirements described in Section 3.3.4 on page 54.
3. The ability to add new topic types, publishers and subscribers dynamically to a running system, facilitating the addition of new `responders` to an existing Microbase system, without the need to restart services. See extensibility system requirements in Section 3.3.4 on page 54.
4. The ability to reconfigure `push subscribers` at run-time, allowing the Web service components of `responders` to be migrated to new servers, without impacting message delivery. This contributes to satisfying system administrator maintenance requirements described in Section 3.3.6 on page 58.
5. Provide the ability to chain messages together: Messages should have a 'caused by' field that allows a message to state that it was published as a direct result of a preceding message. This ability is required to satisfy system-level logging provenance requirements (see Section 3.3.4 on page 54).

In order to support the notification system-specific requirements, the notification system must:

1. Support a 'reliable delivery' mode where: messages are guaranteed to arrive at each subscriber in publication time-stamp order; message delivery operations are repeated if a subscriber is

unavailable; newly-added subscribers are able to receive the entire history of notifications that are relevant to them.

2. Support a ‘no guarantees’ delivery mode: fast message delivery attempts, but no guarantees with regard to message ordering or delivery success.

These features are facilitators of seamless extensibility of a Microbase installation; each message topic becomes a potential point at which future responders could be attached.

4.4 Architecture

The notification system forms the center of a Microbase installation. The notification system is unique in that it is the only Microbase-core service provider that is *not* a responder. The notification system handles the messaging between responders, facilitating asynchronous orchestration of services. The notification system makes no distinction between `core responders` and `domain responders`.

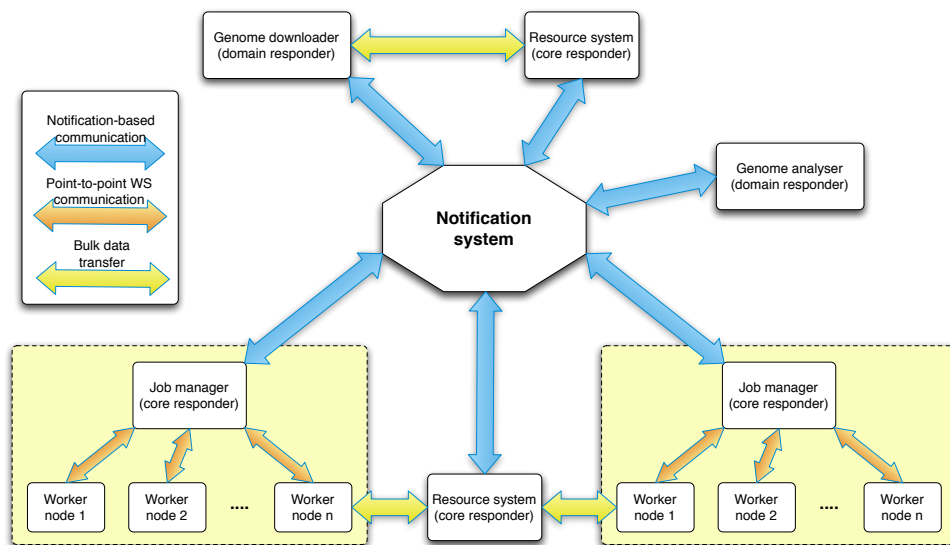
In the previous section (4.3), it was stated that the notification system must support two kinds of message delivery: high-level state change events and low-level coordination and synchronisation events. High-level events are termed `persistent messages`, due to their requirement to be stored as a permanent record of the system. Low-level coordination and synchronisation messages are termed `broadcast messages`.

4.4.1 Handling persistent messages

The centralised nature of the notification system has some important architectural implications. Figure 4.2 shows a typical scenario that consists of several `domain responders` interacting with other domain and core responders. In order to ensure that inter-component messaging does not become a bottleneck and to ensure reliable delivery, it is necessary to have well-defined semantics, defining the behaviour of the notification system and the components that interact with it. Contracts that specify component behaviour supporting the notification system requirements will now be explained.

The contract between the notification system and publishers is as follows:

- Messages are immutable once published



1. The “genome downloader” responder is responsible for injecting new data into the system so that it can be processed. This is a two-step process: firstly the data content is transferred to the “resource system” via a bulk transport mechanism; secondly, a lightweight message is sent to the notification system, announcing that a new sequence is available.
2. The “genome analyser” responder is registered to receive such events. It is responsible for determining the computational work that is needed as a response to the ‘new sequence’ event. Compute-intensive application(s) can be requested to run by sending a set of job descriptions as a new message to the notification system.
3. The job management system (core responder) reacts to this event by adding the requested computational work to a queue. Jobs will be scheduled when available worker node(s) become available.
4. There may be many hundreds of worker nodes; having them all contact the notification service directly would be intractable. Instead, point-to-point communication between the worker nodes and their assigned job manager service is more appropriate. Worker nodes contact a job manager for lightweight communications such as job descriptions, and completion reports. Large data resources, such as the sequence file, are bulk-transferred via the resource system.
5. Once all requested jobs have been completed, the job management system fires a notification stating that the work has been completed.
6. The ‘work completed’ event is sent back to the “genome analyser” responder. A further “analysis complete” message can then be sent.

Figure 4.2: Shows how the notification system might integrate domain responders with the rest of a Microbase installation. This figure shows where it is appropriate to use different kinds of communication to allow a scalable Grid system to be constructed.

- The semantics of a message sent with a given topic should not be changed between publisher versions. Present and future subscribers need to be able to interpret the messages. If different message content semantics need to be used, then a new topic name should also be used.
- The raw message format *may* be changed, but is not recommended. Assuming that the publisher provides a suitable message parsing library, the public API of this library never changes, and the library is able to parse all previous formats, then the underlying message format can be modified.
- Publishers are responsible for ensuring that any `topics` they require are registered prior to sending a message.
- Publishers are responsible for ensuring that messages reach the notification system. For instance, if the notification service is temporarily unavailable, they should resend the message until the notification system acknowledges receipt.
- The notification system guarantees that if it actively accepts a message from a publisher, that the message will not be lost (i.e., the message has been successfully archived).
- The notification system guarantees ordered delivery to each interested subscriber, in spite of sporadic notification system or subscriber failures.

The contract between the notification system and subscribers is as follows:

- Messages are guaranteed to be delivered to subscribers in the correct order.
- Messages will remain queued until the subscriber successfully accepts delivery of a message.
- However, messages *may*, in rare circumstances, be delivered to subscribers multiple times. Although the notification system has been designed to be tolerant of faults, the implementation “fails safe” in some circumstances. This results in the need to re-send messages that *might* not have been delivered, following a notification system server crash.
- Subscribers are required to acknowledge message delivery successes promptly. A message delivery attempt locks various system resources while a message is in transit to a subscriber, and while waiting for the subscriber to acknowledge receipt. An overly-long delivery time may be interpreted as a timeout, and therefore as a failed delivery attempt.

The responsibilities of subscribers are therefore:

- Subscribers should acknowledge successful message delivery as quickly as possible, or risk triggering timeouts which would be interpreted as message non-delivery. Therefore, subscribers are recommended to implement a local message spooling system so that message delivery and acknowledgement processes are decoupled from message processing activities.
- Keep track of and implement protection against duplicate message *deliveries*. The notification system may deliver a message more than once if it suffers particular types of failure¹. Subscribers should only react once to a message, ignoring duplicate delivery attempts
- Subscribers should also guard against the possibility of duplicate message *content*. If a publisher (mistakenly, or otherwise) sends distinct messages with identical content, this may have ill-effects if the subscriber naively processes the duplicate content². The notification system has no way of guarding against this situation since it does not parse message content. Again, handling this situation is optional, depending on the subscriber implementation. It is even possible that duplicate message content is required in some domain applications.

The notification system oversees message history and deliveries, but there is no requirement for it to interpret message content. Therefore, communicating responders must be able to parse each other's messages. The recommended way of implementing this is for publishers to provide an appropriate decoder library so that subscribers are not required to parse the raw message themselves. Message content is not within the scope of the notification system specification. The content of specific messages will be described in subsequent chapters.

4.4.2 Handling broadcast messages

Figure 4.3 shows the role and types of participants involved in message broadcasts.

Broadcast messages are sent between decoupled components when low-level, time- or implementation-dependent requests or status updates need to be sent. Anonymous components not registered with the notification system can also use broadcast messages via pull-subscription as a type of public “message board”. Messages are not guaranteed to be delivered in order, and in fact, are not guaranteed to be delivered at all. However, they are a quick and convenient way to send configuration information anonymously in a loosely-coupled fashion. Broadcast messages may be kept for a configurable

¹For instance, if a hardware or software failure occurs between the point of a successful delivery to a subscriber, and before the associated database transaction (indicating delivery success) within the notification system is committed.

²Such as duplicated work, results, or worse, inconsistencies in results.

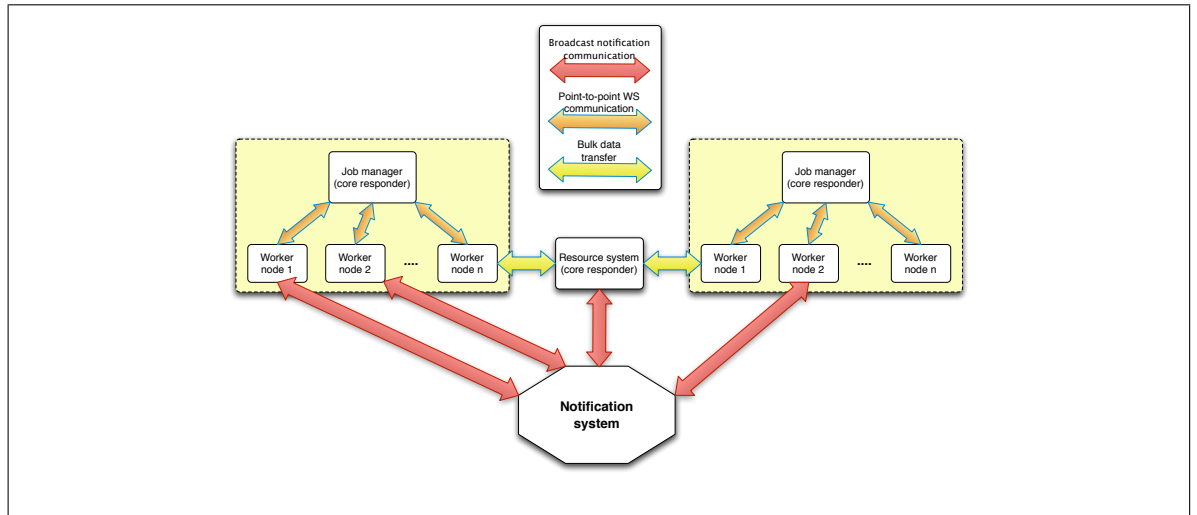


Figure 4.3: Shows how decoupled messaging can be achieved between registered subscribers, and anonymous, unregistered entities (such as worker nodes). The diagram depicts how bulk data transfers between cooperating processes on a set of worker nodes might be initiated with notification event. The advantages of notification-based messaging are maintained: components only receive the messages they are interested in, or in the case of anonymous subscribers, the message types they request. For instance, in this case, the “job manager” responder does not receive resource transfer requests, because this type of message is irrelevant.

amount of time within the notification system, but may be deleted after this time has elapsed. They are certainly not kept forever in the way that persistent notification messages are.

The contract between the notification system and distributed components is as follows:

- Messages will be delivered to a subscriber in a ‘best effort’ fashion; i.e., message delivery will be attempted a certain number of times, after which no more attempts will be made.
- There is no guarantee that a particular message will be delivered to a subscriber.
- There is no guarantee that messages will be delivered in the order that they were sent.
- Messages will be stored within the notification system for a ‘short’ period (minutes to hours). During this time, they can be collected by pull-subscribers. After this time has expired, broadcast messages will be deleted.
- Therefore, ‘important’ announcements should be sent periodically, until the publisher can determine that the message has been received by an appropriate subscriber, and that the requested action has taken place.
- Anonymous message sending is permitted. Messages may need to be sent anonymously not for security reasons, but because the originator is not necessarily registered with the notification system.

There are a vastly larger number of potential participants needing this form of messaging, than ‘persistent’ reliable messaging. From a scalability standpoint, a single central notification server may not be sufficient to accommodate this load. However, replication (multiple instances) of a broadcast notification system are permitted, due to the relaxed delivery requirements of ‘broadcast’ messages compared to ‘persistent’ messages.

4.5 Implementation

The notification system is implemented as a set of XFire Web services.

- Administration service: manages publisher, subscriber and topic registrations
- Publisher service: allows publishers to send messages
- Message service: enables pull-subscribes to collect messages
- Push-subscriber interface: Enables developers to write a Web service capable of receiving messages from the notification system

The process of receiving a message from a publisher, through to message delivery is shown in Figure 4.4. A notification message may be in one of three states:

- Not sent: indicates a new message
- Sending: indicates that the message may have been delivered to some subscribers, but there is at least one outstanding delivery
- Sent to all: indicates that all interested subscribers have acknowledged receipt of the message

When a message is first received from a publisher, it is archived to a permanent message store, and its state is set to “not sent”. If there are no interested subscribers, the message state is immediately updated to “sent to all”. In this case, it will remain in this state until a subscriber registers an interest.

However, if there is at least one interested subscriber, the message state is changed to “sending”. At this point, a delivery queue is populated, consisting of one entry per interested subscriber. The delivery queue is emptied as successful deliveries are made. Once the delivery queue is empty, the message state is updated to “sent to all”. The message will remain in this state until another subscriber registers an interest.

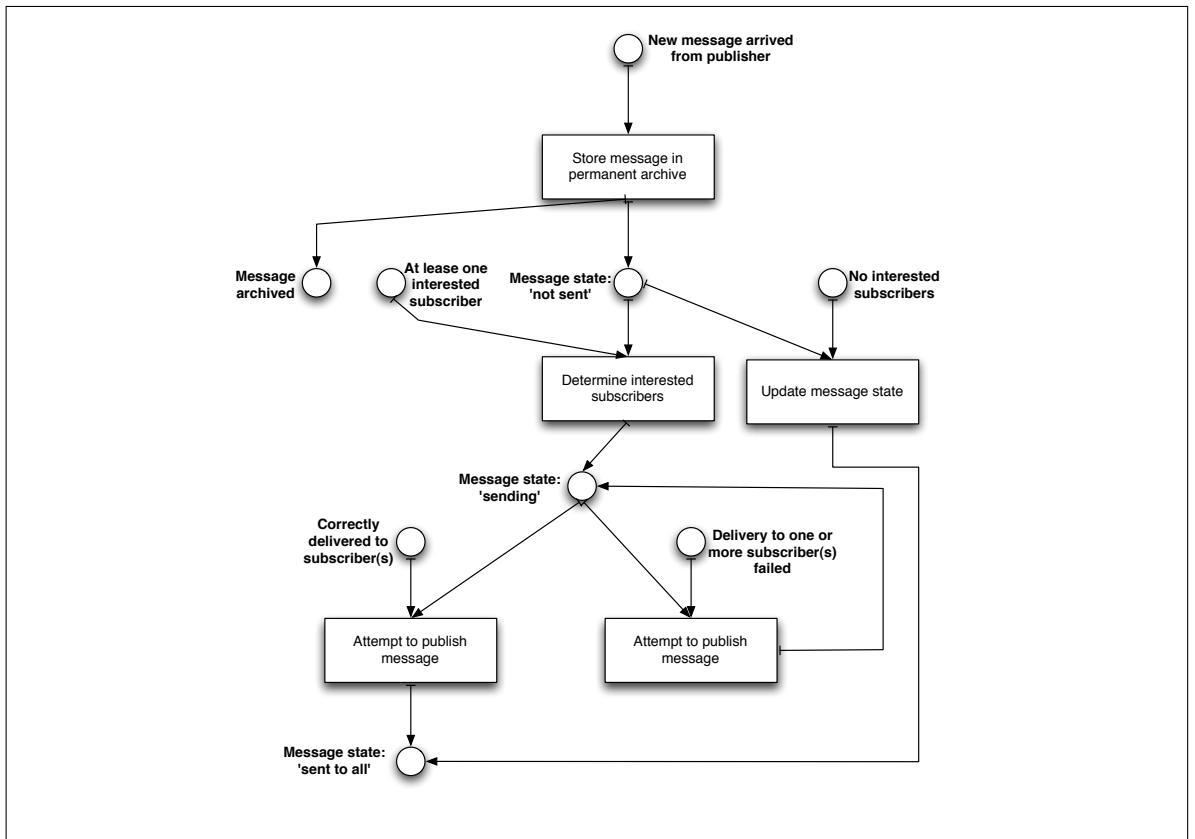


Figure 4.4: Notification system: message delivery states. New messages arriving at the notification system Web service are permanently archived for future use. The topic of the incoming message is analysed in order to determine whether any subscribers have registered an interest in this type of message. If no subscribers are interested, the message is set to state ‘sent to all’. If there is at least one subscriber, delivery is repeatedly attempted until the subscriber accepts the message or a retry limit is reached. Once the message has been delivered to all interested subscribers, its state is set to ‘sent to all’.

4.6 Conclusion

The Microbase notification system provides de-coupled inter-responder communications in one of two ways: a reliable, persistent delivery mechanism; and a transient message broadcast system. Both delivery methods are required for the correct functioning of a Microbase system. Persistent messages are sent between the Web service components of `responder`. These communications must be logged and permanently stored in order to facilitate future expansion of the system, where a newly added `responder` may be interested in receiving previous event notifications. All business-logic messages should be sent using the reliable delivery mechanism for this reason.

In contrast, the transient message delivery system is intended to permit de-coupled communication between unregistered, potentially anonymous subscribers such as worker nodes. The transient delivery mechanism is intended to allow machines to broadcast runtime configuration information such as the announcement of the presence or location of a particular resource. This delivery mechanism may also be used for sending messages between co-operating processes on different worker nodes.

The notification system as discussed in this chapter implies a centralised server architecture. Distributed notification systems exist [164, 190] that are inherently more scalable than a centralised system. The advantage of such systems is the ability to propagate many large messages to hundreds or thousands of recipients through the use of P2P techniques. However, obtaining provenance information such as accurate publication times and therefore guaranteeing message ordering from a P2P system is more difficult. Message delivery failure is also much more difficult to detect in distributed notification systems due to the need to back-propagate ‘acknowledgement’ or ‘time-out’ messages. In light of the requirements placed on the Microbase notification system, we have adopted a centralised notification system. We do not believe that the notification system will be a significant bottleneck since the number of responders installed in a typical system is low — tens rather than hundreds. Given the high-level nature of the persistent messages being sent between responders, the rate at which messages are published should be relatively low. Messages should also be ‘small’ — less than tens of megabytes — since large bulk data transfers should be routed via the Microbase resource system (see Chapter 5 on page 85) rather than the notification system.

Worker nodes do, however, frequently send broadcast messages. Many worker nodes communicating frequently enough would cause a single notification server instance to become a bottleneck. However, broadcast messages are not subject to the same archival, and delivery guarantees as persistent messages. Therefore, it would be permissible to use multiple notification system instances for processing high-frequency broadcast messages. A P2P notification system would also be suitable for

this purpose.

Chapter 5

Resource system

5.1 Introduction

Resource storage systems have several areas of responsibility: file distribution, permanent file archival and version control. Distributed computation systems typically have a central data store that permanently archives all information related to computationally-intensive project(s). A large computational task is split into smaller units. If the task splitting is suitably performed, then large data sets can also be split into manageable chunks for distribution to worker nodes. Each worker node receives only the data it requires to complete its unit of work. It is the responsibility of the data storage system to ensure that worker nodes have access to the appropriate files and that the files are stored reliably.

In a Grid setting, the real time (i.e., the perceived elapsed time from a user's perspective) required to complete a large computational task is the sum of the time taken to perform user computation, plus the overhead of the Grid system. In data-intensive applications, a sizeable portion of the overhead may be caused by large data transfers between the system's permanent data store and the worker nodes. Therefore, in addition to storing files reliably, the resource system is also responsible for efficient distribution of files to remote hosts, such that the overhead time is minimised. The resource system described in this chapter acts as the 'file system' for Microbase responders.

5.2 Motivation

5.2.1 Data identification and storage

Multiple communicating nodes participating in the system must be able to acquire data resources from each other. The nodes therefore require a common naming scheme to resolve resource content. The resource system for Microbase must be capable of uniquely identifying files. The identifier assigned to a resource must never change, and the content of a published resource must also be immutable once it has been assigned an identifier. These requirements are broadly consistent with other naming schemes, such as Life Science Identifiers (LSIDs) [55].

Unlike an LSID, however, once published a resource will be expected to be available for download indefinitely if requested. This can be achieved either with a centralised storage system using dedicated hardware, or a distributed system using commodity hardware. ‘Centralised’ here does not necessarily mean a single server since a set of servers could be combined into a single logical service. A centralised system storing resources for Grid applications provides several advantages:

- Isolation: distributed worker nodes can perform their work in isolation, using their own local storage, rather than higher-latency and possibly contended access to remote resources.
- Straightforward access to results: querying a single (logical) location is easier than querying several distributed data stores, depending on the number of simultaneous queries, their complexity, and the amount of data needing to be transferred.
- Reliability: it is straightforward and convenient to take backup, snapshots or replicate a centralised store. Also, Microbase utilises ‘unreliable’ worker nodes for computational work. Permanently storing data on worker nodes would risk data loss.

In addition, it would be inappropriate for a Microbase system to permanently store large amounts of data on worker nodes. Many worker nodes are likely to be general-purpose desktop PCs and the primary users of these computers require the available disk space for their own use. Therefore using a distributed storage approach for permanent data archival is unsuitable for Microbase.

5.2.2 Data distribution

Data transfer mechanisms are an essential part of any distributed computation system. Data must be distributed to worker nodes from a permanent storage location, processed, and then finally the results

must be transferred back to the permanent data store. The background chapter (see Section 2.3 on page 27) discussed several possibilities for data transfer between nodes in a distributed system:

- File transfer mechanisms such as FTP, HTTP, WebDav.
- Network file systems, such as Server Message Block ([SMB](#)) or Network File System ([NFS](#)).
- Direct access to structured data storage such as a relational database client.
- Distributed file transfer systems such as BitTorrent.

Centralised file transfer protocols such as FTP provide low-latency access to remote resources. The entire file must be acquired by the downloading node before it can be used. Once downloaded, the node has its own local copy of the file, which can be written to without affecting other nodes. However, the server's available bandwidth must be shared between all downloading nodes, reducing the scalability of the system. Mirror servers may be added to a system, but typically require (expensive) dedicated servers, due to the disk space and network capacity requirements. Also, mirror servers may not be able to handle bursts of high activity efficiently, such as many nodes requesting the same file simultaneously — for instance, when thousands of similar jobs are scheduled.

There are several properties of a distributed computation system that suggest that a [P2P](#) transfer protocol may better fit for the data transfer requirements than a centralised system:

- There are a large number of worker nodes active at a given time.
- Given the notification-based pipe-lined approach employed by Microbase, there will be a large number of jobs of the same type, scheduled within a short space of time. Therefore, there will be large demand for the same set of files within a similar timeframe.

Under these conditions, it is likely that a number of worker nodes will be processing the same type of job simultaneously, and therefore require access to the same executable resources, such as Java Archive ([JAR](#)) libraries and executable program files. It is also possible that there will be some overlap regarding required data files, depending on requirements of the application. The transfer performance of distributed protocols such as BitTorrent improves (see Section 2.3.1.1 on page 29) as the number of participants actively transferring a file increases [134]. If worker nodes utilised such a distributed transfer mechanism, then as more nodes started a particular job type, there would automatically be more nodes available to acquire the resource(s) from. After the central data store transfers an initial copy of a resource to a remote node, such a system would dynamically scale to fit the number of worker nodes.

5.2.3 File version control

In order to satisfy Microbase provenance and logging system requirements (see Section 3.3.4), all published resource files must be immutable. Immutability of published data is not seen to be over-restrictive since analysis results do not change after the execution of a program. File immutability is a property of other distributed file storage systems since it is a straightforward means of guaranteeing data consistency with little server overhead other than to ensure the uniqueness of file identifiers[76]. If resources were allowed to change after publication, then the system would suffer from several undesirable side-effects including:

- the loss of an accurate record tracing the production of data items back to a specific system component.
- inconsistencies arising from unexpected concurrent updates to a resource from a different responder. This situation is analogous to un-synchronised access to shared memory in a multi-threaded program. For instance, if a resource update occurred simultaneously with the distribution of that resource to a number of worker nodes, then different worker nodes could find themselves unknowingly using different versions of the resource.

Some resources do need to *appear* to change over time. More specifically, the resource files themselves are immutable, but a *reference* to the latest version of a resource must be provided to allow system components to query the latest version of a resource at run-time. Although this may seem an extravagant use of disk space, it is necessary for the reasons outlined above. For instance, a new version of an application may be released, requiring the ‘replacement’ of the old application resource. However, the older version(s) must still be present within the system in case some system components require an older version. For example, if a new application version produces output files of a different format to the previous version, then some system components may still expect to receive data in the older file format. A resource system supporting version control provides additional benefits: users may wish to run several versions of an application concurrently in order to compare the different outputs for equivalence or debugging reasons. Conversely, users might want to execute multiple versions of a data set through a particular application version. The data management aspects of these types of activities are generic, and therefore should be facilitated by the Microbase resource system.

5.2.4 File querying

Worker nodes in a Microbase system are transient; they can join or leave the pool of available compute power at any time. They are also generic desktop PCs, with a campus-wide ‘common desktop’ environment. Domain-specific software that is not pre-installed will need to be installed to available computers prior to any computation work being performed. Due to the transient existence of these machines as Microbase participants, it is likely that software installations will not survive beyond a compute client termination. Operating in this kind of environment puts additional pressure on the data distribution sub-system; not only do multiple worker nodes require applications, often large ones, to be installed, but the process may have to be repeated several times per day.

Using a distributed peer-to-peer transport protocol alleviates several of the logistical and scalability issues involved with bulk data transfers required when installing domain-specific applications to multiple machines several times per day. However, a P2P transport mechanism alone does not resolve platform-related issues of the heterogeneous worker nodes. Although a `domain responder` has knowledge of the specific version of an application required to perform computational work, it cannot know ahead of time the operating system or processor architecture used by the worker node assigned to process the work. Specifying resources by unique identifier (UID) is acceptable for data resources, but is not suitable for platform-specific executable resources. To overcome this problem, it is necessary for worker nodes to be able to query for appropriate resources at run-time, based on the nodes’ actual hardware and software composition. The resource system must therefore provide query-able annotations attached to resources in order to facilitate such queries.

Annotations are also useful in the wider context for attaching information commonly required of file-systems, such as timestamps, version tags, and file content type information to resources. Some of these annotations are required for operational purposes, while others may be useful for debugging a running system. Resource annotations also contribute resource-specific information to the overall provenance data set maintained by the Microbase system as a whole. It may also be useful for `domain responders` to add their own domain-specific annotations to resources, in addition to the general-purpose annotations. The resource system must allow querying of resources, based on the presence or value of particular annotations.

5.2.5 Pipeline extensibility

Domain-specific applications will often store result data in a structured storage system, such as a relational database. This allows efficient access to result data, and flexible querying. For instance,

querying Blast hits from a set of raw alignment text files is much less efficient than performing a simple query to a relational database table. However, the trade-off is that in order to query a relational database, its structure and semantics must be known. While custom relational databases for domain-specific data may be required to efficiently query data, in order to satisfy the Microbase extensibility requirements (see Section 3.3.4), a more generic storage system is also needed. The resource system can assist in fulfilling this requirement (Figure 5.1 on the next page).

Suppose a computationally-intensive application generates a information-rich, but difficult to parse output file. If only a small portion of the output is actually useful for *current* purposes, then it would seem appropriate to parse the currently relevant information to a structured storage system and discard the original report. However, if a new application is later added to the system that requires access to the output data of the first program, then two data issues arise. Firstly, the new application is unlikely to be able to query a custom data store, unless it was specifically written to do so. Secondly, the new application may require additional information from the original output data file that is not stored in the custom structured data store. Although the new application may be capable of parsing the output files generated by the original application, if the original output files had been discarded, then the computation would need to be repeated to re-generate the files.

The resource system is ideal for storing the raw, un-parsed output files produced by applications. By archiving these files on a suitable 'reliable' archive node, the existence of the data can be guaranteed. Using a P2P transport mechanism, the archived resources are available for mass-transfer at a later time. In conjunction with the logs maintained by the job management system (see Chapter 7 on page 134), the creation of a resource can be traced back to job execution that produced it. Therefore, expandability of the system is improved at the expense of disk space storage for storing an additional copy of raw output data.

5.2.6 Developer usability

The primary task of a domain-application developer is to write applications that are useful to their area of interest. They may view the resource system as necessary for resource distribution, but do not necessarily have the time to gain an in-depth knowledge of its internal workings. However, distributed file protocols, queue management, querying, and meta-data annotations all add complexity to the resource distribution process. If the resource system is too intrusive to the application development process, then developers may choose not make use of the provided capabilities, negating the value of the system. The resource system must be straightforward to use, preferably not passing on

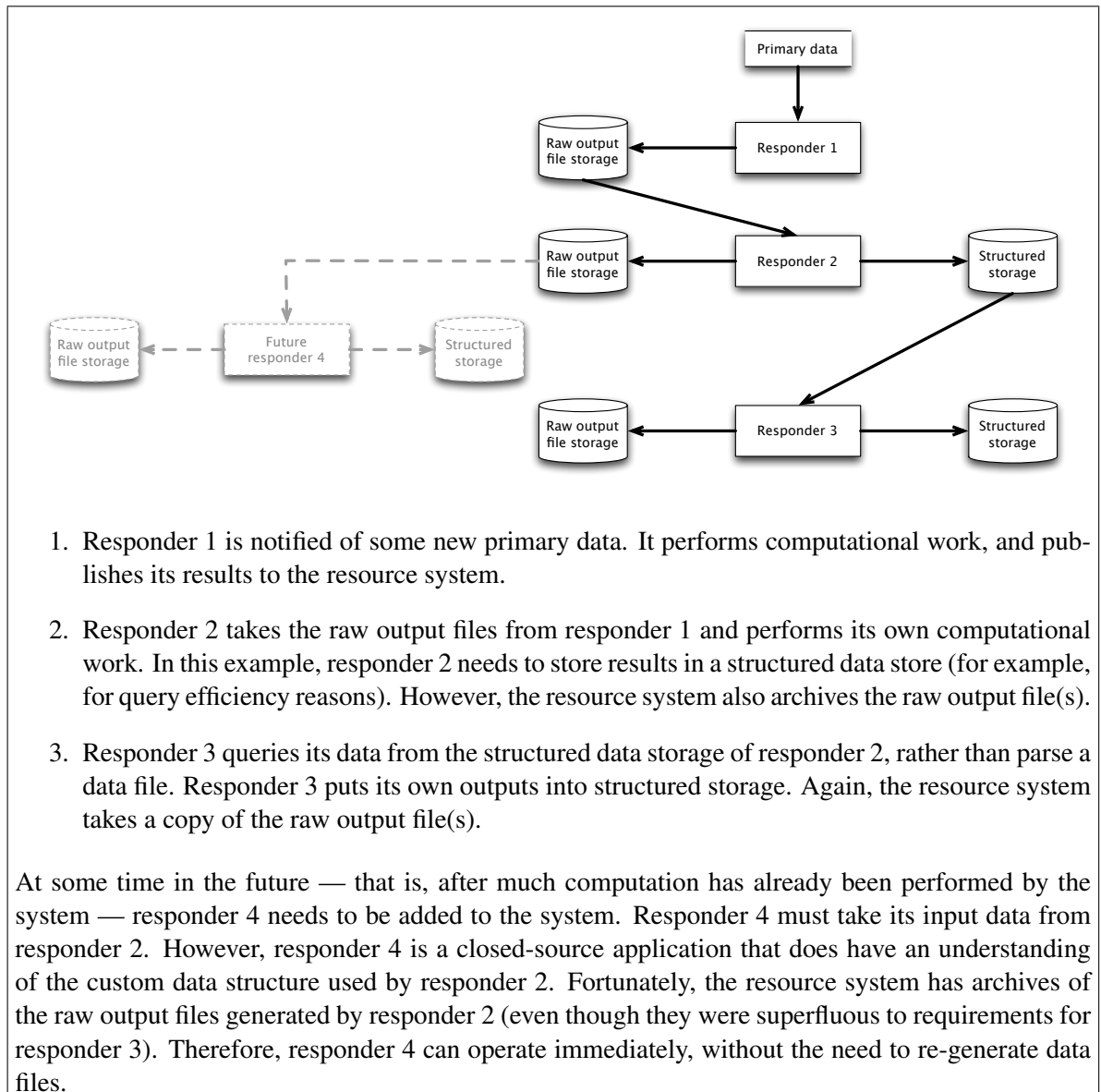


Figure 5.1: Shows how the resource system enables domain responder pipeline extension. The resource system is responsible for storing the raw file outputs as a result of responder computation. Responders are responsible for maintaining their own custom structured data store, if they require one. In this example, it is assumed that the computational work performed by each responder generates one or more data files. This is typically the case when running command line applications. The content of the data file(s) may be parsed into a structured storage system if required.

implementation-specific complexities to the application developer. For instance, domain application developers should not be required to have in-depth knowledge of the transport mechanisms used. Ideally the system should be no more complex than accessing a local file-system. The same is true of the data archival and version capabilities; the resource storage system must be as un-intrusive as possible for developers, particularly in cases where permanently archiving raw data output files are of no immediate benefit for them. The resource system has been designed in conjunction with the responder development guidelines and API (see Chapter 6 on page 107) to achieve these requirements

(see Section [3.3.6.1](#)).

5.3 Requirements summary

The Microbase resource system is a support service that is intended to provide reliable and scaleable data-handling facilities for the rest of the system. In addition to its data distribution duties for worker nodes, it plays an important role in enabling system-wide requirements such as long-term storage, provenance and version control support, to be met. The resource system has been designed with data distribution-specific and Microbase system-wide requirements in mind.

The resource system is also required to assist with providing system-scale properties in conjunction with other core components. It must:

- Support Grid execution on a collection of heterogeneous hardware, in conjunction with the job management system (Chapter [7](#) on page [134](#)). Specifically, it must support efficient transfers to worker nodes, and provide the necessary support infrastructure to resolve issues arising from the requirement to handle a heterogeneous set of worker nodes. In addition, problems arising from worker node failures, such as data loss must be handled gracefully.
- Avoid single points of failure and single points of contention.
- Contribute to the overall system-wide provenance trail by providing file version control support and meta-data annotations.
- Publication logs should be kept in the form of notification messages. These messages should allow the resources to be traced to the responder that published them. Using the notification system (Chapter [4](#) on page [69](#)) to achieve this also facilitates the system extensibility requirement (see Section [3.3.4](#)), since future responders may need to be informed of resource publication history.
- Assist developers by hiding details such as torrent files, distributed copy counts, and availability as much possible (in conjunction with the responder developer environment, Chapter [6](#) on page [107](#)).

For the resource system to satisfy its obligations to the system-wide requirements, it must:

- Store data and executable resources, and dependencies including: platform-neutral Java classes, and associated dependencies as well as platform-specific application packages.

- Assign unique identifiers to each immutable resource.
- Allow resources to be obtained by resolving their unique identifier.
- Allow resource meta-data annotations to be queried in order to resolve matching resource [UIDs](#).
- The use of a peer-to-peer protocol to enable efficient transfer of resources among worker nodes.
- Integrate with the notification system (see [Chapter 4](#) on page [69](#)) to provide appropriate notification messages for resource archive success or failure events.
- Provide an [API](#) that masks as much of the underlying transfer mechanisms as possible.

5.3.1 Terminology

Resource file In Microbase, `resources` are coarse-grained immutable blocks of data; that is files, rather than table rows. The resource system is intended to transport medium to large resources — in the range of megabytes to gigabytes. Resources may be transferred among worker nodes and archiver nodes.

Resource [UID](#) a 128-bit [UID](#). This is the resource system’s equivalent of a file-name.

Tag key/value pairs that can be used to annotate a resource. Resources can be annotated with multiple tags. This enables categorisation of resources for organisational reasons. It also enables querying of the resource system at runtime if the exact `resource UID` is not known.

Archive node A machine running an instance of a resource system Web service that guarantees to provide an amount of permanent, reliable capacity for archiving files.

Permanent storage refers to storage capacity provided by archive nodes.

Cache storage refers to temporary storage capacity available to a worker node.

5.4 Architecture

The resource system operates in conjunction with the Microbase notification system in order to provide a reliable storage distribution system. The overall architecture of the resource system is shown in [Figure 5.2](#) on the following page. It consists of several parts:

- Torrent registry (Web service): maintains a registry of '.torrent' files and their associated annotations. This service allows querying over archived resources.
- Archiver node (Web service): a server-based component responsible for distribution and permanent archival of resource files.
- A client library and developer API: maintains node-local incoming and outgoing queues of resources. The library is capable of communicating with various resource system services. It provides a convenient API through which user applications may gain access to the resource system.

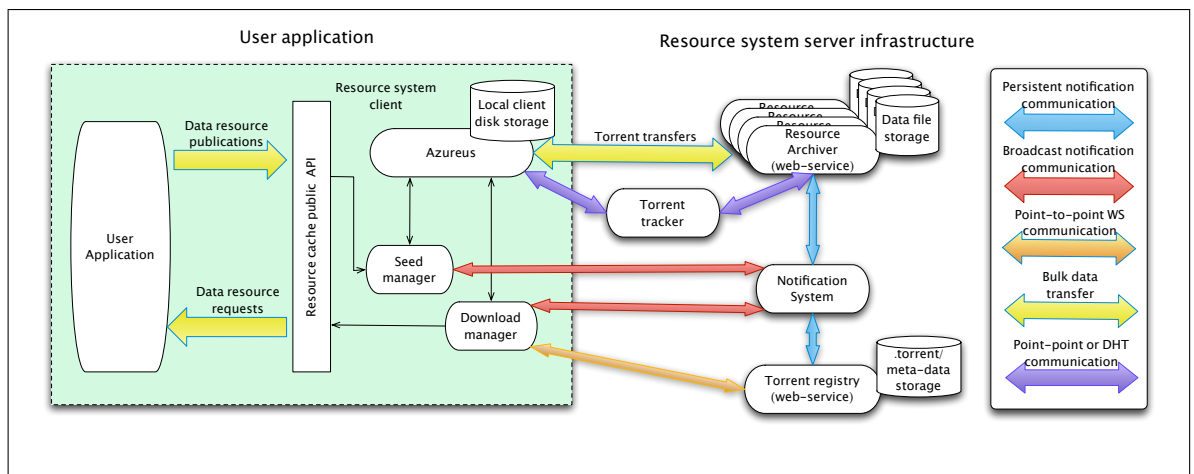


Figure 5.2: Resource system architecture - applications make requests to the resource system via an API that hides the actual details of file transfers and communication with remote web services. This figure shows how the various components that comprise the resource system interact with each-other, and how they interact with other core Microbase services.

The various components and the types of interactions between them will now be discussed.

5.4.1 Bulk data transport protocol

The BitTorrent protocol is used for bulk data transfers between nodes within the resource system. The BitTorrent protocol has been shown to be both scalable [89], and practical in terms of hardware resource usage [62]. Specifically, the Azureus¹ BitTorrent implementation was chosen to provide the underlying transport mechanism. Azureus offers several advantages to Microbase over other BitTorrent implementations. Its distributed tracking capabilities are useful in providing a scalable, multi-site transport mechanism. Although a centralised set of trackers are recommended to minimise peer resolution latencies, Azureus also supports a distributed tracker that can be used either as a

¹Azureus is now known as Vuze. It is available from: <http://azureus.sourceforge.net>

backup to dedicated tracking servers or as a load-balancing mechanism. The distributed tracker is based on a distributed hash table database. A portion of this database is stored by each Azureus node. The distributed tracker works across the Internet (firewall permitting), potentially enabling a compute Grid to transfer data among nodes located at multiple geographically distant sites in a decentralised manner — that is, without the need for ‘well known’ tracker server URLs. The majority of worker nodes available to Microbase reside in common cluster rooms with restrictive network access policies. The ability of Azureus to ‘reseed’ its distributed database from a user-specified location allows restricted nodes to access the external distributed database via a ‘gateway’ node less-encumbered with firewall rules, located elsewhere within the campus LAN.

Additionally, Azureus is available under a suitable open-source license that permits access and modification to the source code. Several modifications were required to enable handling large numbers of torrents and to accommodate network peculiarities of a campus environment. Finally, Azureus is written in Java, allowing it to be more easily integrated into the rest of the Microbase system.

5.4.2 Resource client API

The operation of the resource system depends on many low-level interactions between intra-resource system components and several high-level messaging operations to external components — that is other `core responders` or `domain responders`. Management of BitTorrent transfers and the storage of resource meta-data adds to the complexity.

The resource system client (Figure 5.3 on page 97) has been designed to serve several purposes:

- **Developer interface:** provides a straightforward API for developers to acquire and publish resources in their programs.
- **Abstraction layer:** provides an abstraction layer over the BitTorrent transfer protocol.
- **Scalability:** manages upload and download queues for user applications.

The client API provides facilitates publishing and retrieving data, and querying for the existence of resources. This API operates in terms of resource **UIDs** and annotation tags, rather than torrent files and magnet URLs. For instance, to retrieve a resource, an application does not need any knowledge of the underlying BitTorrent communication mechanism. The API also provides a means for querying resources based on annotations attached to resources.

The abstraction layer sits between the public interface that developers see, and the underlying BitTorrent implementation. Requests from the high-level interface in terms of resource **UIDs** and standard

Java constructs such as ‘files’ and ‘streams’ are mapped to torrent files and placed into transfer queues. The abstraction layer also provides a local disk cache so that frequently-requested resources do not need to be continually downloaded. If an application requests a file that is already present in the disk cache, then the application is informed immediately that the ‘download’ has been completed. Any resource file present in the disk cache is potentially available for ‘seeding’ to other resource system nodes. Seeding files to remote nodes occurs automatically and without the knowledge of the user application. This functionality allows nodes to transparently participate in a Grid distribution network.

The resource cache implementation provides concurrent asynchronous upload and download capabilities. Requests from the user application must be managed appropriately to avoid excess resource consumption. If a user application were to submit hundreds of resource download requests within a short space of time, system resources such as RAM and network ports could easily be entirely consumed, resulting in decreases in efficiency or even node failure. Node-local system resources must be managed in such a way as to ensure that the node receives all its requested files at the maximum rate that available hardware resources allow, for instance through transfer queues and caps on the maximum simultaneous active transfers.

When allocating local system resources such as network bandwidth, the distribution requirements of the Grid as a whole must be considered in addition to the needs of individual nodes. Nodes should allocate a portion of their available network bandwidth to sharing resources with other nodes, even if they have a large queue of outstanding downloads. Resource system nodes would become net bandwidth consumers if they were to download resources only, without providing any means of sharing data with other nodes.

Although a small number of ‘selfish’ nodes can be accommodated, if every node participating in the network were to behave in this manner, resource distribution bottlenecks would arise. Even though a distributed peer-to-peer protocol is in use, the system would essentially be reliant upon a small set of seeding distribution points if nodes refused to share resources after obtaining them. In cases where many nodes are downloading a similar set of resources, it is essential to maintain balanced incoming and outgoing traffic flows, even under high-demand situations. The responsibility for this balance lies partly with the Microbase-specific client implementation and partly delegated onto the underlying Azureus BitTorrent implementation.

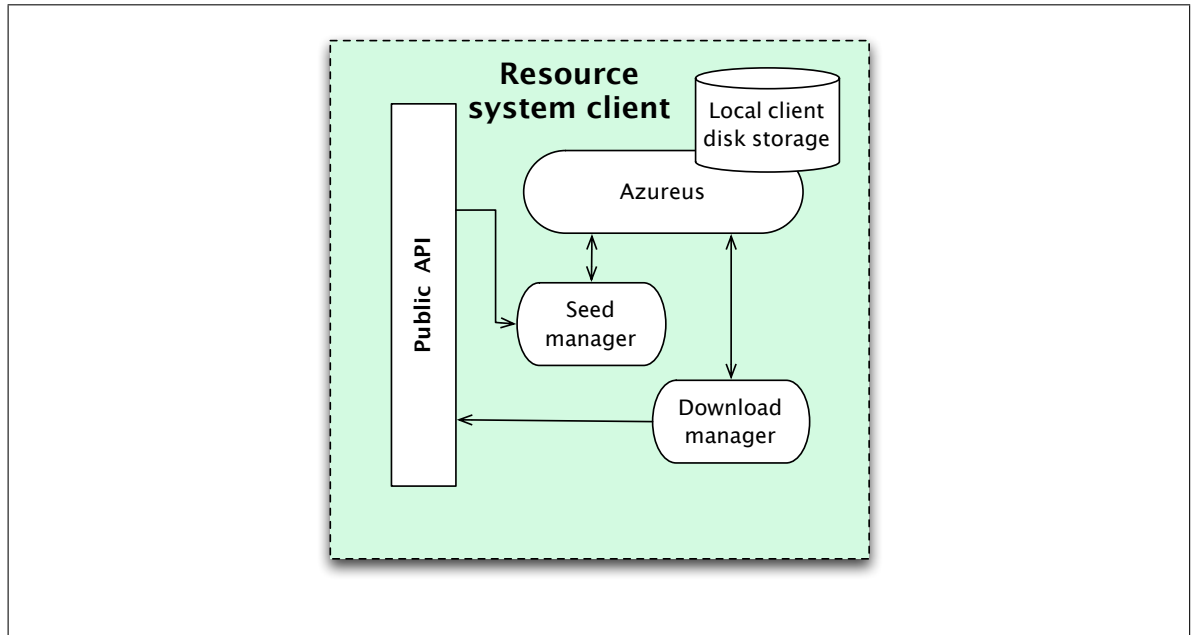


Figure 5.3: Resource system client architecture. The Microbase resource system consists of many such clients, usually one per computer. Client instances may communicate with each-other in two ways. Lightweight events may be sent via the notification system for coordination purposes. Bulk transfers are achieved by direct peer-to-peer communication between Azureus components. The download and seed managers translate high-level resource download/upload requests from a user application into transfer-specific BitTorrent requests.

5.4.2.1 Azureus-Microbase integration

Azureus was originally developed as a BitTorrent client intended for desktop use, where users will typically not have hundreds or thousands of torrent files queued, and certainly nowhere near that amount active at any one time. Every torrent item queued within Azureus consumes system resources, whether they are actively transferring or not. For instance, system memory and network connection resources are used to periodically perform housekeeping operations such as *scraping* (see background Section 2.3.1.1 on page 29). The more torrents that are queued within Azureus, the more system resources it consumes. Microbase nodes — archive nodes in particular — must be able to support many thousands of inactive torrents, and several tens of actively transferring torrents. Clearly, the Azureus queue alone would not scale sufficiently to be able to support the intended use.

To address this problem, Microbase implements its own queuing mechanism in addition to the Azureus queue (Figure 5.4 on page 100). For this purposes of this discussion, this additional queuing system will be termed the ‘Microbase queue’. The Microbase queue is intended to contain *all* torrent items located on a computer. It is required to have a static memory footprint, regardless of the number of torrents it contains. Torrents can be in one of two states: ‘idle’ and ‘active’. Torrents that are present only in the Microbase queue consume no system resources, other than disk space, and

are considered 'idle'. A subset of the items contained in the Microbase queue can be placed into the Azureus queue when data transfer with other networked nodes needs to take place. Torrents present in both the Microbase and the Azureus queue are considered 'active' with respect to the Microbase queue.

The Azureus queue may contain torrents in several states. The main Azureus states of interest here are: 'downloading', 'seeding', and 'queued'. The states 'downloading' and 'seeding' indicate that the torrent is actively transferring data to at least one other node. The state 'queued' indicates that no other node is presently connected to that torrent, but Azureus is actively seeking for nodes to acquire the file from, or nodes that wish to acquire the file. Although each item in the Azureus queue consumes system resources, there is an advantage in keeping it populated with as many items as possible. If torrents exist within the Azureus queue, and are in state 'queued', Azureus can decide to spontaneously start them (that is, change their state to 'downloading' or 'seeding') if it detects that they are needed by a remote node. This is a useful form of load-balancing since Azureus can detect the number of other nodes presently sharing or downloading a particular file and can adjust its own queue states either to take advantage of good file availability, or to assist other nodes when a file is scarce.

In order to download or share a file, it must be added to the Azureus queue. Since there are potentially a far greater number of files in the Microbase queue than can be accommodated in the Azureus queue, there is a need to manage the number and type of 'active' queue items such that:

- Incomplete files eventually get a chance to complete. That is, if downloads are made 'inactive' for some reason — for example, after a server restart, or download prioritisation — then they need to be reactivated.
- Completed (seeding) files remain seeding for as long as possible in order to increase the availability for other nodes downloading the same file.
- Completed, 'inactive' files may need to be reactivated if another host requires access to them after having been removed from the Azureus queue.

The management processes described above are performed by the 'download manager' and 'seed manager' parts of the resource system client (See [Figure 5.3 on the preceding page](#)). The two management processes cooperate to ensure the balance between downloading and seeding files is maintained.

The download manager is responsible for ensuring that incomplete files are eventually completed. It does this by periodically checking the progress of each incomplete file and taking appropriate action. Incomplete ‘inactive’ files are added to the Azureus queue if necessary. If an ‘active’ download is not making progress due to an inadequate number of peers to acquire data from, then a broadcast message is sent to the notification system to request that other peers start seeding the file.

The seed manager is responsible for ensuring the locally-produced files, such as results of computation, are available to be collected for archiving. This is important since locally produced files are, by definition, the only available copy on the network. If the results are lost — for instance if the node is removed from the pool of worker nodes — then the computation will need to be repeated at a later time. The seed manager is also responsible for ensuring that the node plays a part in the overall distribution of resources to nodes. The seed manager responds to notifications from other nodes requesting that a particular file is made more available. If the requested file exists locally, then the seed manager may decide to honour the request by sacrificing a less sought-after torrent.

Balance between the numbers of actively downloading and seeding files is maintained by Azureus. System-wide resource availability balancing is overseen by the resource system client, and enabled by decoupled broadcast events delivered by the notification system.

5.4.3 Torrent registry

The torrent look-up service maintains a mapping of resource **UID** to torrent data. It also maintains a set of annotations on resource entries in the form of key-value pairs (tags). Meta-data tags can be used to annotate a resource, and resources can be annotated with multiple tags. This enables categorisation of resources for organisational reasons. It also enables querying of the resource system at runtime if the exact resource **UID** is not known, for instance if it is necessary to resolve a run-time dependent, operating-system specific file.

The torrent registry database does not store the actual resource data. It is designed to be lightweight, optimised for handling simultaneous torrent lookup or tag query requests from multiple resource system clients. Resource system client instances may query a torrent look-up Web service for:

- A **UID**, resolving to torrent data for use with Azureus.
- A set of key-value annotation pairs, resolving to a set of matching **UIDs**.

The torrent registry database is essential to the resource system. Its point-to-point Web service communication with worker nodes provides scalability, while its integration with the Microbase notifica-

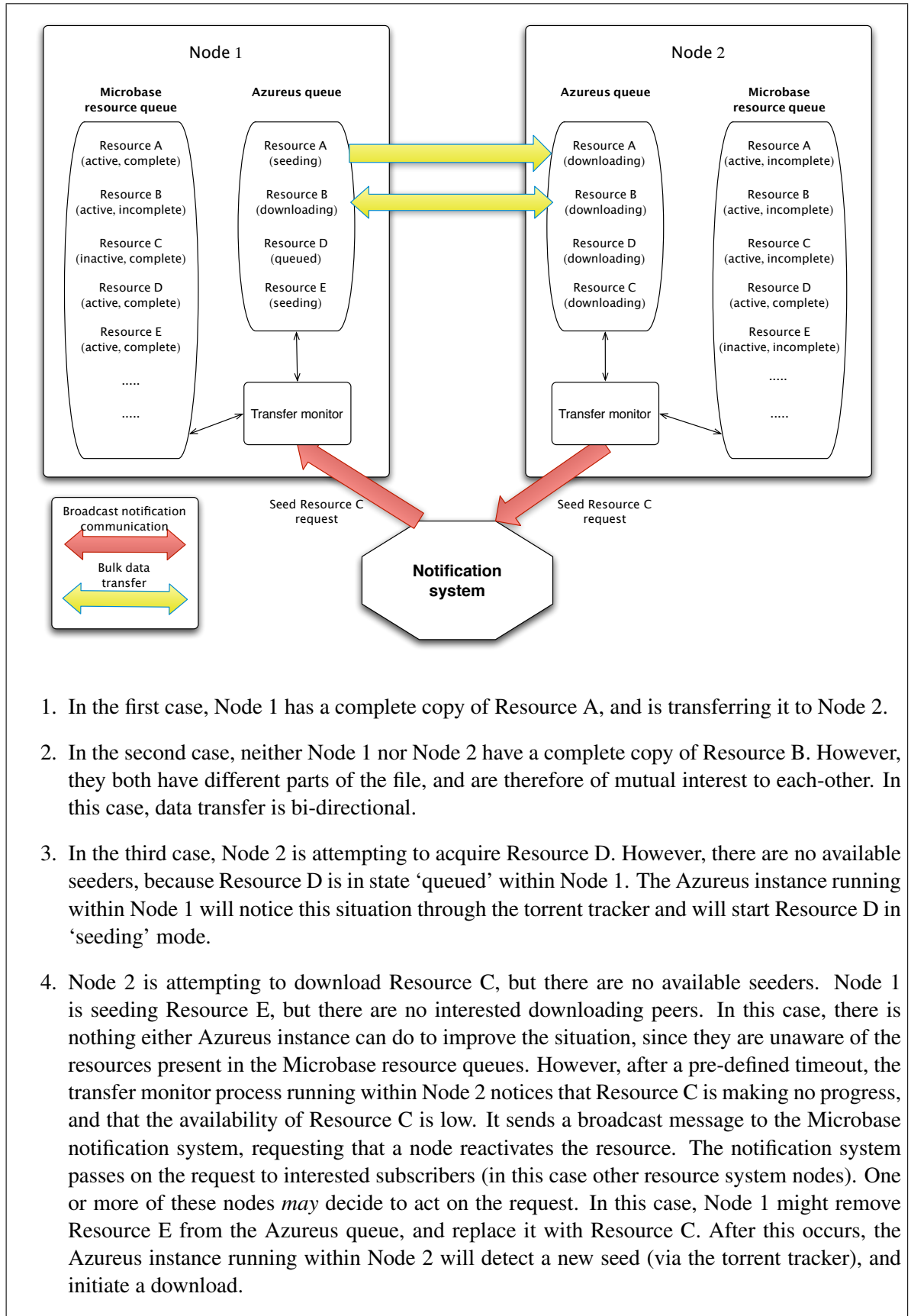


Figure 5.4: Shows how the two queuing systems interact to perform resource sharing logistics. Two participants in a Microbase resource system are shown. In this simplistic example, each Azureus queue is allowed to have four items in total.

tion system allows other responders to be informed of newly published files. In addition to providing a functional look-up service, it facilitates the resource system’s wider obligations to provide a provenance record for each file stored in the resource system.

5.4.4 Resource archiving

A resource archiver node is a server-based component that is responsible for the permanent archival of a set of resources produced by other responders present in a Microbase system. The archiver node architecture is shown in Figure 5.5. A resource archiver is actually an instance of a resource system client exposed as a responder. Resource archiver nodes are registered to receive requests from the notification system about the availability of newly published resources. Archiver nodes are responsible for downloading and maintaining a permanent copy of every resource file produced by every job execution. When a download is completed successfully or fails, a notification is sent to inform other system components of the success or failure. Resource archiver services should be installed on ‘reliable’ server machines with large amounts of disk capacity. More than one instance of a resource archiver may be present within a Microbase installation for load-balancing or redundancy purposes.

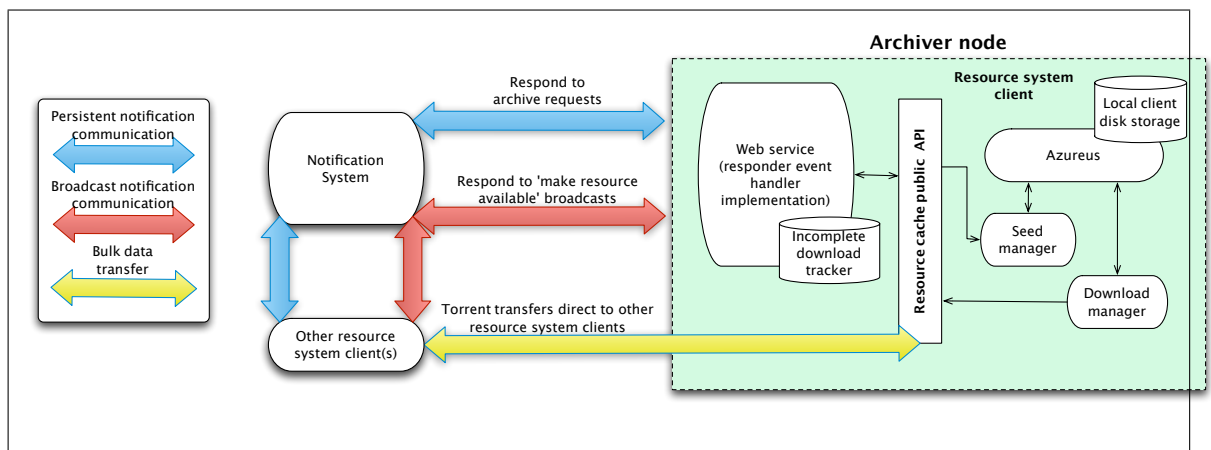


Figure 5.5: The architecture of an archive node consists of an instance of the resource system client wrapped inside a Web service. The Web service is actually a Microbase core responder that responds to requests for resource archival. It also responds to requests from other nodes to activate specified torrents.

5.4.5 Downloading a resource

There are two ways that a node may request a resource: either directly by its unique resource UID (if known), or by querying for a set of key/value tag pairs. Unique identifiers of resources matching

the key/value pairs will be returned. The downloading node can then choose which of the matching resources it wishes to acquire. If a downloading node notices that there are no seeds, or limited availability of a torrent, the broadcast functionality of the notification system is used. As described in Section 4.4 on page 76, the broadcast functionality provides a non-reliable, but low latency message delivery service. Unlike ‘ordinary’ notification messages, ‘broadcast’ messages are not stored permanently, and are therefore an ideal low-overhead mechanism for components wishing to message each-other in a loosely-coupled, anonymous, fashion for synchronisation purposes. In this case, the node downloading the file broadcasts a ‘please (re-)seed torrent X’ message. If a resource archiver node, or another worker node receives this message, it may decide to start seeding the required torrent by using an available upload slot. The node requiring the file may then proceed to download it. The details of this process are shown in figure 5.6 and are entirely hidden from the client application, except for the additional time taken to find suitable peers.

5.4.6 Publishing a resource

Resource publication requests are initiated from the user application (see Figure 5.7 on page 104). The user application provides a file containing the content to be published, a **UID** to identify the file content, and a set of key/value annotations for query purposes. The publication process itself is asynchronous, allowing the client to continue processing while the resource system archives the requested file. On receipt of a publication request, the resource system client generates the appropriate `.torrent` file locally on the worker node. The `torrent` data is simultaneously transmitted to the resource system torrent look-up Web service, and exposed via the locally-running Azureus instance. Following a notification event indicating that a new file should be archived, one or more archiver nodes are assigned to the backup operation. If the resulting BitTorrent transfer is a success, then the archive operation is complete, and the resource can be considered ‘safe’ for use in future operations. In this case, a final “resource archive successful” notification message is sent, which may be used as confirmation that the file has indeed been permanently archived.

Two failure modes are also considered: temporary and permanent. Temporary unavailability of a BitTorrent share may be the result of queue management operations described in Section 5.4.2.1 on page 97. In this case, appropriate broadcast notification messages are sent from the archiver node requesting that the torrent file be re-activated. Permanent failure occurs when an archiver node reaches a configured number of retry attempts or a time-out value is reached. In this case, the archiver node assumes that the file will never be available and so will never be archived correctly. This

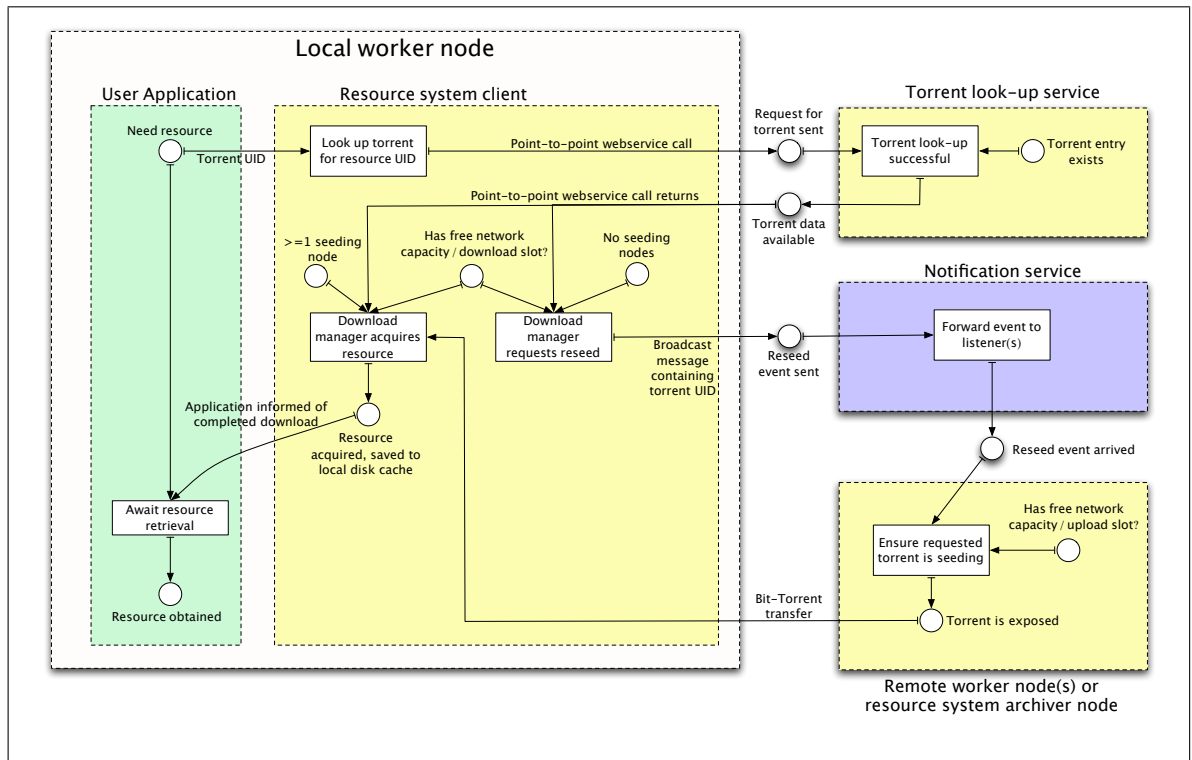


Figure 5.6: A request for a download from a user application involves several other system components. A request for a file with a specified **UID** is made from an application (green). The resource system client then attempts to resolve torrent data for the specified file by querying the torrent look-up service. Assuming the **UID** is recognised, the torrent data is returned from the Web service (data size 1kB-150kB). The torrent is then added to the locally running download manager, within the resource system client. Assuming at least one other Azureus node has a copy of the file, then the file is transferred and the user application is informed of the completed download. However, if no other nodes are currently seeding the required resource, then a request is sent to the notification system (blue). This request is forwarded to interested nodes: archive nodes, or worker nodes. If another node subsequently makes the required file available (a seed), then the local resource system client can then proceed to download it.

situation may occur if a worker node is removed from the Microbase system due to user interruptions, reboot, or other failure before the file can be successfully copied. If a permanent failure occurs, an appropriate notification message is sent from the worker node to instruct interested subscribers. At a minimum, this results in the `torrent` entry being deleted from the torrent look-up database.

As with the resource acquisition process (described in the previous section), the BitTorrent implementation details are masked from the user application.

5.5 Discussion

A resource distribution system has been developed to meet the scalability, reliability and extensibility demands of the Microbase system. Distributed collections of worker nodes are typically better

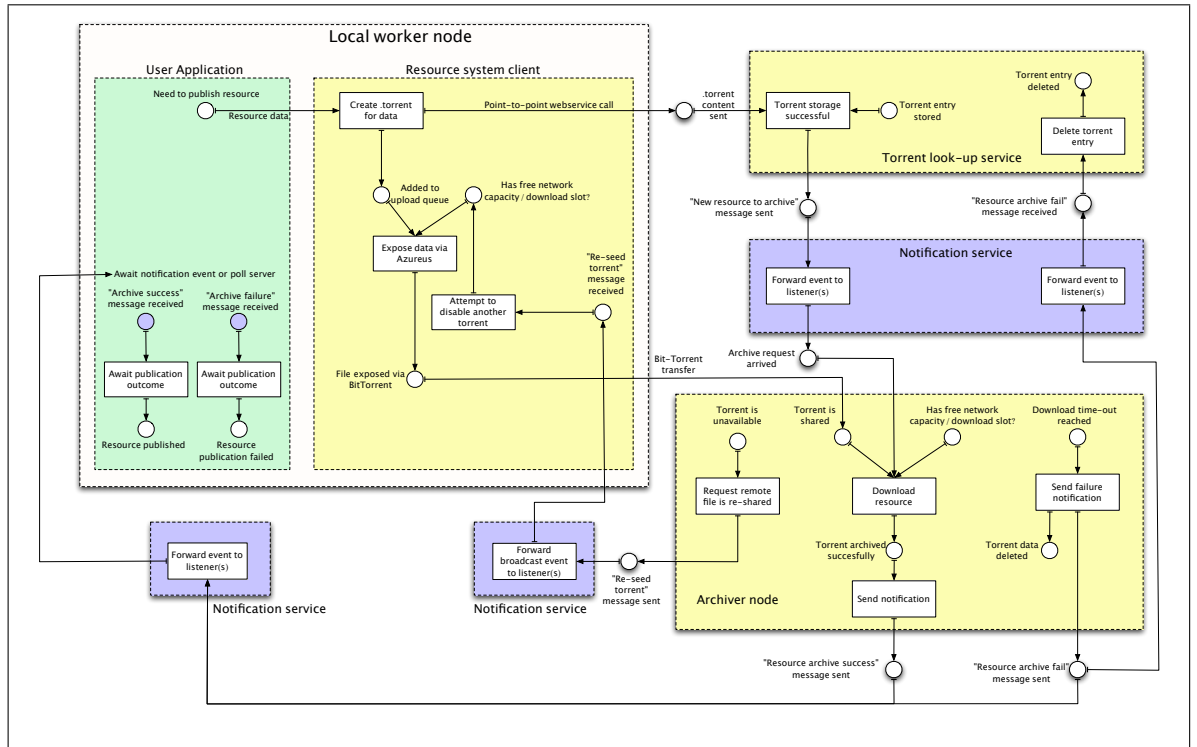


Figure 5.7: Shows various possible operations and state changes in different subsystems as an application publishes a data file to the resource system. User applications running within worker node(s) (green), call a ‘publish’ method on the locally-running portion of the resource system — the client library (yellow, within boxed area). A Web service call is made to the resource system’s torrent look-up service in order to register lightweight `torrent` data and its associated annotations. This triggers a message to be routed via the notification system (purple), arriving at a resource system archiver node (yellow, bottom). The archiver node attempts to perform the bulk transfer operation via BitTorrent, directly from the worker node. The eventual outcome of this transfer operation is published as another notification message, allowing listeners to perform the appropriate operations.

suitable to course-grained application-level parallelism, rather than finer-grained, distributed thread-based parallelism (see background chapter Section 2.2.1 on page 18). Exploiting application-level parallelism allows each instance of an application to execute independently on its own worker node, with little or no IPC required between worker nodes. Importantly for Microbase, this form of parallelism does not require modification of existing applications (see Section 3.3.5), provided that a suitable data staging and execution environment exists. Specifically, of particular relevance to the Microbase resource system, application-level parallelism typically requires large blocks of input data to be staged prior to application execution, and blocks of result data to be retrieved post-execution. This is in contrast to thread-level parallelism, where multiple threads may be working on the same distributed shared memory model of, for instance, a matrix, requiring near-constant communication of small sections of that matrix. P2P data transfer methods are well suited to the former case, and the Microbase resource system has been designed with this in mind in order to efficiently distribute data in the form of files to a large number of nodes.

By sitting as a layer between a domain application, the computer hardware, and the job scheduling component of a responder, the resource system can contribute significantly to the high-level Microbase system requirements. The resource system facilitates the de-coupling of job scheduling operations from the execution of jobs on specific worker nodes. The ability to annotate resource files enables worker nodes to perform runtime look-up operations in order to determine appropriate platform-specific binary files. Therefore, scheduled jobs may be run on any available operating system/architecture distribution that happens to be available at a particular point in time, provided that suitable native-executable files are available for the platform(s). This helps to meet system-wide environment requirements (see Section 3.3.1 and 3.3.5). The use of the BitTorrent protocol facilitates large-scale data and software distribution by permitting worker nodes to work co-operatively. Each node contributes network bandwidth and temporary disk space to the distribution system (see Section 3.3.2).

The use of a worker node's local disk space as temporary storage of result data is advantageous to system-wide efficiency and scalability for a couple of reasons. Firstly, the ability to cache result files on worker nodes allows nodes to process new work while concurrently uploading the result files from a previous job execution. Secondly, output files from one program may need to be used as input data for another program. If the second program happens to execute on the same worker node, then the required input data will still be available locally, and so no data transfer operation will be required. If the second program executes on a different worker node, then files will need to be transferred to that node. However, file transfers can occur directly between the two nodes via BitTorrent, requiring minimal server overhead (Section 3.3.2).

The server-resident portion of the resource system, responsible for permanent archival of data, is trivially extensible and scalable both in terms of disk capacity and in terms of load balancing worker node requests. Larger numbers of worker nodes can be accommodated by simply installing additional archiver node instances, which can be done at runtime by starting another server application container. File transfer protocols such as FTP can make available an unlimited number of files simply by exposing a shared directory. BitTorrent implementations cannot support limitless numbers of 'shared' files simultaneously because each shared file consumes system resources. However, by combining the BitTorrent implementation embedded in the resource system with the notification system, de-coupled co-operation between nodes is possible. Messages can be sent requesting that particular files are exposed via BitTorrent, making it possible to share as many files as there are disk space, albeit not simultaneously and with a messaging delay overhead. Therefore, the Microbase system-level extensibility requirements have been fulfilled (see Sections 3.3.2 and 3.3.4).

The resource system provides a facility for permanently archiving file-based resources. Stored resources can be queried by name, version, or any other combination of implementation-defined annotations. Each version of a file is also uniquely identifiable. While this functionality alone is insufficient to enable arbitrarily-extensible pipelines, it is necessary. When taken together with the extensible notification system (see Chapter 4) and the way in which the job management system (Chapter 7) works, the resource system provides an essential repository of software and previously-completed data files required as ‘hooks’ to which future responders could be attached. Additionally, file versioning permits hot-patching of user job implementations, including third-party native executable files since, by default, the job management system selects the latest version of a given file. Meanwhile, the archival of old software versions means that it is possible to re-run older software if necessary, for example, to compare program outputs. The resource system described in this section therefore addresses the scalable distribution requirement (see Section 3.3.2 on page 53), and assists with system wide extensibility and maintenance requirements (Section 3.3.4).

Regarding the raw data transport layer, the Microbase resource system compares favourably with the data transfer protocols used by other desktop Grid systems, including support for client-side data caching and server striping. The use of BitTorrent permits data transfers via Wide Area Networks (WANs), such as Internet-connected nodes. Although the Microbase resource system does not scavenge desktop storage space permanently as other systems such as FreeLoader [308] do, the resource system does temporarily cache large amounts of data locally, facilitating scalability in the presence of bursts of high-demand for particular files.

Importantly for responder developers, no knowledge is required of the underlying BitTorrent transport mechanism. Data publication and retrieval operations are exposed via a public API and deal in terms of well-known, higher-level concepts such as files, unique identifiers and annotation metadata. Handling of torrent files, file availability co-ordination and system resource management of active files is handled by the underlying implementation and is not exposed to the application developer (see Section 3.3.6.1).

Chapter 6

Responders

6.1 Introduction

The Microbase core components (notification system, resource system and job management system) form a generic Grid infrastructure in which applications can operate. Microbase provides the means to perform distributed computation for the end-user, enabling the execution of domain-specific analysis implementations for end users.

Domain applications need to be able to operate within the Microbase environment. This involves one of the following: writing applications that make use of Microbase services explicitly; writing a Microbase-compatible wrapper around an existing application; or modifying third-party application programs to be “Microbase-aware”. This chapter discusses the first two options only. The modification or recompilation of third-party software is not within the current scope. Such modifications are likely to be application-specific and in any case are not always possible, for instance where the source code of an application is not available, or is sufficiently complex to prohibit modification. Therefore, this chapter discusses encapsulation of existing applications within a Microbase-aware wrapper that is responsible for interfacing an application with the rest of the system.

Chapter 3 introduced responders as modular components through which a Microbase system can be extended arbitrarily. This chapter describes the composition of a responder in more detail and also discusses a framework, and a design pattern that has been developed in order to simplify the development process for responder software developers. Previous works have emphasised the need for Grid application programmers to structure their software into appropriate modules providing different types of functionality, in order to allow the best use of available hardware and to assist runtime program mobility [21]. Other works have highlighted the need for distributed application

development to be straightforward and convenient for application developers, while being conducive for automated deployment in environments where administrative restrictions may be in place [116]. The framework described in this chapter aims to provide the necessary abstractions and support infrastructure to provide a convenient means for developers to write Microbase Grid applications in this way.

The term ‘responder’ arises from the event-driven nature of a Microbase system; a responder will typically remain idle until triggered by a message received from the notification system. A responder is a loose collection of modules that when taken together, encompass the entire scope of a domain application’s existence within a Microbase system. This scope includes: the command-line application itself, which may include one or more identical instances on worker nodes; an application-Microbase bridge component; and server side modules such as Web service query interfaces and associated structured databases. The term ‘responder’ refers only to the collection of modules defined by the developer-imposed functional relation between the modules; it does not specify the physical deployment of responders, such as the location or number of instances of each sub-component that are deployed to a system. The responder as a whole can be installed into an existing Microbase system by installing each individual module to its appropriate location.

6.2 Motivation

6.2.1 Bridging Microbase and domain applications

For many typical bioinformatics command line applications, the idea of running anywhere other than a single machine is an alien concept. There is therefore a need to bridge the conceptual and practical differences between an application’s ‘world view’ of running on a single machine, and the reality of a highly parallel and dynamic distributed environment such as Microbase.

When operating on a single machine, or dedicated computer clusters, domain applications are generally installed once and used repeatedly as required. In the Microbase environment, there is a system-level requirement (see Sections 3.3.1 and 3.3.5) for applications to be installed to worker nodes on-demand, and then removed when a worker node is re-claimed by a higher-priority task, such as a user log-in. Per-node installations may have to be performed multiple times over the course of a task enactment due to the volatile nature of the available worker nodes. There is also a system-level requirement to support multiple operating systems and processor architectures, where this is feasible. There is therefore a need for responders to expose domain applications in a form suitable

for cross-platform, automated deployments and executions. In addition to cross-platform deployment, responders also need to facilitate platform-agnostic execution of applications. Executing the same domain application on different operating system platforms may necessitate slightly different command line strings. For instance, an obvious example is the different executable file names used on different platforms: `.exe` or `.bat` on Windows vs `.sh` or the executable file permission bit on UNIX. In addition to platform-specific binary files, differences in program paths and names leads to a requirement to encapsulate domain applications into convenient cross-platform packages. Previous work has highlighted the need for completely automated installations of software in distributed computing environments [271]. If packaged software were capable of being deployed to worker nodes via the Microbase resource system, then this would also satisfy both the requirement for a responder to function in a heterogeneous environment (see Section 3.3.1), and also the requirement for applications to be conveniently repeatedly installed (see Section 3.3.1 on page 52). If these packages were also distributed by the Microbase resource system, then the scalability requirements for distributing applications repeatedly to many worker nodes would also be met (Section 3.3.2).

There is also a system level requirement to handle job execution failures gracefully (see Sections 3.3.1 and 3.3.4). One way in which transient job execution failures can be dealt with is to migrate them and their associated executable and data resources to a different worker node for another execution attempt at some point in the future. This further demonstrates the need for easily-migrated, cross-platform executable packages of domain applications, since there is no guarantee that a future execution attempt will occur using a worker node with the same operating system or processor architecture as previous execution attempt(s).

In addition to deploying executable applications to remote worker nodes, run-time data transfers also need to be handled. Most command line applications will expect their data input(s) to come from arguments specified as part of the command line, or files on the filesystem local to the worker node. Worker nodes cannot be expected to have the necessary files pre-loaded locally, and cannot be relied upon to store output data files for more than a few hours. Therefore, data input files will need to be transferred from a centralised, permanent file store to the worker nodes prior to execution. While inter-node data transfers are handled by the resource system, the responder must specify (in a cross-platform manner) which input files should be copied to a worker node in order to process a particular job. Additionally, many programs will create new files during their execution. Some of these files may be useful result files, while others are temporary intermediate files to be discarded. The responder must therefore specify which output files should be copied from the worker node to permanent storage on completion of an execution.

6.2.2 Responder pipelining, extendibility and developer convenience

Another Microbase system requirement is the ability to form automated pipelines of tasks (see Sections 3.3.2 and 3.3.5). The required responder functionality discussed in the previous section is not, by itself, sufficient to support this system level requirement. There are two issues to be overcome: automated command line generation and pipelining.

Executing a command line application on a single machine involves the operator locating the correct input files, determining appropriate values for command line arguments, and finally, invoking the command. If the same program must be run with different sets of data, or the same data with different parameters, then multiple command lines will need to be invoked. This process is fairly straightforward since the researcher should be able to provide the appropriate values. Automating this process is more difficult. If the computational work to be completed is known ahead of time, then a suitable batch script or equivalent Condor-submit file could be constructed to execute each command line in turn, either sequentially or in parallel on a set of worker nodes. However, in order to satisfy Microbase system requirement (see Section 3.3.5), computational work must be determined dynamically at run-time, with no human intervention. The responder therefore needs to be able to receive event messages from the notification system, interpret these messages, and translate them into computational work units (command lines) for distribution to worker nodes.

There is a system-level requirement to allow organising multiple domain applications into a structured workflow (see Section 3.3.5). In order for this requirement to be fulfilled, it must be possible to co-ordinate these applications, and to permit data flows between them. In Microbase, the notification system (Chapter 4) provides the necessary high-level messaging functionality required for responder co-ordination, while the resource system (Chapter 5) allows scalable data transfers between different processing stages of a pipeline. The responsibility of the responder framework discussed in this chapter is to make this functionality conveniently accessible to responder and pipeline developers.

6.3 Requirements

The responder development framework has been designed to accommodate the overall system-level requirements as described in Chapter 3:

- Facilitate scalable application deployment through domain-specific, modular, extensible components.

- Extensibility and maintainability: each responder should be self-contained, and be able to co-exist with other responders. A particular responder should not have unintended side-effects on other responders present in the system.
- Extensibility: responders must use the Microbase resource system to store result file outputs. Some or all of these files may or may not be required for the pipeline originally intended to accommodate a given responder. However, future responders may require the existence of these files. In order for future responders to operate correctly, and to avoid repeating work, such files should be stored in the resource system rather than be discarded.
- Provide cross-platform computational abilities: it should be possible to write responder implementations in an interpreted language such as Java, or have the capability of linking to platform-native executable programs for multiple architectures.
- Development convenience: The responder framework must provide a straightforward API which to wrap existing applications for use with Microbase. It must also be straightforward to deploy responders and their wrapped applications.

In order to meet the above system-level requirements, the responder developer framework must provide application developers with the following capabilities:

1. Permitting responder developers to create cross-platform, distributed applications as a series of Java Plain Ordinary Java Objects (POJOs). The compute-intensive work will be performed either within one of these POJOs, or the POJO will act as a thin wrapper around a third-party command line application.
2. Access to bulk data transfer capabilities of the resource system and the messaging capabilities of the notification system.
3. Insulation from having to interact directly with Microbase Grid Web service APIs (notification system and resource system) as discussed in previous chapters.
4. Enabling domain-specific platform-native applications to be encapsulated within packages capable of being efficiently transferred to multiple worker nodes via the Microbase resource system.
5. Responder implementations must be able to specify their required input and output resources, so that Microbase can provide the appropriate operating environment on worker nodes.

In order to integrate effectively with other responders, the framework must provide the facilities to:

1. React to external events from other responders present within a Microbase installation.
2. Generating new events for publication to the notification system in order to report completed work or error conditions.
3. Respect the notification system requirement for reasonably-sized messages and timely message receipt acknowledgements (see Section 4.4.1 on page 76).
4. Provide appropriate means for other responders to interpret generated notification messages. For example, via provision of suitable message parsers.

It is unreasonable to expect a system administrator to unravel inter-dependencies between related groups of responders. In order to meet system administration user requirements (see Section 3.3.6.2), the following should be provided:

1. Self-registration of responders, including registration with the Microbase configuration service and notification system. Appropriate notification topics and subscription entries will need to be created.
2. Convenient installation of each responder module — some responder modules will need to be deployed to Tomcat servers, others to the Microbase resource system.

6.3.1 Responder structure

The requirements for a Grid application development framework were discussed in the previous section. This section describes the development of a framework intended to allow easier construction of Grid-aware applications. Although writing applications that use Microbase Web services directly is certainly possible, it is challenging for a number of reasons. There are a several different core services, which may be distributed over several physical servers. For many cases, in-depth knowledge of Microbase service APIs is not required, if a suitable insulation layer is provided. Developer assistance is provided in the form of a software design pattern, which if followed, facilitates the following:

- Abstraction over the notification system. There is only a need to handle domain-specific message content, rather than responder-based handling of message queues, error recovery and so on.

- Simplification of data transfers from the resource system to job implementations executing on worker nodes.
- Computationally intensive jobs may be implemented as a Java ‘bean’¹, where bean properties specify data inputs and outputs.
- Enforcement of a clean separation between server-resident and worker-node resident portions of the responder.
- Automatic detection of executable resources for publication to resource system.
- Assisted deployment to remote servers.
- Automated registration of a deployed responder with Microbase.

A responder wraps an entire unit of domain-specific functionality (see Figure 6.1). This typically comprises:

1. a server-side component for responding to external event notification messages from the Microbase system
2. compute job component(s) that perform the CPU-intensive operations. Multiple instances of these components will be run in a distributed fashion.

The architectural split is necessary in order to fulfill system-level requirements and to make the best usage of the capabilities provided by the underlying technologies. For instance, a Web service-based event handler is well-suited to receiving events from external components via the notification system as a push subscriber. Having an ‘event handler’ module that implements the push subscriber interface satisfies requirement to react to new data (see Section 3.3.5), allowing a responder to react to external stimuli. Additionally, if the responder requires structured storage, or Web service query interfaces, a server-based component is a logical and convenient place to put this functionality. The server-resident portion of a responder is expected to remain idle until an incoming message is received, at which point, it may schedule computationally-intensive work to be performed. Event handler modules are responsible for:

- Announcing their presence and current Web service endpoint to a Microbase system. This involves registration with the notification system in order to receive appropriate notifications

¹<http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp>, accessed 2009/05/13

events. These announcements also permit event handler mobility, for instance, in the case where excess server load or hardware failure requires the redeployment of a responder.

- Responding to relevant notifications events in order to schedule computationally-intensive work.
- Deciding how to split a large computational task into multiple smaller units of work, each suitable for execution on individual worker nodes.
- Publishing new notification events to inform other responders of the success or failure of work items.
- Event handler modules should only perform lightweight operations, since they execute within a server environment with limited processing resources

Meanwhile, a computationally-intensive job implementation can be placed in a separate, de-coupled module. This module can be uploaded to the Microbase resource system and distributed to worker nodes. Separating the computationally-intensive parts of an application from the organisational parts (i.e., the event handler) ensures that system scalability requirements (see Section 3.3.2) are satisfied by enabling BitTorrent distribution of program files to temporary worker nodes, whilst also providing a ‘well known’ location for the routing of notification events. Environmental requirements (see Section 3.3.1) are also met, since it would be possible to provide different executable modules for different pre-compiled architectures. Job implementation modules have the following properties:

- Perform the vast majority of the ‘heavy-lifting’ computational work required by an application.
- Have a cross-platform or multi-platform implementation, where appropriate.
- Able to be deployed efficiently to large numbers of worker nodes.
- In case of worker node failure, should not perform any work that cannot be repeated elsewhere.

6.4 Developer support for responders in Microbase

Implementing any complex software component, such as a Microbase responder can be a fairly involved process, particularly given the range of technologies used. Software development can be made more straightforward in several ways, including:

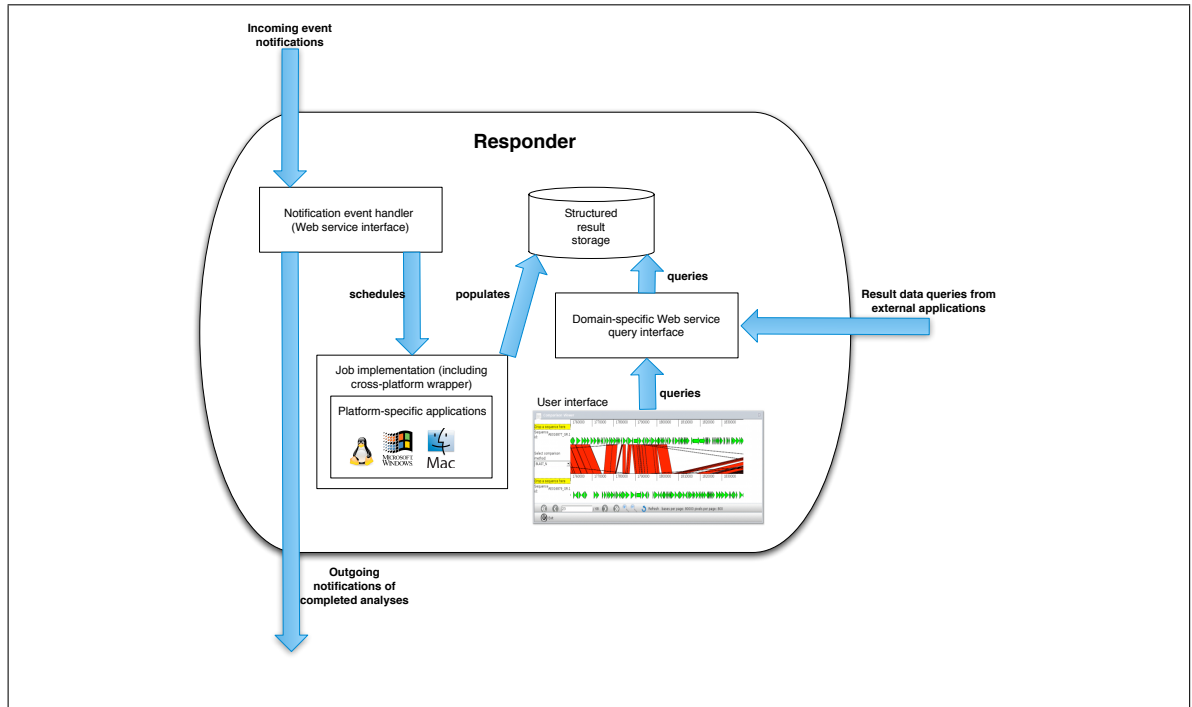


Figure 6.1: Responder architecture: A responder is a self-contained collection of modules providing a broad range of domain-specific functionality. Different parts of the responder are deployed to different types of computer hardware. For instance, databases are deployed permanently to reliable server-grade hardware, whereas computationally-intensive job instances are deployed on-demand to one or more worker nodes. Job implementations for multiple platforms may be provided.

- Providing library support that implements commonly-used features.
- Providing design ‘templates’ of individual program files, or even entire projects. This allows the developer to start with a basic skeletal structure of a file or project and incrementally add features as the need arises.
- Providing project layout recommendations provide familiarity across multiple projects. For example, if implementation files are be laid out the same way in each project, program code follows similar implementation patterns, and similar operations are performed in the same types of module then moving from one project to another becomes easier.

The various responsibilities for providing the appropriate environment for executing domain applications in a distributed environment are split between the Microbase core services and the responder implementation. For instance, file transfers and archiving, management of Structured Query Language (SQL) databases, and data flow and co-ordination operations all require part-involvement of Microbase core services and part-involvement of responder modules. Of the responder implementation’s responsibilities, there is sometimes a choice between whether a particular piece of functionality is better implemented in the server-based event handler module, or the worker node-based job

implementation module, or split between the two. It is the responsibility of the responder developer to decide sensibly where (i.e., which responder module) to implement a particular piece of functionality. While developing the Automated Genome Analyser (AGA) pipeline (see Chapter 8), we discovered a pattern of commonly-needed project structures or functionalities that have since been incorporated into the responder development framework described here.

This section describes implementation possibilities of commonly-required functionality, and suggests how a typical responder implementation might be achieved. Where generic library support for a particular implementation task has been provided as part of the responder development framework, this is pointed out. Finally, a Maven [114] project layout for the implementation of Microbase responders is presented. The advantages of using Maven archetypes for software development are discussed in Section 6.5. If the suggested Maven archetypes are used, the result will be a responder that has strict modularity, can benefit from semi-automated installation into a Microbase pipeline, and whose design structure will be familiar to other responder developers, promoting code re-use.

The intention of the responder development framework described here is to provide a straightforward self-contained development structure that minimises exposure to Microbase core service APIs wherever possible. For example, `abstract` classes are provided for event handler and job implementation modules. These classes provide much of the commonly-required functionality and require only that the developer add or override specific methods in order to provide domain-specific functionality.

A complete tutorial on how to write a responder is beyond the scope of this chapter. An example of a simple, but fully complete and working responder is provided in Appendix A. Complete listings of example event handler and job implementation responder modules can be found at the Microbase project Subversion repository: <http://microbase.svn.sourceforge.net/viewvc/microbase/trunk/microbase-examples/?pathrev=284>.

6.4.1 Responder initialisation

The initialisation process of a responder involves the following actions:

- Registration of the Web service endpoint with the Microbase system.
- Registration of the responder with the notification system so that the responder is permitted to send and receive notification messages.

After registration, if the event handler Web service is subsequently moved to a different physical server, the host name part of its endpoint will change. In order to continue receiving event notifications, the notification service would need to be informed of this endpoint change. By ensuring that service registration occurs at every service start, the Web service can be migrated between different servers without any need for reconfiguration. This behaviour satisfies maintainance and flexibility requirements .

A minimal event handler implementation might resemble the following fragment:

```
public class MyEventListener
    extends AbstractEventResponder
{
    public MyEventListener()
        throws ConfigurationException
    {
        setIncomingTopicIds( ... list of topic IDs interesting to this responder ... );
        setOutgoingTopicIds( ... list of topic IDs published by this responder ... );
    }

    @Override
    protected void responderInitialisation()
        throws ConfigurationException
    {
        // ... responder-specific configuration ...
    }

    @Override
    protected void dealBroadcastMessage(BroadcastMessage messageItem)
    {
        // ... handle broadcast message here ...
    }

    @Override
    protected void dealMessage(MessageLogItem messageItem)
        throws UnrecoverableException, TransientException
    {
        // ... handle message here ...
    }
}
```

Behind the relatively simple event handler implementation, a complex set of operations must be performed to fully initialise and register the responder with a Microbase system (see Figure 6.2). These operations are performed entirely by the responder support library.

The Web service endpoint of the event handler is determined automatically, and registered automatically. Notice that there are no references to the notification system Web service client at all. Instead,

the `constructor` contains two method calls that inform the support library of the `UIDs` of the notification topics that are of interest to this responder implementation. If processing of the `constructor` completes successfully, then the responder can assume that it has been successfully registered as a `publisher` and a `push subscriber` with the notification system. Required message topics will also be created, if they did not already exist.

After successfully registering with the notification system, the `responderInitialisation()` method allows responder-specific initialisation to take place. The content of this method are entirely responder-specific, but would be a suitable place to connect to database pools or perform consistency checks before any message processing begins.

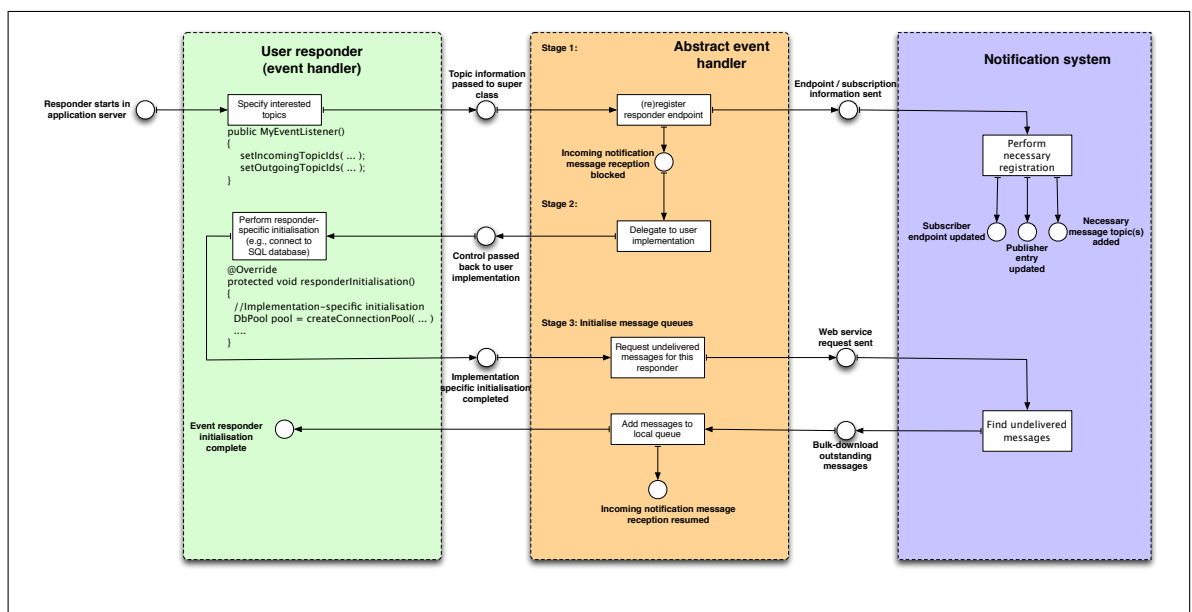


Figure 6.2: Shows the background tasks undertaken by the responder support libraries when a responder event handler starts inside an application container.

Stage 1 involves blocking incoming messages while the responder is still in an initialisation state. The current Web service endpoint is determined, and (re)registered with the notification system. Required message topics are also created, if this is the first time the responder has been started.

Stage 2 allows the responder to perform implementation-specific initialisation operations. This might include contacting external services, setting up database connection pools, and so on.

Stage 3 involves the bulk-download of existing notification messages that were perhaps sent while the event handler was off-line. Finally, incoming event notification is re-enabled, allowing the event handler to respond to new messages.

6.4.2 Handling notification events

On successful completion of the responder-specific initialisation section, the event handler becomes idle until there is a message for it to handle. When a new message arrives, either `dealMessage()` or `dealBroadcastMessage()` are called, depending on whether the message is a 'normal' message or a 'broadcast' message (see Chapter 4). Each of these methods are passed an object that includes

the content of the message, including meta-data listing publication time, which responder published the message, and so on. Notice that there is no direct involvement with the notification system. The responder developer only needs to override the appropriate method in order to process incoming event notifications.

Figure 6.3 shows how a typical event handler portion of a responder operates as part of a chain of responders. The operations shown in ‘User Event Handler 1’ are not absolutely required; an event handler may perform any kind of operation as determined by the responder developer. Typically, most event handlers will respond to an incoming event by requesting an amount of computational work to be performed. If there is a ‘large’ amount of computational work, then the event handler is responsible for splitting the required computational work into multiple parts for execution on different worker nodes. It will then wait for the result of the computation, before sending a notification to indicate that an analysis operation has been completed.

Interpreting incoming messages

The first challenge in processing an incoming message is to be able to interpret its meaning. There are two issues for a responder developer to consider: message content and message format. We make the assumption that if a responder registers an interest in a particular topic, then it should at least have an understanding of the content of those messages. For the formatting of the message, the specification of the notification system does not stipulate any formatting guidelines for the message body. The convention we have adopted for internal Microbase messages, as well as messages sent between [AGA](#) responders is to represent message content in the form of standard Java data beans. This approach has the following advantages:

- Provides convenient access to message properties.
- Complex data types (sets or lists of items) can be used, if required.
- Java data beans can be easily serialised into an Extensible Markup Language ([XML](#)) message body through the standard Java serialisation libraries.

The caveat, however, is that a responder wishing to parse a message sent by another responder must have access to the appropriate message data bean in order to de-serialise the message content. Since different responders are separate, modular projects, this access is not provided by default. A solution to this issue is discussed later.

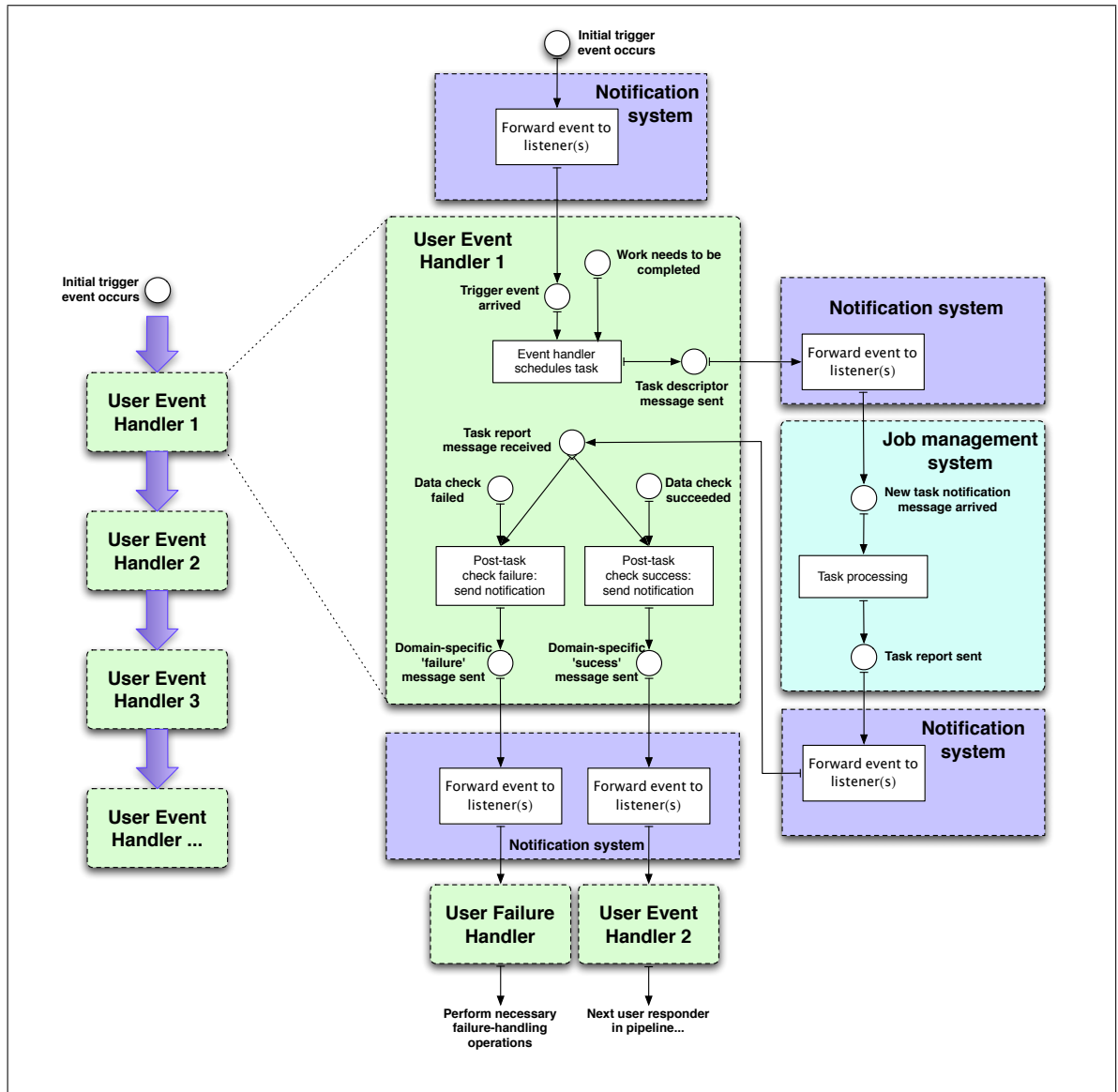


Figure 6.3: Green shaded boxes are user responders, purple boxes represent actions performed by the Microbase notification system. Actions performed by the task management system are represented by the cyan-shaded region.

A pipeline of responders is shown on the left of the diagram. The first responder has been expanded to show the types of operations performed by a typical event handler:

- 1) A notification message is received.
- 2) If this event requires some computationally-intensive work to be performed, then the type of work needs to be determined.
- 3) A notification event is published by the user responder, containing details of the work to be performed.
- 4) This message is forwarded by the notification system to the job management system.
- 5) The job management system uses available worker nodes to complete the work. On completion, a task report is published, and forwarded via the notification system, back to the user responder.
- 6) The event handler should inspect the task report for successful or failed jobs. The event handler should use application-specific knowledge to determine whether the computation was successful in order to determine what further action to take.
- 7) On successful completion, the next responder in the pipeline is triggered by the publication of a 'success' message, indicating that new data is available.
- 8) On failure, the responder developer can choose what action to take. For example, either the event handler could send a 'failure' message to alert another component to the problem, or it could attempt a corrective action itself.

Sending a message

Sending messages from a responder to the notification system is straightforward. The abstract event listener provides appropriate methods for publishing messages. A responder simply needs to provide a message topic, and a suitably-serialised message body.

Reliability, scalability, failure handling

There are a number of situations that may impede the efficient operation of responders, and the operation of the notification system in general. If notification message delivery is tightly coupled to message processing, then receipt of a message only occurs when successful processing of that message is complete. This behaviour is required so that if message processing fails for some reason, then it can be retried at a later time — when the notification system retries delivery. However, the problem with this approach is that a notification system message delivery thread is occupied the entire time that a message is being processed by a responder.

Queues are well suited to this type of producer-consumer problem by permitting both processes to continue at their own rate; the notification system to wait for slow message processing operations. The responder support library de-couples message delivery from message processing by maintaining a buffer of messages received from the notification system. Each responder has its own unique, independent buffer. Instead of processing a message immediately upon reception — a potentially lengthy operation — the message can be added to the queue, permitting a successful delivery acknowledgement to be sent back to the notification system straight away. This approach allows notification system delivery threads to be freed quickly, allowing messages to other responders to be processed (see Figure 6.4). However, in accepting the message and signalling a successful delivery, the responder must take responsibility for handling errors that might occur during message processing.

There are two types of failure that may occur: permanent and transient. Permanent failures are those where no matter how many times an action is retried, it will always fail. For example, if a data entry is missing or inconsistent, or if a program bug prevents a successful operation. Other failures may be transient, that is, if they are retried at a later time they may succeed. For instance if an event handler must connect to a database or other external services in order to process a message, there is no guarantee that those external services are continuously available. If such external services are unavailable then message processing will fail. However, if the failed external service is repaired, then a subsequent retry will succeed. A message processing attempt will also fail if the server hosting the event handler suffers a failure, such as a power loss or unexpected reboot.

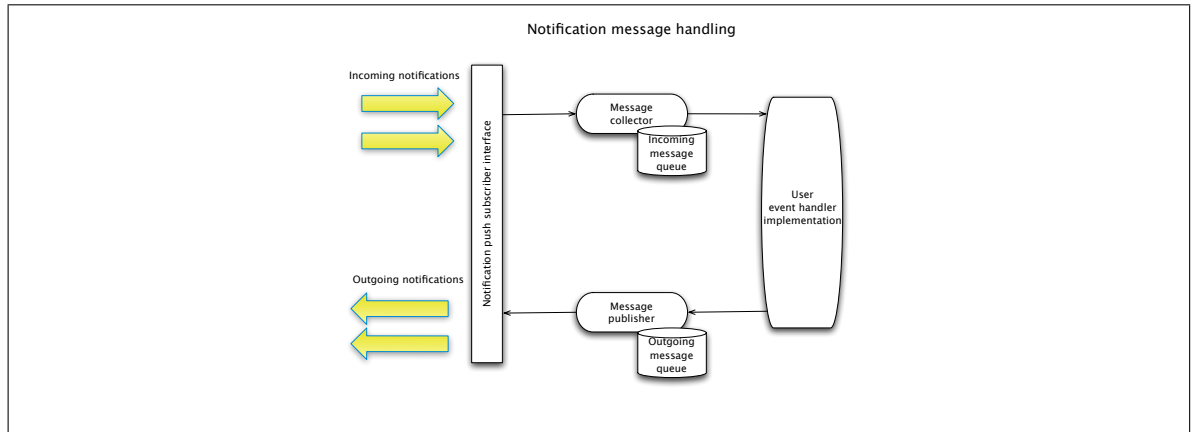


Figure 6.4: Messages received from the notification system are stored persistently in a queue, local to the responder. Message delivery acknowledgement occurs as soon as a message is stored in the queue. This allows the notification system to continue with further deliveries. From the perspective of the notification system, the message has been successfully delivered and is no longer its concern. Meanwhile, queued message items are processed in order, at the fastest rate achievable by the responder. If message processing is successful, the item is removed from the queue. If a fault occurs during message processing, the message item will not be removed from the head of the queue, allowing it to be retried after a delay.

A similar queue is used to temporarily store outgoing message publications from the responder. Instead of sending a message directly to the notification system, outgoing messages are first queued. If the notification system is temporarily unavailable, then the event handler library will handle retry attempts. This approach enables a responder to continue functioning in the absence of the notification system. It also adds reliability by ensuring that outgoing messages are not lost if both the notification system and the responder suffer a crash simultaneously.

Additionally, the notification system itself is not guaranteed to be available continuously. If the notification system is unavailable for a length of time, then it cannot accept messages published by responders during this time. If the responder also suffers a failure while the notification system is unavailable, then messages pending publication may be lost.

To address these issues, the responder support library stores messages in the local queue until they have been successfully processed. A processing failure will result in the message being retried at a later time. Message processing attempts that repeatedly fail due to non-environmental, transient problems are skipped after a retry limit is reached.

6.4.3 Executing command line applications

Job implementations in Microbase can be thought of as an extended Java bean. Microbase functionality is exposed indirectly; the developer does not need to interact directly with data transfer mechanisms or core Microbase functionality. Bean properties are used to hold or reference input and output data. Similar kinds of approach have been used in previous works (see Background Section) as a means of simplifying development of distributed computation systems.

Consider the following Unix command line, which will execute the **BLAST** program over two sequence files to produce an alignment:

```
./bl2seq -p blastn -e 0.00058 -i sequence1.fasta -j sequence2.fasta -o blast_output.txt
```

The command string may be broken into several parts:

- Executable program name: **bl2seq**
- Inline command line arguments: **-p blastn** and **-e 0.00058**
- Command line arguments requesting input files:
-i /genomes/NC_000964.fna and **-j /genomes/NC_002570.fna**
- Command line arguments representing output files: **-o blast_output.txt**

The program name corresponds to the file name and location of the executable on the computer's disk. Some input parameter values are passed to the program in-line — the content embedded within the command line string is a data item that will be used as-is by the application, or parsed into an appropriate data type, such as a floating point number. Input values embedded in a command line are necessarily small data items in order to fit within an operating system's command line buffer space. When large quantities of data need to be accessed by a program, such as a genome sequence, the data is placed into a file instead. In this case, a command line parameter is used with an appropriate 'pointer' to the file containing the required data content. Similarly, command line parameters are used to specify where a program places its outputs.

The distinction between the different parts of a command line is important when executing an application in a distributed environment. Worker nodes will not necessarily have the required input data files, or indeed the executable files, on their file-systems. These files therefore may need to be copied to the worker nodes at run-time. Microbase is responsible for handling these file transfers, but it must be told which files need to be copied; this is the responsibility of the responder developer. Microbase makes a distinction between input parameters and input files because it uses separate transfer mechanisms for each. The Microbase resource system is efficient at transferring large data items, such as sequence files, but there is a prohibitively high overhead when transferring small items of just a few bytes in size. Therefore, input parameters that do not represent file names (i.e. large resources) are transferred as part of the job description sent to worker nodes via a Web service call. Again, the distinction between these is application-dependent, and therefore the responsibility of the responder developer to inform Microbase appropriately.

Dealing with job I/O

Java annotations allow information to be attached to `classes`, `properties`, `methods`, or even other annotations. This information can be introspected at runtime if necessary. Microbase makes extensive use of annotations in order to allow the job implementer to specify which bean properties should be treated as Input/Output (I/O) entries for the job.

There are two annotations provided by the responder framework that can be used to specify the inputs to job implementations:

@InputParameter allows the developer to specify ‘small’ data items to be passed to the Java implementation. Valid java data types are: primitive types such as `int`, `long`, `boolean`, as well as `String` values. Although it may be possible to send ‘large’ (megabytes) Strings as parameters, this is not recommended.

@InputResource allows ‘large’ resources to be sent efficiently to a job implementation via the Microbase resource system. `@InputResource` allows the use of many structured data types to be used within a Java compute job without the need for the implementation to have knowledge of where the resource originated from, or how to marshal and un-marshal objects across a network. Almost any serializable Java type may be specified used with `@InputResource`. Microbase will handle deserialisation of complex types, including: `Maps`, `Sets`, and `Lists`. The Java type `File` may also be specified for data items that will not fit into a worker node’s available RAM, or for objects that need custom (de)serialisation. In this case, the raw file is available to the job implementation. Using `File` is useful when input data is required to be passed to a command line application, rather than for consumption by the Java wrapper.

An annotation is also provided to be used to specify the outputs of job implementations:

@OutputResource allows result items (such as structured Java objects, or files created as a result of executing a command line application) to be ‘collected’ and archived by Microbase. The same data-types supported by `@InputResource` are also supported by `@OutputResource`.

The following example illustrates how these annotations can be used:

```

public class ExampleJob
    extends AbstractJob
{
    private long simpleInput;
    private File inputFile;
    private Map<String, Integer> complexTypeExample;
    private File resultFile;

    public ExampleJob()
    {
        throws ConfigurationException
    }

    @Override
    protected void doWork() throws JobProcessingException
    {
        // Computationally-intensive tasks go here ...
    }

    @InputParameter(inputName="simpleInput")
    public void setSimpleInput(long simpleInput)
    {
        this.simpleInput = simpleInput;
    }

    @InputResource(inputName="complexTypeExample")
    public void setComplexTypeExample(Map<String, Integer> complexTypeExample)
    {
        this.complexTypeExample = complexTypeExample;
    }

    @InputResource(inputName="inputFile")
    public void setInputFile(File inputFile)
    {
        this.inputFile = inputFile;
    }

    @OutputResource(outputName="outputFile")
    public File getResultFile()
    {
        return resultFile;
    }
}

```

The standard bean property ‘get’ and ‘set’ methods are annotated. At run time, the job management system (described fully in Chapter 7) reads the content of these annotated methods in order to set the appropriate values before control is passed to the job. If appropriate annotation values are used, the job implementation can assume that the input values will be downloaded, de-serialised and ‘set’ automatically. Likewise, when the job finishes executing the ‘doWork()’ method, the job management system calls the appropriately-annotated ‘get’ accessor methods in order to retrieve the result objects, exposing them to the resource system for archiving.

Dealing with platform-native executable packages

In order to satisfy the requirement, it must be possible to handle native executables in a cross-platform manner. Non-Java platform-native applications must be downloaded and installed to the worker node in the same way as other input resource files. However, there is an important difference — there is no way to know until run-time which executable package file will be needed. For this reason, native executable packages cannot simply be treated as Java Files in the same way as a input data files, because it is not possible to resolve them directly by their UID. Instead, the responder developer framework provides the class NativeExecutable, which represents a native application package.

In order for a native command line application to work with Microbase, it must be ‘packaged’ appropriately. Essentially, ‘packaging’ entails copying an application’s file structure, verbatim, into a standard zip file, along with a Microbase-specific mappings file. The mapping file is a text file that

maps a developer-assigned name to each executable file path. For example, an application's directory structure might contain the following:

```
|-- bin
|   |-- bl2seq
|   |-- blastall
|   |-- blastclust
|   |-- blastpgp
|   |-- copymat
    ... etc ...
```

Then, the a valid mapping file might be:

```
bl2seq = blast-2.2.18/bin/bl2seq
blastall = blast-2.2.18/bin/blastall
blastclust = blast-2.2.18/bin/blastclust
... and so on ...
```

Application resource files cannot be specified by a resource [UID](#) in a job description, so another means of identification is necessary. Two further annotations have been provided:

PlatformSpecificResource This annotation is used to modify an existing `InputResource` annotation. It specifies that the job input is a platform-specific resource, and allows the name and version of a required software package to be specified.

UseSharedCopy If a worker node has multiple processing cores, it is possible that two or more cores may be running the same application but with different data. The presence of this annotation or absence of this annotation determines whether each job instance uses a shared native executable installation, or whether each instance has its own unique copy of an application.

Program information specified by a `PlatformSpecificResource` annotation, combined with the operating system and processor architecture information provided by the worker node runtime environment are used to query the resource system for a matching file (Figure 6.5). The ability to tag and query files in the resource system is essential for this runtime selection process to function.

6.5 Maven project layout

The conceptual architecture of a component in a typical software system may be distinct from its physical implementation. For instance, particular implementation files can be placed within a directory structure at the choosing of the developer. Maven [114] is a project and build management

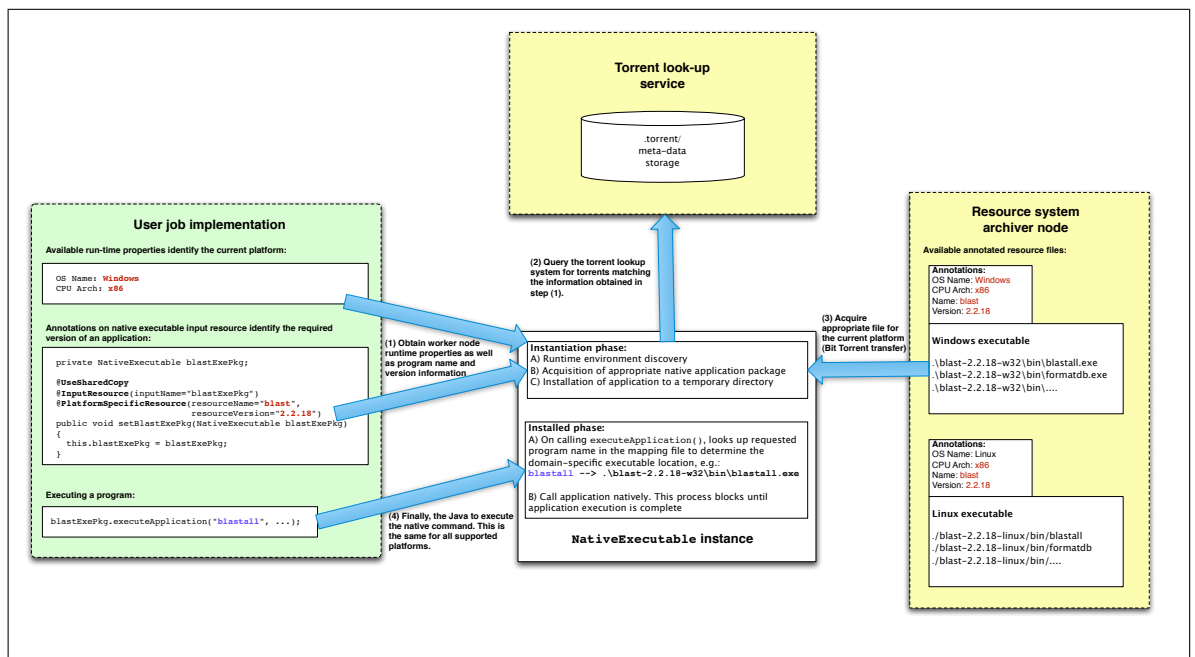


Figure 6.5: Shows the way in which platform-specific applications are supported in the responder development framework. The box on the left hand side shows the required Java code fragments to be written by a responder developer to:

- specify that a particular version of a program is downloaded and installed, and
- how to execute a command line.

When a job implementation is instantiated on a worker node, the Java annotation values are taken in combination with the runtime platform information in order to query the resource system (top) for resource files with matching tags. Assuming that a suitable file is found, it is downloaded via the resource system — either from an archiver node or from another worker node with the same application installed. The downloaded file is then extracted and is ready for use. The command name mapping file (introduced in Section 6.4.3) is used to map a platform neutral name into a platform-specific pathname. In this case: ‘blastall’ \mapsto ‘blastall.exe’.

system designed to facilitate the development of large, complex software projects. Notable Maven features that are essential to large software developments are: uniquely-identifiable projects, dependency management among projects, and project templates - archetypes . Microbase leverages Maven’s “design by pattern” approach [248, 247, 266] to specify a mapping between high-level concepts and implementation details in terms of project layout, through project templates. At first glance, this approach appears to involve an additional learning curve for the application developer, and restricts their free rein over the implementation process, with restrictions on where particular files must be placed. However, the archetype-driven approach provides significant advantages. For instance, it simplifies the development process and encourages modular design by providing well-defined locations for placing implementation files that provide a particular kind of functionality. Once the layout of a single responder is mastered, the layout of every responder will be familiar. Furthermore, this approach facilitates the re-use of responder projects by: providing self-contained projects that can be plugged into another pipelines with minimal modification; the well-known project structure brings some familiarity to ‘foreign’ responder program code written by other developers. Finally, this approach facilitates semi-automated installation of a responder into a Microbase system, saving the developer or system administrator some considerable effort. Correct installation of a responder is a multi-step process, requiring the interactions of several Microbase components.

6.5.1 Responder project layout and interdependencies

The recommended project layout of a responder is shown in Figure 6.6. There are three typical sub-projects:

1. The Web service specification Java Archive ([jar](#)) project should contain all publicly exposed features features. These include: Web service interface and Java client factory, data beans returned by Web service queries, and notification message beans. Other Maven projects and in particular other responders can specify a Maven dependency to the ‘public’ jar so that can gain access to the provided Web services and data.
2. A Web Application Archive ([war](#)) project containing the private implementation of the server-based event handler module of the responder. This project should contain Web service implementations and other ‘private’ entities, such as [SQL](#) database queries. The content of this project is for deployment to an application container. Its endpoint should be registered with Microbase in order to receive event notifications.

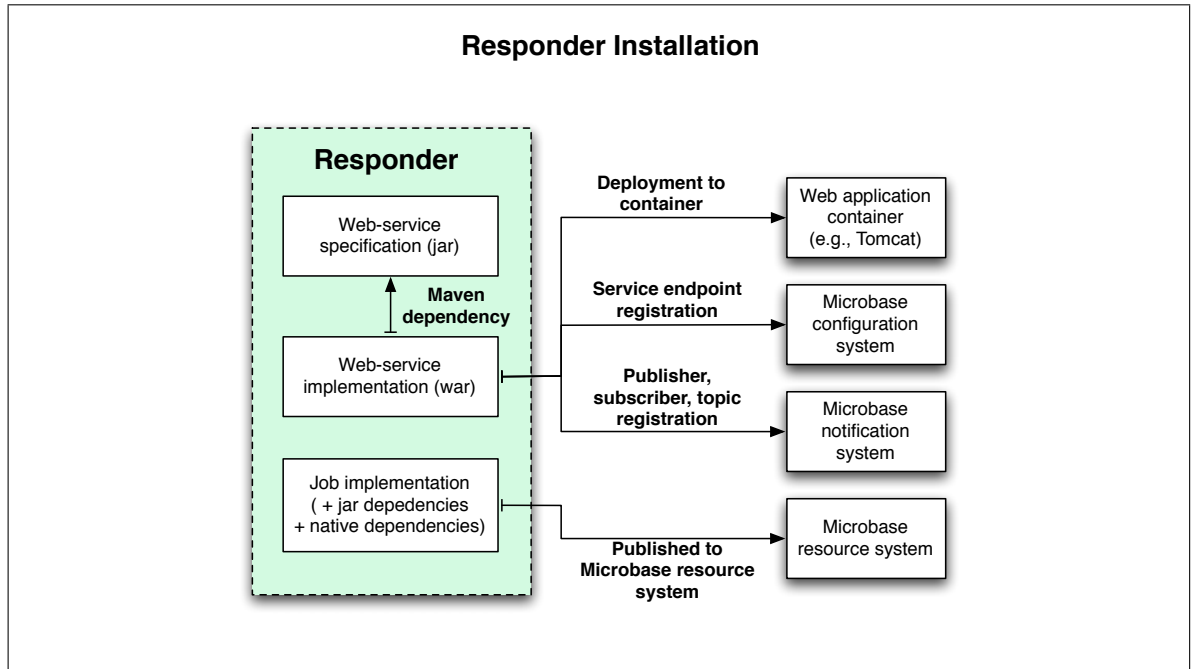


Figure 6.6: Shows the recommended responder sub-projects. A `jar`-based project should contain all the publicly-accessible features of the responder, including: Web service specification interfaces, Web service client factory utilities, data beans used for data transfer via a Web service, and notification message beans. If this layout is followed, responders can be deployed to a system via a provided automated installer utility.

3. A `jar` project containing one or more compute job implementations. Separating computationally intensive work from the management logic (implemented in the `war` file) enforces modularity and provides a convenient package that may be distributed to worker nodes.

More than one event handler and job implementation project are permitted. Microbase only specifies the layout of responder ‘event handler’ and ‘compute job’ sub-projects. The developer is also able to add as many other non-Microbase-related sub-projects as they require to a responder root project. These additional projects are under the complete control of the developer — i.e., they will be ignored by the Microbase installation facilities.

Installing a responder into a Microbase system involves multiple steps: web application component(s) must be transferred to a suitable deployment server; deployed web applications must be initialised; and compute job implementations (and their associated dependencies) must be published to the Microbase resource system. These steps can be automated if the components of a responder are suitably separated, and a machine-interpretable project layout is available. Maven provides such a machine-parsable project description, and the Microbase-provided responder archetypes enforce suitable modularisation. Microbase provides an installation application that is able to interpret responder project layouts. Given a base directory, the installer is capable of searching for Maven

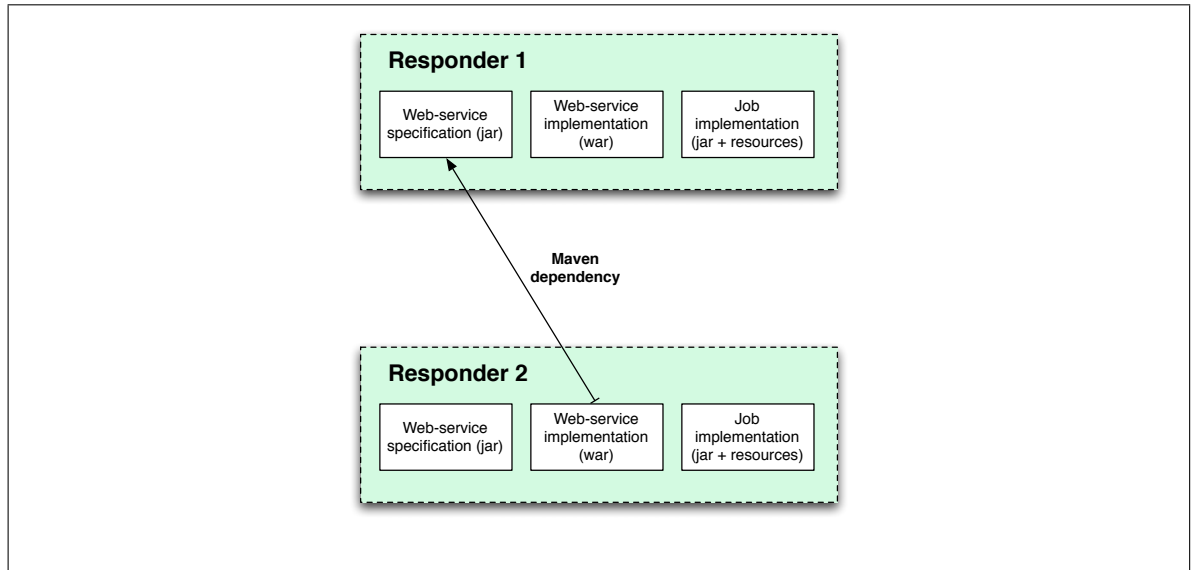


Figure 6.7: Interdependencies between responders. The public Web service specification `jar` file can be used to share Web service client factories, data beans, and notification message parsers between responders.

archetypes that are either event handler or job implementations. Responder projects that follow the suggested project layout therefore benefit from the installation infrastructure provided by Microbase. Details of the installation process can be seen in Appendix A.

Inter-operations between responders are facilitated by sharing the content of the public Web service specification `jar` with other responders via Maven dependencies. For instance, if the second responder in a chain requires information from a previous responder, it may query the Web service interface of the first responder by obtaining an appropriate Web service client, packaged in the public specification `jar`. Access to responder-specific data beans is also possible, since they are also exposed via the public specification `jar`.

6.5.2 Runtime role of Maven artifact information

Job implementation projects may require dependencies on other libraries. Common examples include database clients, Web service clients for other responders, and libraries for parsing files. In order to operate, these dependency libraries also need to be present in the resource system so that they may be installed to worker nodes at runtime. The role Maven plays in enabling a functional Microbase goes beyond project structuring and compilation. Specifically, Maven project data — the ability to uniquely identify projects, and their dependencies — is used at runtime. When a job implementation `jar` project is installed and uploaded to the resource system, the installation tool also checks its direct and indirect dependencies. Dependency `jars` are uploaded to the resource system along with the

job `jar`. In addition, resource system annotations are used to tag the uploaded files with maven `artifact` and dependency information. The presence of Maven `artifact` information attached to resource files enables worker nodes to resolve dependencies at runtime, allowing them to download and dynamically class-load the relevant Java code.

In addition to standard `jar` dependencies, Microbase system requirement also requires the ability to handle platform-native programs. The standard Maven dependency mechanism applies to all platforms, and are therefore insufficient for this task. Using the Maven dependency mechanism would result in every platform native package being ‘required’ and therefore downloaded and installed to every worker node, regardless of the actual worker node platform. The standard Maven `jar` `artifactType` has been extended with an additional directory ‘mb-resources’ to hold platform-specific resources. The advantages for this extension are twofold. Firstly, it provides developers with a consistent location to place packaged platform-native executions. Secondly, this extension allows all platform native files to be uploaded to the resource system at installation time, but only requires relevant files to be downloaded to matching worker nodes at runtime; the dependencies are ‘soft’ - the final link is only made at runtime (see Figure 6.8).

Appendix contains a directory listing of a sample job implementation project.

6.6 Conclusions

This chapter has discussed an application development framework that has been developed for creating applications that can take advantage of the distributed computing facilities provided by Microbase. A responder development framework has been provided for assisting pipeline developers to adapt existing analysis applications for execution within a Microbase environment. The responder development framework presented here makes use of `POJO` style programming in order to hide the complexities of accessing Microbase Web service components directly. The notification system, resource system and job management system are not visible to event handler or job implementations. Additionally, Microbase responders share some similarities with mobile agent-based approaches. For instance, the wrapper layer in [117] is analogous to Microbase responder job implementation wrapper. The wrapper is responsible for specifying job input and output requirements. Unlike the system outlined by Fukada et al., inter-job `IPC` is not directly supported, since `IPC` is usually considered to be highly implementation-specific and Microbase is more focussed on running existing, unmodified programs. Instead, the framework described here provides access to the Microbase resource system, allowing applications to take advantage of BitTorrent data transfers. The Microbase responder event

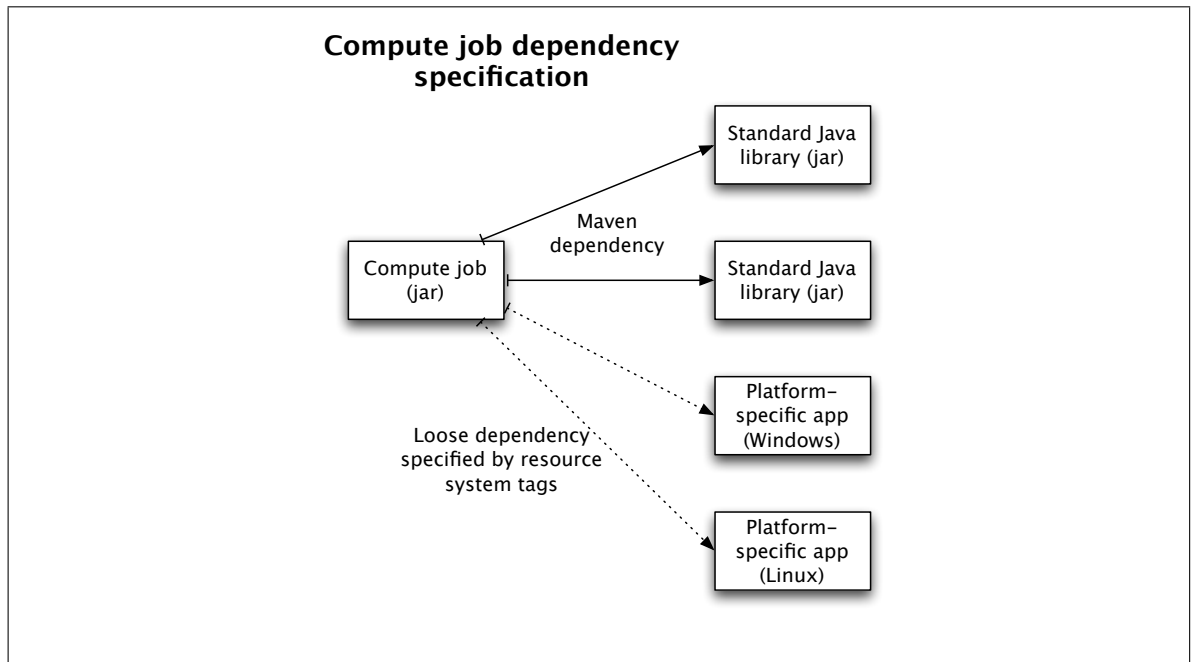


Figure 6.8: Shows dependencies between a job implementation project, and two third-party `jar` libraries it needs to operate. ‘Soft’ dependencies to two platform-native executable packages are also shown. All of the packages shown will be uploaded to the Microbase resource system when the responder is installed. Worker nodes will download and temporarily install all `jar` libraries, because they are specified in the Maven dependencies. However, only the relevant native software package will be downloaded.

handler is analogous to the mobile agent layer in [117]. In Microbase, event handlers are usually static entities on designated servers, but can be redeployed to another application container with minimal disruption, since the event handler will re-register itself with the system at startup.

Instead of interacting with the notification system messaging directly, an event handler must only know how to interpret incoming messages. Likewise, an event handler only needs to specify which message topics it is interested in; it does not need to explicitly register or have a concept of being a subscriber or having a Web service endpoint, since this functionality is handled by the responder support libraries.

In terms of file resource handling, it has been shown that Java annotations on job implementations can be used to initiate BitTorrent transfers at runtime. The developer does not need to have knowledge of where input resource files come from, nor how they are transferred. Non-file based data resources, such as complex object graphs may also be annotated as being input or outputs of a job execution. The job management system will ensure that the appropriate serialisation or de-serialisation operations take place so that these entities can be handled by the Microbase resource system in the same way as ordinary files.

Finally, an automated installation utility makes use of the standard project structure to allow devel-

opers to install the various modules of their responders to the appropriate locations.

Chapter 7

Job management and enactment

7.1 Introduction

In large-scale distributed computation systems, there are often several users needing to execute numerous computationally-intensive workloads on a limited number of hardware resources. By definition, hardware resources such as CPUs and disk storage units are spread over a number of locations. If users were to submit work directly to worker nodes — for example, via [SSH](#) [335] — the result would be chaotic, with some worker nodes becoming overloaded while others sit idle. A job scheduling system must be employed to manage the allocation of computational work to available hardware resources, and to smooth spikes in user demand. Systems such as [LSBATCH](#) [314], [PBS](#) [128], [LSF](#) [58], and [Condor](#) [192] have been developed for managing large computer clusters, accessible via several users. These systems allow users to submit jobs to a central point at any time. Instead of being sent for immediate processing, user submissions are added to a queue. Items in this queue are processed according to the Quality of Service ([QoS](#)) implementation employed by the management system. Strategies might include first-come, first-served (First in, first out ([FIFO](#))), priority awarded to ‘shorter’ tasks, enforced fairness based on the number of jobs a user submits or the number of CPU hours they accrue, or a heuristic-based approach [84, 338, 35]. Using a job management system applies an organisational layer to work distribution, permitting more effective load balancing over the available hardware. In addition, ‘greedy’ users can be accommodated by reducing the priority of their jobs at times of high demand.

Job management systems provide a number of other advantages. For instance, during execution there is a possibility that one or more software or hardware components of a distributed system will suffer a failure. Most job management systems provide facilities to automatically retry failed jobs without

user interaction. Job management systems also often provide facilities to stage input data to, and retrieve result data from remote nodes. Another common feature of these systems is the ability to match the hardware requirements of a particular workload with a suitable worker node, ensuring that large workloads are not placed on inappropriate machines.

7.2 Motivation

In a typical university campus environment, desktop computers located in cluster rooms are often idle, even at peak usage times. At Newcastle University there are approximately 2500 CPUs from computers participating in a campus-wide Condor pool¹. Depending on the time of day, much fewer CPUs are available to the Condor pool since the primary purpose of most of the machines is to support users in their daily work. Operational overheads including data transfers, application installation and user interruptions further reduce the effectiveness of a computational Grid system running ‘in between’ users in such an environment. Nevertheless, harnessing even a fraction of the available power would be worthwhile, providing a valuable contribution of processor cycles required by many fields of computationally-intensive research.

Although heavily used by both staff and students, many campus desktop PCs spend a considerable amount of their time idle, that is, with no user logged in. Many of these computers are located in common cluster rooms and run the university-wide “common desktop”, which is a Windows XP environment customised with a set of applications commonly required by many departments. There is no possibility of administrator access to these machines; all operations must run under the Condor system user in a controlled environment. Other platforms available to the Condor pool (approximately 10% of the machines) consist of 32- or 64-bit Linux machines. Some Linux machines are desktop PCs located in common cluster rooms, while others are rack-mounted server clusters, dedicated to high-throughput computation. Both the Windows and Linux desktop machines remove themselves from the Condor pool when there is a user logged into the local console. However, the resources of Linux machines may be shared between Condor processes and remotely logged in users; i.e., if there is no user present at the local console, then Condor processes on the Linux machines may co-exist with remote [SSH](#) user sessions. If a machine is removed from the Condor pool while processing a unit of work, some or all of its progress may be lost depending on how often the job synchronises its state with an external server.

Before the potential computational power can be harnessed, several properties of the available hard-

¹<http://bsu.ncl.ac.uk/condor/> [accessed 2009/10/02]

ware must be considered:

- The primary users of the machines in the Condor pool take precedence, and may interrupt any worker node at any time.
- The heterogeneity of worker nodes, and changes in the availability of particular platforms require that work distribution strategies processes are evaluated dynamically.
- Worker nodes may need specialist software to be temporarily installed for the duration of a computation.
- There is a need for input data files to be staged efficiently to worker nodes.

The heterogeneity of the available worker nodes in terms of operating systems and processor architectures complicates the issue of distributing work. Condor's 'class ad' system can be used to determine whether a given user job is suitable for execution on a particular worker node. However, the user or software process that schedules jobs for processing must have knowledge about the current availability distribution of particular platforms or processor architectures in order to schedule jobs in the correct ratios, otherwise the distribution of work across platforms will not necessarily be balanced. There is therefore a need for a job management system that can react dynamically to the changing availability of worker nodes in a heterogeneous environment.

In spite of the constraints imposed by environmental properties and administrative policies, we believe that useful amounts of computational power can be extracted from the idle time of Newcastle University desktop PCs. Harnessing this power may reduce the load on existing dedicated computer clusters, while at the same time providing greater energy efficiency from existing infrastructure, given that desktop CPU cycles would otherwise have been wasted.

In addition to utilising under-used desktop computers, there is also the possibility of utilising remote processing resources, such as Amazon's Elastic Compute Cloud [1]. CPUs located in the Amazon cloud can be leased in order to extend the processing power available locally. Amazon CPUs have the advantage that processing will not be interrupted by user logins. However, efficient data staging and software installation operations are essential due to the low bandwidth of Internet connections compared with the throughputs achievable via local Local Area Networks (LANs).

7.3 Requirements

The job management system provides functionality essential for the operation of a Microbase installation. The job management system must support the computational needs of the responders present within a given Microbase installation by providing access to hardware resources in an orderly fashion. Hardware resources must be shared fairly among the processes competing for computational resources. While the primary purpose of the job management system is to fulfil its component-level requirements detailing its work management obligations, the job management system also plays an important role in fulfilling high-level, system-wide requirements. For instance, its presence on all worker nodes puts the job management system in the unique position of being able to assist the P2P resource system. CPU cycles, network bandwidth and local disk capacity of worker nodes can be used to share file distribution loads, and to provide more nodes for the distributed BitTorrent tracker. Communication between worker nodes for these system-level requirements is facilitated by messages routed via the notification system.

The component-level requirements of Microbase job management system are to:

- Accept work submitted from responders.
- Notify responders when submitted work has been completed.
- Hold jobs in a queue until all required input files are available and an appropriate worker node is available to process them.
- The job management system must match jobs to worker nodes based on worker node capabilities, such as CPU, RAM, disk capabilities.
- Maintain detailed logs of job enactments. These logs will assist system administrators tracing infrastructure faults, and developers locating bugs. Logging information also provides an essential source of timing information, allowing system efficiency calculations to be made.
- The job management system should mask individual job enactment failures from responders as far as possible. Jobs must be retried a suitable number of times to be sure that a failure is a result of the job implementation, rather than an artefact of the unstable environment it is executing within.
- Match job requirements to worker node capabilities (CPU, RAM, disk requirements).

- Handle job processing failures due to environmental conditions, such as worker node hardware failure and user login interruptions. Units of work must be migratable so that they may be retried on different worker nodes.
- Provide automated set-up and tear-down of transient execution environments on worker nodes. This includes automated software deployment, and data resource transfers. Post-execution tidy-up operations must clean intermediate temporary files to free disk space for the next job enactment, while ensuring that the result data files are kept long enough to be archived.
- The job management system must facilitate cross-platform job development and enactment by abstracting hardware platforms from job implementations.

The job management system also participates in achieving the following system-level requirements:

- Scalable file transfer requirement (see Section 3.3.2): providing hardware resources, including network bandwidth and disk capacity to the resource system.
- Worker nodes have a responsibility to keep downloaded files longer than are required by an individual node, in case other worker nodes require the same files. In this case, the resource system can take advantage of the additional ‘seeders’, reducing load on the central file distribution servers.
- Worker nodes are required to balance the need to maintain files on their local disks for the purposes of sharing with other nodes, against their own individual requirements for local disk capacity.

7.4 Architecture

The job management system has been designed to serve the computational requirements of responders present within a Microbase system. Its duties include scheduling jobs, enactment of jobs, providing a provenance trail for future auditing and debugging exercises, as well as dealing with a range of potential failures that might occur during job processing. In addition, it shields the responder components from requiring detailed knowledge about the actual hardware and software configurations regarding the pool of available worker nodes. The job management system is composed of a job server component and a job enactment client (Figure 7.1). One or more instances of a job server are deployed to a Web service container, such as Tomcat [113]. An instance of the

enactment client runs on each worker node. The job management system collaborates with other core system services in order to meet its component- and system-level requirements.

When a responder within the Microbase system requires a large amount of computationally intensive work to be completed, it should instruct the job management system to carry out the work, rather than complete the computation itself. The job management system is intended to de-couple components requesting computation from components that provide computational power. Loose coupling between a responder requesting computational work and the task scheduling system means that responders do not need to know implementation- or even installation-specific configuration information regarding the available worker nodes. Therefore, configuration changes can be made to computer clusters (for instance, the addition or removal of nodes) without the need to reconfigure every responder present within an installation. If more nodes are added to the system, or existing nodes are upgraded, responder components automatically benefit from increased performance.

The job management system represents computational work in the form of `jobs` and `tasks`. A `job` is a unit of work small enough to execute on a typical desktop computer. However, many real world problems (`tasks`) require more hardware resources than a single desktop computer provides. Therefore, large computational problems are represented by a `task` composed of multiple `jobs`, each of which may potentially be run in parallel on multiple worker nodes. The responsibility of splitting a computational problem into `jobs` lies with the `responder`, since the work division process is inherently problem-specific.

The process of administrating a computational task is as follows. A `responder` requests computational work by sending a `task description` message (Figure 7.2 on page 141) via the notification system to the job management system. This message contains details of the computational work to be completed, such as the Maven `artifact` information of the responder job implementation `jar`, as well as the Java class name and input parameters to use for each `job`. The descriptions of these units of work are added to an internal queue within a job server instance and distributed to worker nodes appropriately. The job management system then sends a `task completion` message back to the responder once all job enactments have completed. These task reports contain a summary of the overall task enactment, including which jobs ran successfully, which jobs failed and the resource system `UIDs` of result data files.

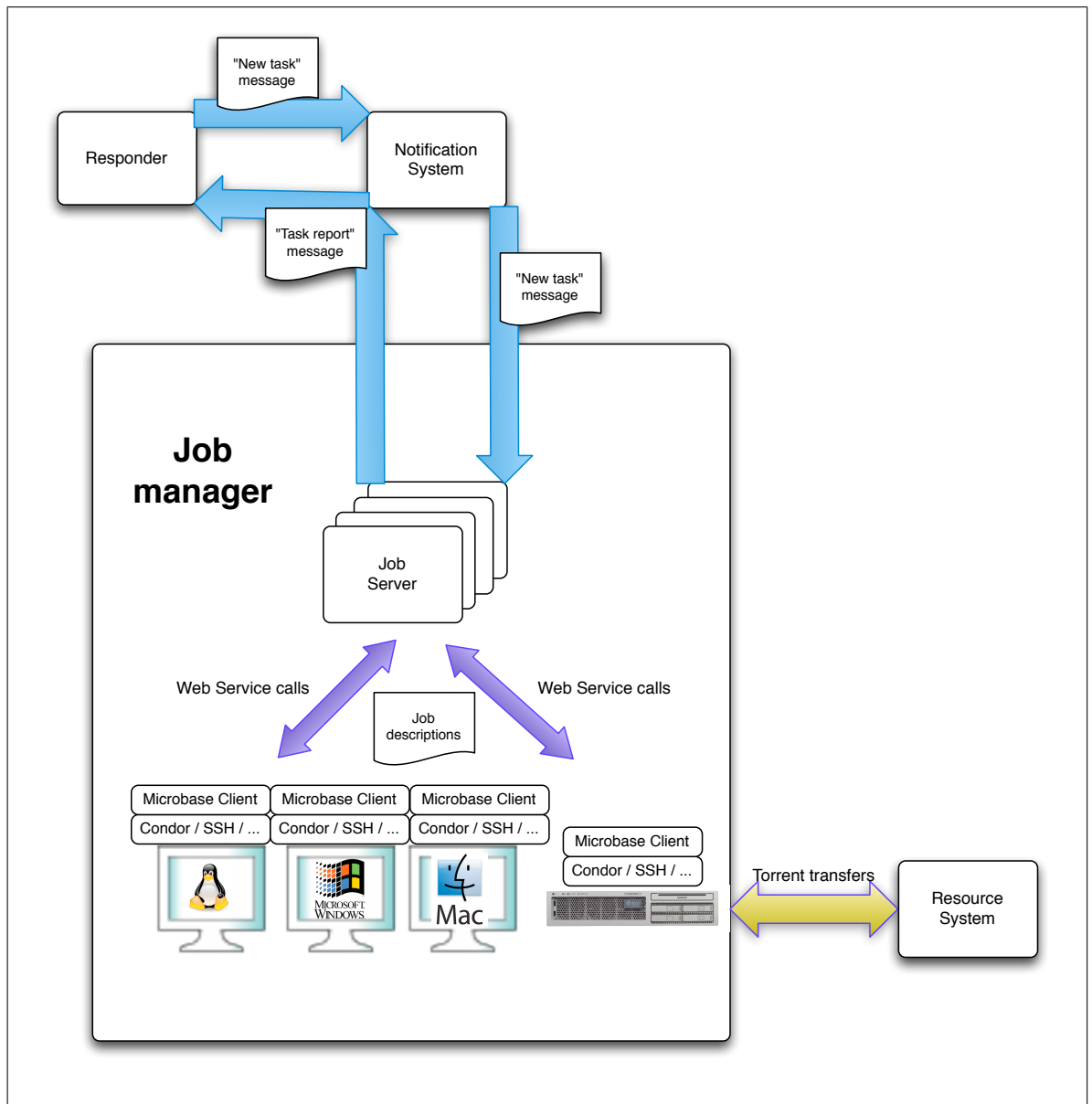


Figure 7.1: The Microbase job management system. Interactions with other Microbase components are shown. Communication between a responder requesting computational work and the server-resident portion of the job management system is mediated by the notification system. Worker nodes running the job enactment client must communicate with the resource system in order to acquire resource files necessary for executing computational work. Worker nodes must also respond to requests from other participants of the resource system to make files available via BitTorrent when required. In this case, co-ordination is achieved via Web service calls and ‘broadcast’ notification messages, while bulk data transport operations utilise P2P BitTorrent-transfers.

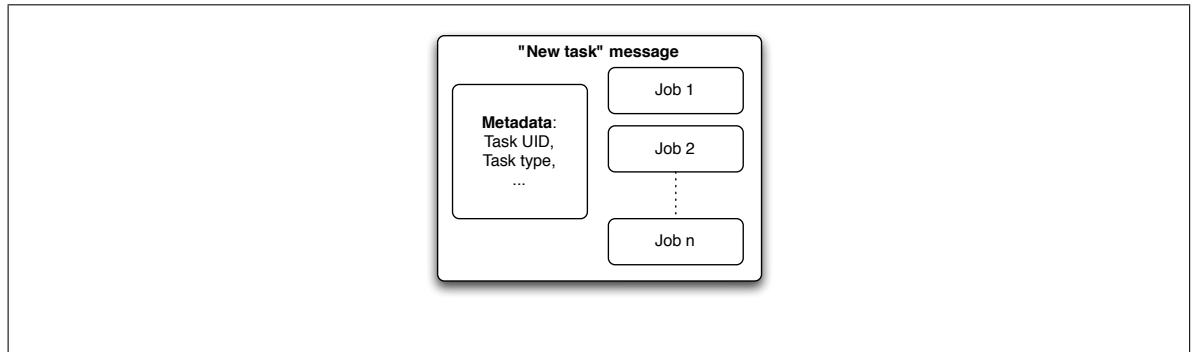


Figure 7.2: Contents of a task description notification message. The description consists of metadata relating to the task itself, such as its type name, the level of parallelisation to use and other high-level properties. A set of descriptions for each job is also provided. Job descriptions specify the executable programs and data files required to execute a job. Large data resources are *not* passed within the message itself. Instead, Resource system [UIDs](#) are used as a reference to large data files.

7.4.1 Failure handling

Job execution failures can occur for several reasons: an environmental failure; corrupt or incorrect input data; or a bug within the job implementation itself. The Microbase task enactment system enables responders to handle each of these failure types.

Environmental failures are not caused by a job implementation bug, but are the result of a problem with the enactment environment itself. Environmental failures therefore have a wide range of potential causes, including hardware failures, network problems, or software-related problems. Environmental failures are often transient. Therefore, Microbase deals with them by retrying a failed job at a later time, possibly on a different worker node. By retrying the job at a later time, problems arising from congested networks or overloaded shared resources (e.g., SQL databases) can be overcome. Re-executing a job on a different worker node overcomes local transient issues, such as a full disk. These types of failure are almost completely masked from the responder. Microbase will re-try job executions, potentially on different worker nodes, many times until a successful execution has been achieved or a retry limit is reached. Only if the retry limit is reached will the responder that originally requested the work be informed that there has been a job failure (see [Figure 7.3](#)).

The Microbase task enactment system allows hot-patching of job implementations. If a job enactment failure is caused by a bug in a job implementation, a repaired version can be uploaded to the Microbase resource system. No other changes or server restarts are necessary, provided that the bug-fixed job implementation keeps the same public interface as the original (i.e., takes the same number and types of inputs, and the same number and types of outputs). The ability to hot-patch job implementations in a large system is important, since server restarts may be disruptive to other unrelated responders.

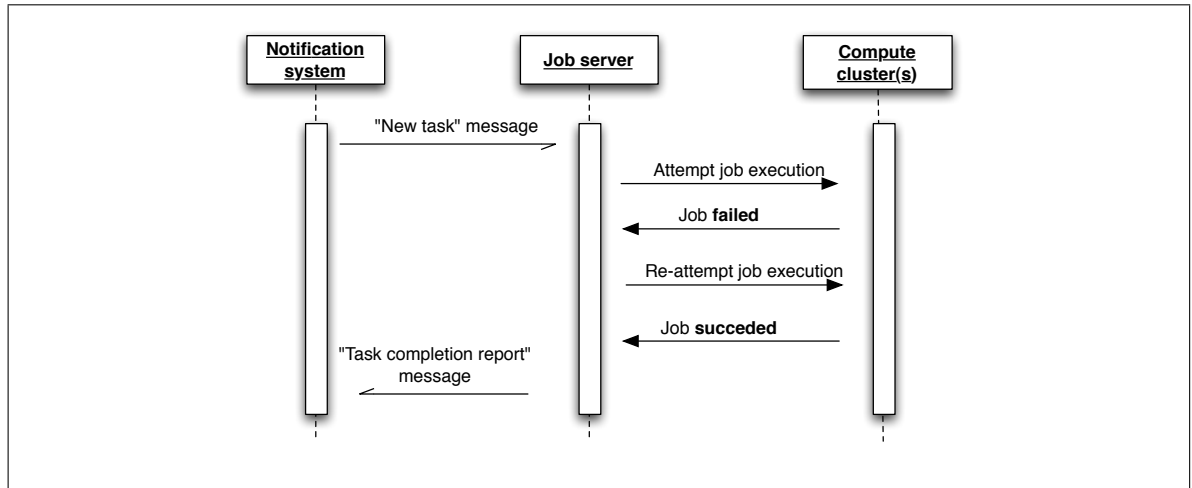


Figure 7.3: The job manager insulates the responders from certain types of job failure caused by the execution environment by retrying jobs on different nodes until there is a successful execution. An execution failure is only reported back to a responder if the job failed ‘too many’ times.

7.4.2 Logging

Provenance is important to long-running, large-scale systems, particularly when the intended environment consists of heterogeneous computer clusters. Informative logs are particularly useful when debugging a system, or when a provenance trail detailing which programs and data combinations produced a particular result is required. The following types of data need to be included:

- [UIDs](#) for every resource stored.
- Data resource version(s) used.
- Executable program version(s) used: this needs to include both the job implementation (.jar) and any 3rd party application distributions, for example, Blast .
- Host name of the worker node used to execute the job.
- Operating system name and version.

It is important to record where (i.e., which worker node) job execution attempts were performed (see provenance and logging requirements [3.3.4](#)). A host name is guaranteed to uniquely identify a computer at any particular point in time. However, node configurations may change over time, perhaps as a result of a hardware change or administration change (for instance reduced disk capacity available to the Microbase client due to repartitioning, or user quota changes). These changes may positively or negatively impact the ability of a particular job implementation to function on a worker node. Therefore, storing only the hostname is not sufficient as it does not take into account hardware

or system configuration information. As a result, the job server database must maintain a record of the worker nodes in terms of their hardware specification and several environmental properties in order to uniquely identify them:

- Worker node host name,
- Hardware capabilities (CPUs/cores, RAM available to the JVM, writeable disk space),
- Operating system (name, version, architecture),
- JVM (Java version, Java VM version, VM vendor).

When a compute client first starts up, it registers itself with a job server, providing this information. As a result of this registration step, the client receives a unique identifier which may then be used to request work. If the compute client is subsequently restarted on the same worker node at a later time, it will receive the same unique id if none of the registration information has changed. However, if one or more details have altered (perhaps due to a memory upgrade, or new JVM version), a new identifier will be assigned. If a worker node change is rolled back (for instance, if an old JVM version is restored), then the previous identifier will be re-used. All job execution attempts are associated with the worker node configuration they were processed with.

7.4.3 File versioning

The Microbase resource system provides versioning capabilities and unique identifiers for every resource it stores, whether data or executable program resources. The unique resource identifiers (**IDs**) play an important part in job execution logs. Each job execution report stores the **UID** of the input resources used, including the **UID** of the job implementation (.jar) used perform the computation. Logging the version of executable file(s) used as well as the data file(s) is important when a Microbase installation may outlive a particular job implementation deployment.

All old job implementation versions are kept within the resource system indefinitely. This is important to ensure repeatability of results, should a comparison between the new and old software versions be required at some point in the future. By default though, new executions will use the latest version of a job implementation.

7.4.4 Overseeing computational work

7.4.4.1 Process of enacting a task

Once a notification message from a responder requesting work has been received by the job management system, the system begins to track the progress of the task (Figure 7.4). Each work unit (`job`) is extracted and added to a job queue. The progress of jobs through the job management system will be discussed in the next section. As far as task progress is concerned, a `task` can be considered to be complete when all of its component jobs have completed, either by successful execution, or by reaching a failure limit.

On the completion of a task, a task report is composed. This report contains details of each job's execution: the resource `UIDs` of generated data files, whether the job enactment was successful or not, execution logs and error reports of failures. The task completion report is published to the notification system. At this point, the task is considered to be completed, and is no longer the concern of the job management system.

During the enactment of a task, detailed logs are stored. These logs contain state changes, timing information and runtime environment information about each job enactment. Log entries and calculated statistics may be queried at a later time through the job management system's Web service interface.

7.4.5 Job enactment

Enacting a job involves complex interactions between several core Microbase components. This section explains the processes that occur from initial submission of a job that has been extracted from a task message, to job migration and execution on a worker node, and finally the completion report and results obtained as a product of the enactment.

Jobs are executed on available worker nodes. The job management system differentiates between a `job` and a `job execution`. Over time, there may be multiple execution attempts for each job, although only one job execution runs at any given time.

There are seven possible states a job can be in. A job can only be in one of these states at any given time:

New a newly submitted job, and yet processed by the scheduler.

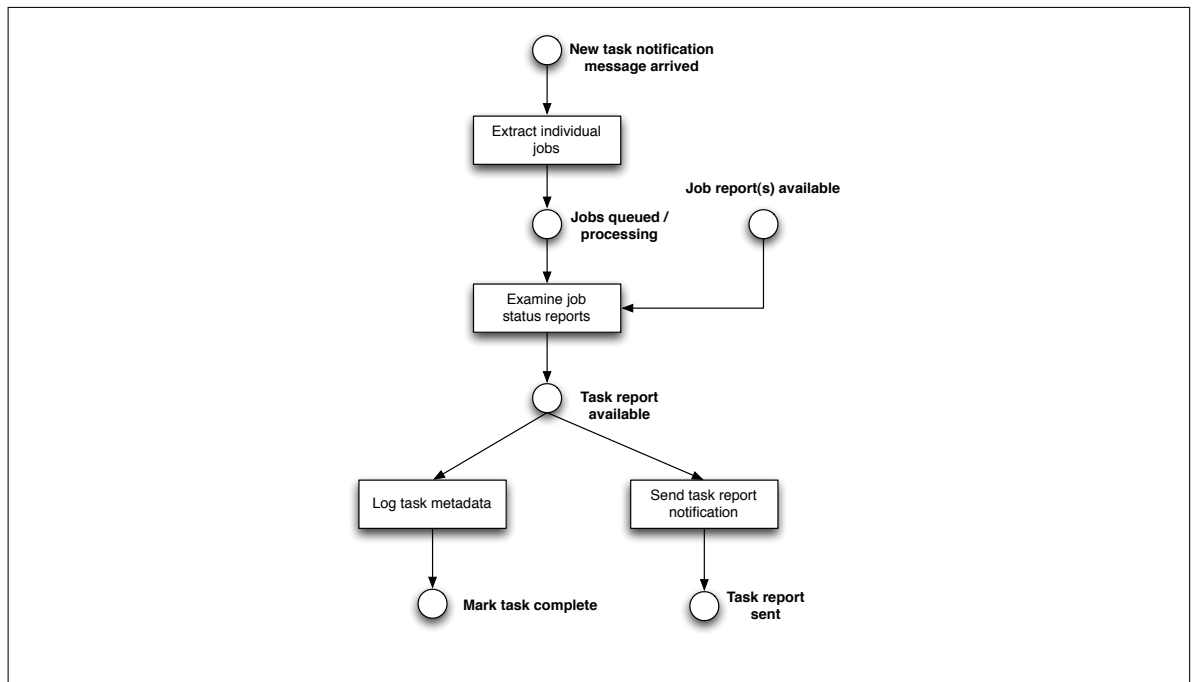


Figure 7.4: The processing of a task by the job manager, from initial receipt of a “new task” message, through to the sending of a “task complete” notification. A task remains in an ‘incomplete’ state as long as at least one of its jobs is queued or is processing. Once all of the jobs of a particular task have been completed, i.e. they have either successfully executed or have exceeded a retry limit, then a task report can be generated. The task report contains an entry for each job. A job entry consists of details such as whether it was successful, the worker node(s) it executed on, how much system time was spent setting up the environment, how much effort was expended on actual processing, which input resource files were requested during processing, and which output files were produced. The task report message is then published to the notification system to inform the responder that requested the computation.

Waiting the job has been recognised by the system, but is not ‘releasable’ yet because one or more required input files are not yet available in the resource system.

Queued the job is ready to execute once a worker node becomes available.

Processing the job has been leased for execution by a worker node.

Archiving indicates that a worker node has completed processing the job, either successfully or otherwise. Output resource files need to be copied from the worker node to the resource system.

Success indicates that job processing has completed successfully and that all output resources were archived correctly.

Failure this state represents a final job failure; i.e., a job that has exceeded its maximum number of retries.

An overview of job state changes is shown in Figure 7.5. This figure fits into the high-level overview diagram discussed in the previous subsection (Figure 7.4) between the transitions “Jobs queued / processing” and “Job report(s) available”.

When a job is first entered into the system, its state is `new`. This means that the entry has been accepted, but has not yet been processed. Jobs in this state are waiting in a queue to be processed by the job scheduling system.

Periodically, the job scheduler checks for `new` jobs. Upon noticing a `new` job, the job scheduler examines the job’s requirements. The job moves into state `waiting`. It remains in this state until its input resources are satisfied (exist within the resource system).

Once a job’s required input files are all available within the resource system, it is ready to be executed. The job’s state will be changed to `queued`, and it will remain in this state until it is chosen to run on a suitable worker node. If a worker node matching or exceeding the system requirements of the job requests more work, the job may be selected to run. In this case, the job is leased to a worker node, and its state is changed to `processing`. The job remains in this state until: a) the worker node reports a successful completion of the job; b) the worker node reports an enactment failure, or c) nothing is heard from the worker node; i.e., the worker node fails to renew its lease after a specified time-out period.

Microbase makes the distinction between failures caused by the enactment environment, and failures of the job implementation itself. Environmental failures include hardware failures as well as failure of any Microbase core service. These failures are distinct from job implementation failures, which

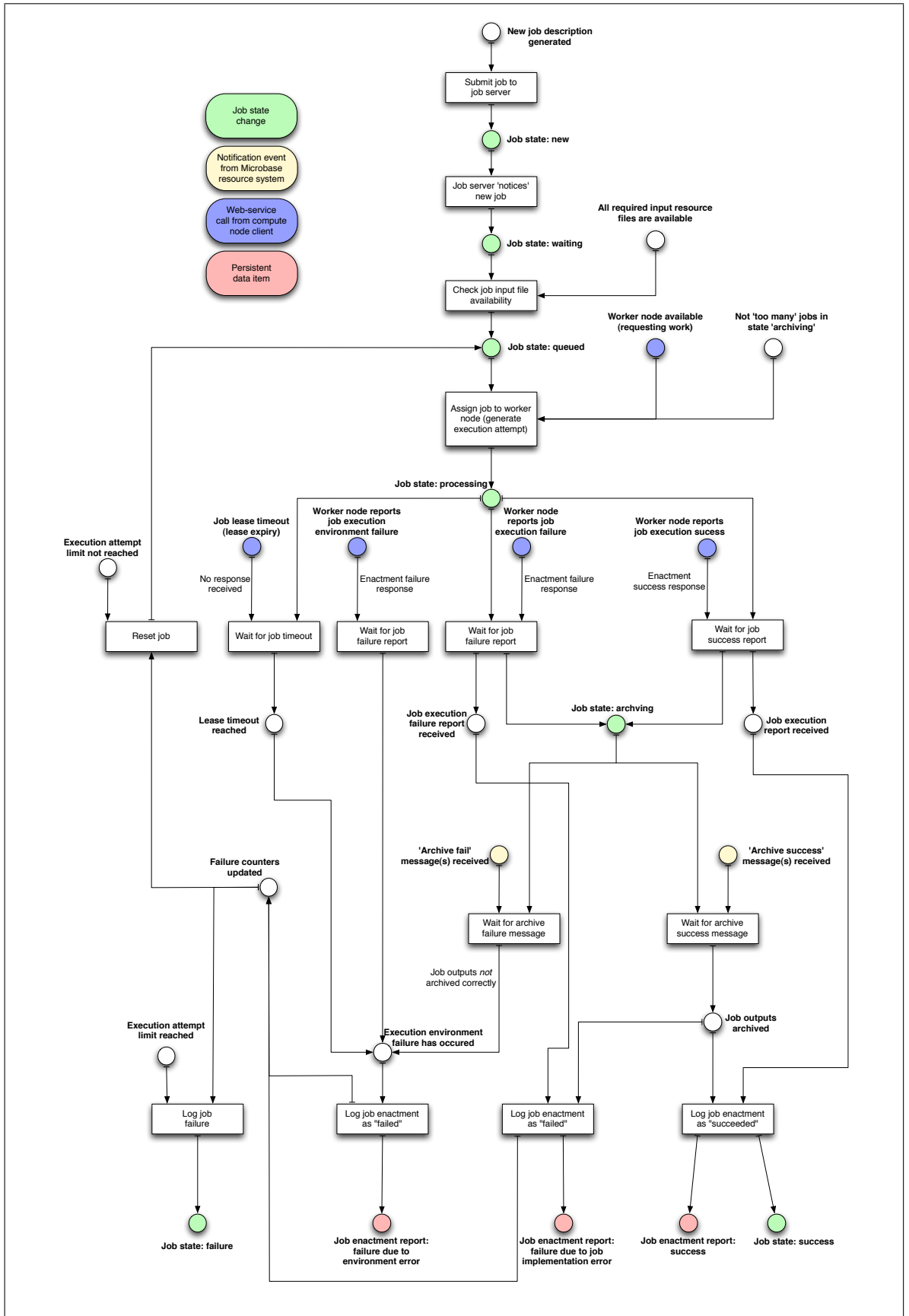


Figure 7.5: The various states a job progresses through during its lifetime, starting with `new`, and ending in either `success` or `failure`.

originate from within the user job itself. These failures may be the result of a bug in a user job, or incorrect data being passed to a user application. The job management system records which failure types occur and appropriate error messages and logs that may aid system monitoring or debugging.

If either the domain-specific job implementation or the Microbase enactment environment suffers a failure, then the job must be retried at a later time. In the case of a job execution failure, rather than simply deleting the results of execution (e.g., output data files), any result files that were produced are archived to the resource system. Although this incurs additional overhead in terms of increased CPU load, network bandwidth and disk storage requirements, having copies of output files, logs, and error streams of a failed process is invaluable when debugging applications running on remote worker nodes.

Two counters are used to track the number of times a job fails: one counter records Microbase environment failures; another for failures originating from within the job implementation itself. If both counters are within their retry limits, then a failed job's state is reset to `queued`, allowing it to be re-executed at a later time. However, if the maximum number of retries for either counter is exceeded, then the job state is changed to `failed`. This state indicates a permanent failure.

In the case where a job execution completes successfully, the job it represents changes state to `archiving`. In the `archiving` state, the job is effectively complete except that the output result files are still located on the worker node that processed it and are therefore not 'safe'. During the `archiving` phase, the resource system is instructed to make a permanent copy of these output files. If it succeeds, then the job execution is logged as successful, and the job state is changed to `success`. If resource archival fails for some reason, for example if the worker node is interrupted before an archiver node can copy the result files, then the job execution is recorded as a failure caused by a fault with the environment.

7.5 Compute client

An instance of the Microbase compute client runs on every worker node participating in the system. The compute client is a Java application, so can be run without modification on any platform with a suitable Java Virtual Machine (JVM). The compute client is intended to provide services for the jobs that will run on worker nodes. These include: registration of worker nodes with a job server; dynamic (temporary) installation of job implementations and third party applications they require; acquisition of input data resources for jobs; a temporary workspace for the job to use as scratch space; publication of result files; and reporting of job execution completion to a supervising job server.

The compute client application consists of several subsystems that primarily operate to provide an execution environment for the job management system, but also play a role in satisfying Microbase system-level requirements. Figure 7.6 shows the conceptual layout of the compute client, and its interactions with other Microbase components. A compute client instance contacts a job server to obtain a job description. The job description is added to a queue of jobs local to the worker node. In order to execute computational work specified in a job description, the worker node must first acquire capability to perform the domain-specific work. This task is performed by the `job dependency manager` using Maven dependency information attached to resource system artefacts, as described in Chapter 6. The dependency manager interacts with the local resource system client instance to download the necessary `jar` files. A standard Java classloader is then used to dynamically add a job implementation to the local Java runtime environment. Meanwhile, input data resources and platform-native software are also acquired via the resource system through the `job resource manager` and the `local software manager`, respectively.

The resource system client embedded in the compute client maintains a local copy of all downloaded files, as well as published result files, allowing the disk space of a worker node to be used as a local cache. By caching previously downloaded files, future job executions that require some or all of these files will benefit by not having to wait for them to be downloaded. The resource system as a whole also benefits from reduced load. The resource system client allows each worker node to become part of the distribution system for files that are also required by other worker nodes. Torrent availability between nodes is co-ordinated via broadcast messages routed via the notification system.

7.6 Job execution by compute clients

The process of acquiring and enacting a job is summarised in Figure 7.7. If the worker node has at least one idle CPU, then it contacts a job server to request a job description. On receipt of such a description, the worker node first ensures that the job lease is updated periodically so as to remain in control of the job. If the compute client fails to update the job lease then the job management system may decide that the worker node has crashed, or otherwise been removed from the pool of available computers. If this happens, the job may be assigned to a different compute client running on a different node.

Since the response from the job server only contains an `XML` description of the job, the next task to be performed is to download the job implementation itself, as well as the input data it requires. If the job has platform-specific binary executable dependencies, only the files relating to the currently

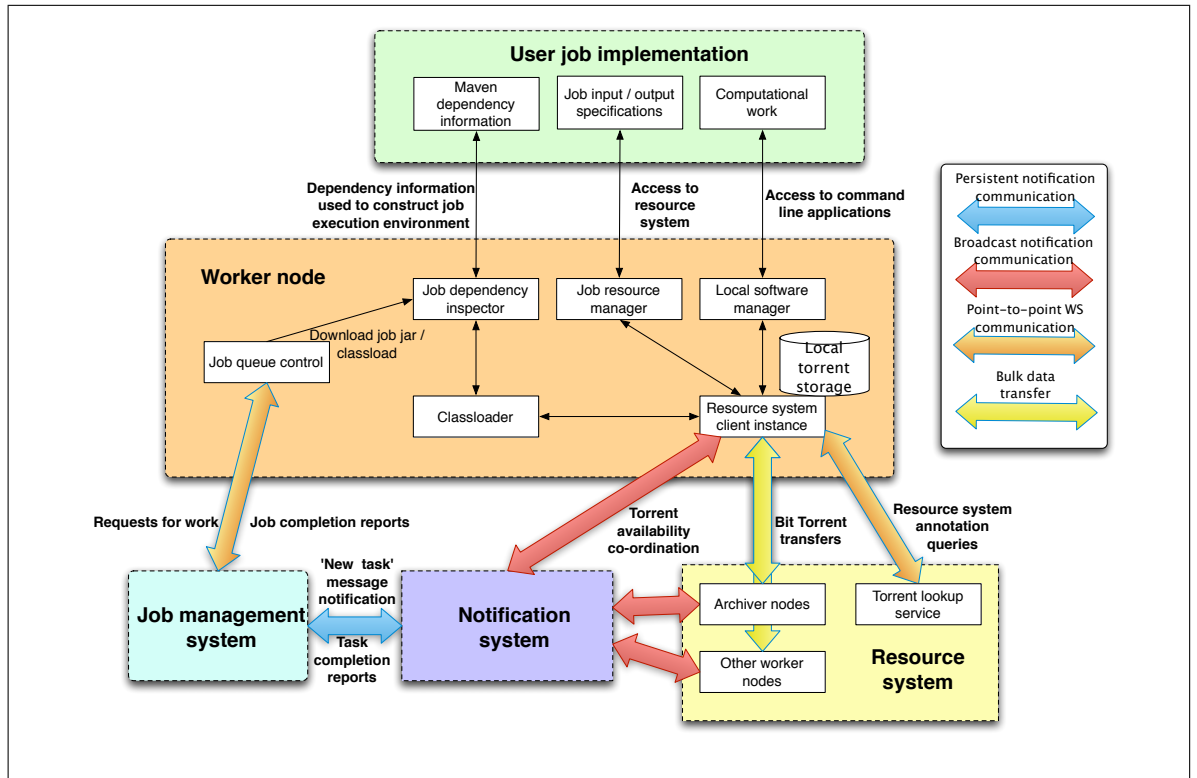


Figure 7.6: The internal components of the compute client and how they interact with other Microbase components. A job queue holds jobs until their required files are all present, and a CPU lease is available to execute them. A dependency manager determines which third-party *jar* files are required to execute the job, and ensures that they are added to the Java class path. A local software manager ensures that required platform-native applications have been installed prior to job execution. The resource system client instance underpins all bulk data transfers, co-ordinates transfers with other worker nodes, and provides remote resource system nodes access to files stored on the worker node's local disk.

running platform are downloaded. If this step fails, then the node is unable to continue. In this case, the job server is notified of an environment failure and the compute client requests another job to process instead.

Assuming all required input resources were successfully downloaded, control is handed to the newly installed compute job. Execution of the job can terminate in one of two ways: successfully, or with an exception. In either case, any output files produced are made available to the resource system for archival purposes. In the case of a successful termination of the compute job, the compute client contacts the job server to report a successful job enactment. If the job execution raised an exception, the job server is contacted to inform it of the failure. In either case, output resource files are kept on the worker node for as long as possible (usually until the disk is full and files need to be deleted, or the compute client exits). Keeping resource files available to the BitTorrent client for the maximum length of time enables other nodes to acquire the resources, even if the current client has no further need for them. This cycle repeats until the worker node is removed from the pool of nodes.

7.7 Performance analysis

7.7.1 Introduction

The computational effort expended by a system can be divided into two parts: the amount spent performing ‘useful’ work, i.e., processing user jobs; and the amount of ‘wasted’ effort expended on system overheads such as network *I/O*. Although any computer system suffers from CPU under-utilisation if it becomes data-starved, the effect is more pronounced in a distributed system where it may be necessary to transfer large files over a network before any processing can take place. In Microbase there are additional overheads including the management of transient software installations on worker nodes, and the requirement of worker nodes to share the file distribution load — i.e., a worker node may need to transfer files that are not relevant to its current job processing to other worker nodes in need of input data.

The performance of the Microbase job management component can be evaluated through the analysis of various timing measurements taken of a running system. In the case of the job management system, we are primarily interested in the amount of user work that can be achieved within a given timeframe on a particular collection of computer hardware. The job management system depends on all of the other core Microbase components at some point during a task enactment. Therefore, determining the efficiency of a given task enactment demonstrates the efficiency of the system as

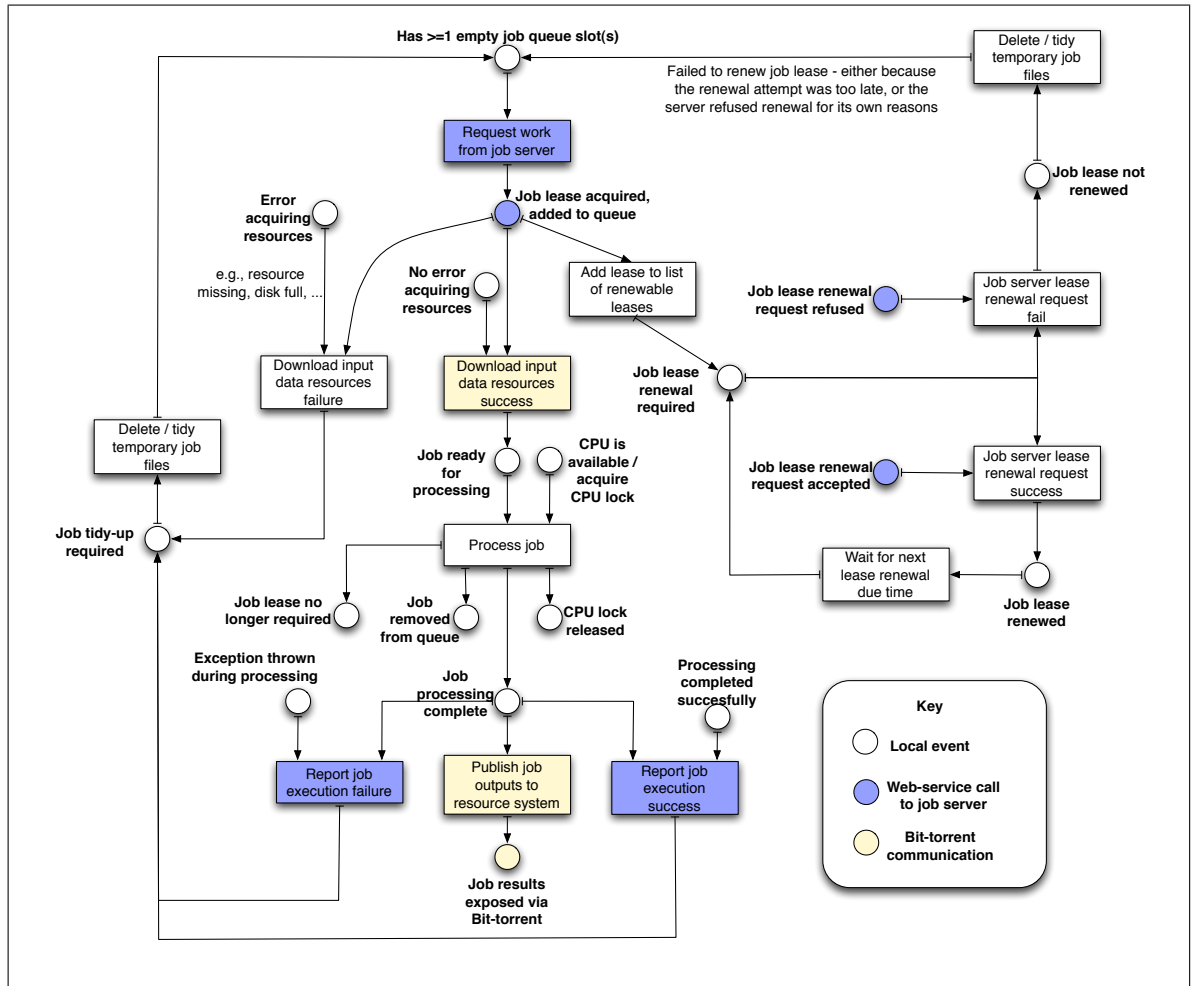


Figure 7.7: The processes involved within a compute client instance when acquiring and enacting a job. Each node has a work queue. The maximum size of the queue is determined by the number of CPUs available, and the amount of job pre-caching that is performed by the node. If there is an empty queue slot, a Web service call is made to one of the job server instances. If there is work available — jobs in state ‘releasable’ — then a lease is acquired, and required data files are queued for downloading. File transfer and job-specific software installation occurs concurrently with other executing jobs. However, processing of the newly downloading job does not start until there is an available CPU; another job has finished processing. On completion of job processing, generated data files are published to the resource system. Temporary data files that do not need to be archived are removed, freeing disk space for the next job.

whole, since the overheads imposed by other system components are also taken into account.

Computational effort may be measured by timing how long a job, including all of the necessary support activities, fully utilise one or more distributed hardware resources. The collected timing measurements can then be used to calculate system properties such as:

- task throughput: the number of processed jobs per time unit
- total system throughput: job rate over all concurrently executing tasks
- pipeline efficiency: average efficiency for each stage of a processing pipeline
- system efficiency: required system effort for a given amount of ‘useful’ work output; i.e., the proportion of computational effort spent performing user-requested work, as opposed to the time spent performing overhead ‘housekeeping’ operations.

Different groups of users may be interested in different aspects of system performance. For instance, end users may be interested in the wall-clock time required to complete their computational work; system administrators are interested in computer cluster saturation, whether adding additional worker nodes improves overall throughput of the system, or whether worker node failures or job execution re-assignments are having a significant adverse effect on system efficiency. Meanwhile, developers are interested in tuning their job implementations to minimise overheads and maximise the amount of system time spent performing ‘useful’ computational work. Therefore, there are a number of ways in which raw performance measurements can be interpreted. In order to accommodate these different viewpoints, a detailed breakdown of how the system spends its time while performing a task enactment is required. Raw timing information can then be represented in different ways to suit the different user perspectives. Typically these different interpretations determine which system events are included in ‘useful work time’, which are ‘overheads’ and which are not included at all.

Timing measurements within Microbase are performed at several levels: an individual job execution attempt; the aggregation of job execution attempts required to successfully complete a job; the sum of all of the jobs that are required to complete a task; and the sum over a set of completed tasks in a pipeline required to satisfy the processing needs of a given set of input data.

The following definitions have been used to clarify various time measurements throughout the rest of this section. For the purposes of benchmarking the system, it is assumed that the processing of jobs, tasks, and pipelines will *eventually* succeed — that is, run to completion without error — even though individual job execution attempts may fail.

Generic terms, applicable to any level of timing granularity (level-specific variations will be defined next):

elapsed time the wall clock time taken to perform a particular job execution, task, or pipeline.

system time the total number of CPU hours consumed while executing a job, task, or pipeline. This time includes both user job processing time and housekeeping overheads such as data transfers between worker nodes. System time is therefore a measure of the total effort expended by a system over a particular period of elapsed time .

overhead time the portion of the system time that is consumed by housekeeping operations or which is spent blocking due to contention for shared compute resources.

processing time the portion of system time observed to be spent performing requested user computation, excluding all overheads such as data transfers. Processing time is therefore a measure of the effort expended by the system on ‘useful’ work.

theoretical maximum processing time the total number of CPU hours available to Microbase over a specified elapsed time , given a particular hardware configuration.

total speedup the quotient of the total system time by the elapsed time .

useful work speedup the increase in performance of the system, given the processing time achieved within an elapsed time .

efficiency the ratio of processing : overhead times.

node utilisation the percentage of a task’s elapsed time for which a node was contributing system time .

7.7.2 Data collection and analysis

Timing information is collected from a number of different sources, including job manager server components as well as individual worker nodes. Measurements are also made at different levels of granularity, from individual job execution attempts through to the pipeline-level timestamps. These timing measurements must be interpreted carefully in order to obtain meaningful statistics. Interpretation is made more difficult by the properties of a distributed system. The inherent parallelism of processes spread over multiple computers as well as multiple threads executing within nodes makes

it difficult to determine how much time is spent performing useful work, and how much is spent on overhead operations such as file transfers.

The timeline of events for processing a typical task is shown in Figure 7.8. While the elapsed time for a task can be used to give some idea of the performance of the system, it is not sufficient to accurately gauge the speedup achieved, especially if multiple tasks from several responders are competing for computational resources at the same time. The elapsed time measurement also does not provide a detailed analysis of how the system spends its effort - how much effort is expended on useful work and how much on housekeeping overheads.

Worker node utilisation

Perhaps a more realistic task enactment case is shown in Figure 7.9. Again, a task enactment consisting of five jobs is displayed. In this case, there are three worker nodes available to complete the work. All three worker nodes are fully utilised until approximately half way through the elapsed time. At this point, jobs 1, 3, and 4 are complete, and job 2 is almost complete. Because only one job is left incomplete (job 5), it is inevitable that not all worker nodes can be utilised with respect to *this* task. From the perspective of the displayed task, the total utilised time is defined as the sum of the time periods for which the worker nodes were actively contributing to the progression of the task; the utilised time for a task is the average utilised times for each worker node contributing work to the system. In a real-world system where multiple tasks are simultaneously active, the 'idle' times shown in the diagram would in fact be used to process jobs from other tasks. The presence of other tasks in the system does not affect how the utilised time for the displayed task is calculated. A high utilised time as a percentage of the task elapsed time indicates high levels worker node dedication to a task. A lower percentage utilised time indicates that worker nodes are each contributing a proportionally smaller amount of their time to processing a task, indicating that the system as a whole may be under high load with many tasks competing for computational time.

Worker node efficiency

Efficiency is the ratio between the amount of effort expended on useful work and the total effort expended by a system. Microbase calculates the efficiency of worker nodes during the portion of time that they are utilised by a particular task. This means that worker node time that is *not* spent processing a task for some reason — i.e., idle time as shown in Figure 7.9 — is not counted as being inefficient.

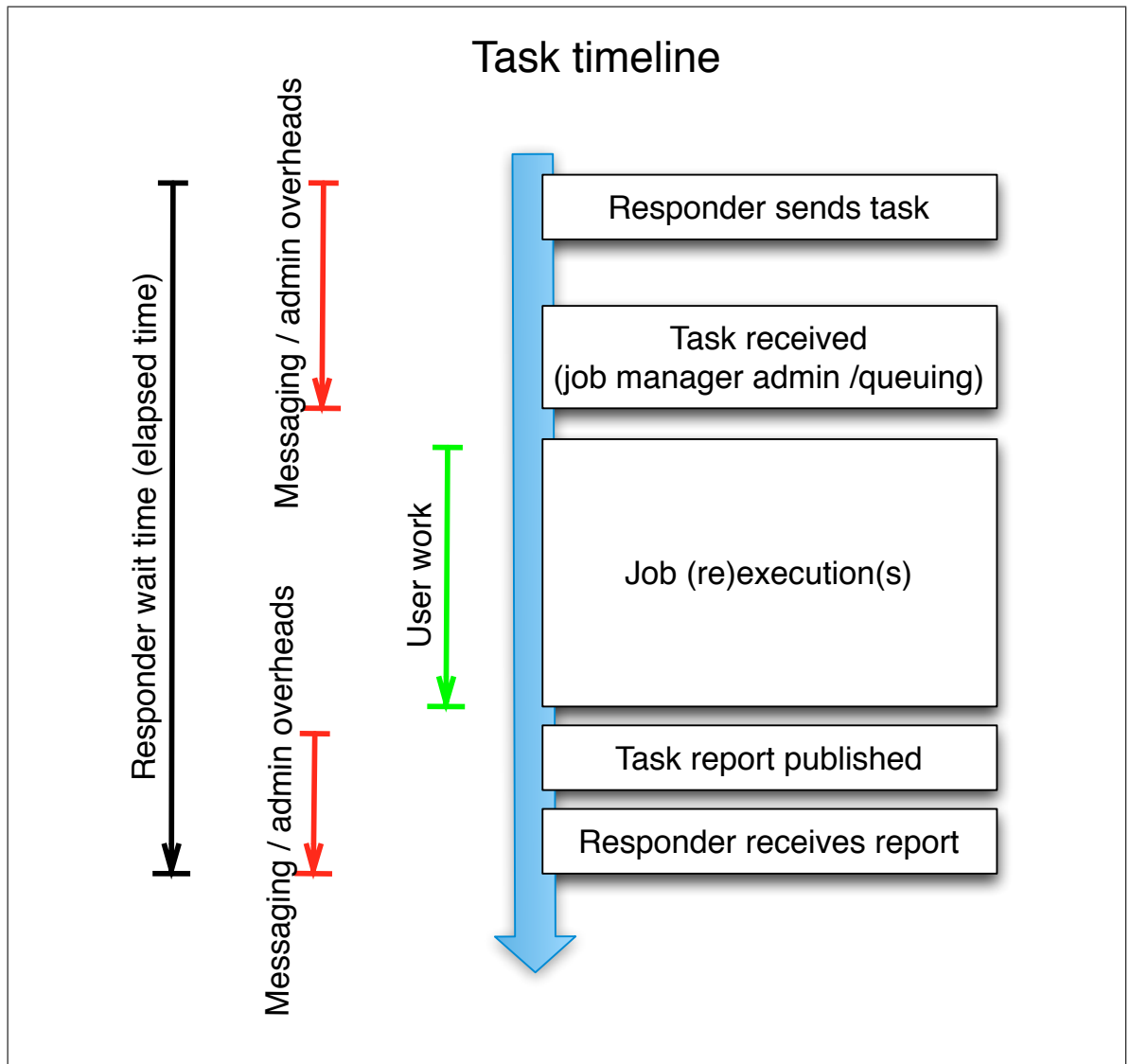


Figure 7.8: Shows the stages involved in task enactment. From the perspective of a responder requesting computational work, the elapsed time of the entire task is the most important. This is the time taken from initial task submission, through to the time at which a ‘task completion’ report is received by the responder. The elapsed time therefore includes more than just job processing times — it also includes overheads incurred through the use of the notification system, administration operations performed by the job management system, and the idle time that jobs spend in a queue while waiting for an appropriate worker node to become available for processing.

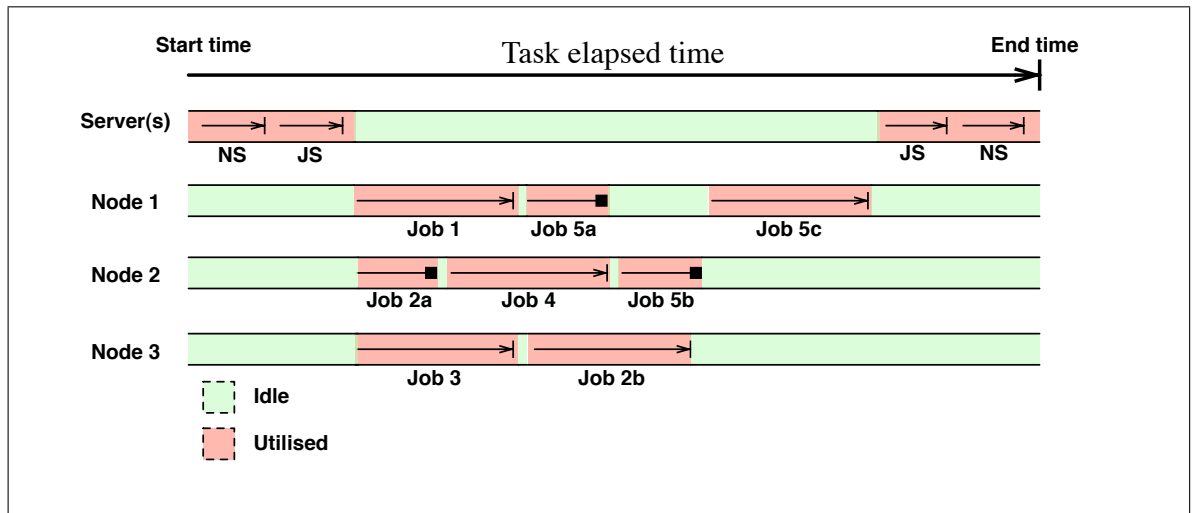


Figure 7.9: The execution of five jobs by three worker nodes. Job 1 and job 3 execute successfully, but the first attempt at executing job 2 fails. After the failure of job 2, node 2 goes on to execute job 4, which succeeds. Meanwhile, the failed job 2 happens to be retried on node 3, where it then successfully completes. Job 5 requires three execution attempts before it finally completes successfully.

Green shaded regions show periods of time where machines are idle with respect to this task. Red shaded regions show periods of time where the machines are fully utilised in processing this task. No worker node is 100% utilised while processing the displayed task of five jobs, and the server appears to have a large portion of 'idle' time. If multiple tasks were simultaneously active, then many of these 'gaps' would be filled by performing operations for other tasks.

There are several housekeeping operations that can potentially reduce the efficiency of a worker node. The 'useful' processing time performed by a worker node is preceded by job execution environment set-up operations, and followed by environment destruction and result archival operations. Environment set-up costs, including data file downloads and software installations, are classed as overhead time, as are operations to archive the results (see Figure 7.10).

The Microbase compute client has been designed to mitigate overheads as much as possible by pre-loading the 'next' job while the 'current' job is executing (see Figure 7.11). The result files generated

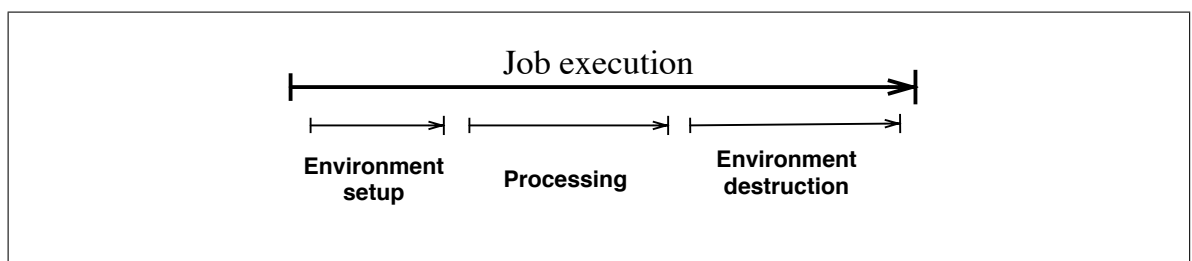


Figure 7.10: Executing a job first involves setting up an appropriate environment. This includes downloading data files and software packages from the resource system. Once the environment has been constructed, user job processing can begin. On completion of the computation, result files must be copied back to a resource system archiver node, and domain-specific software must be removed.

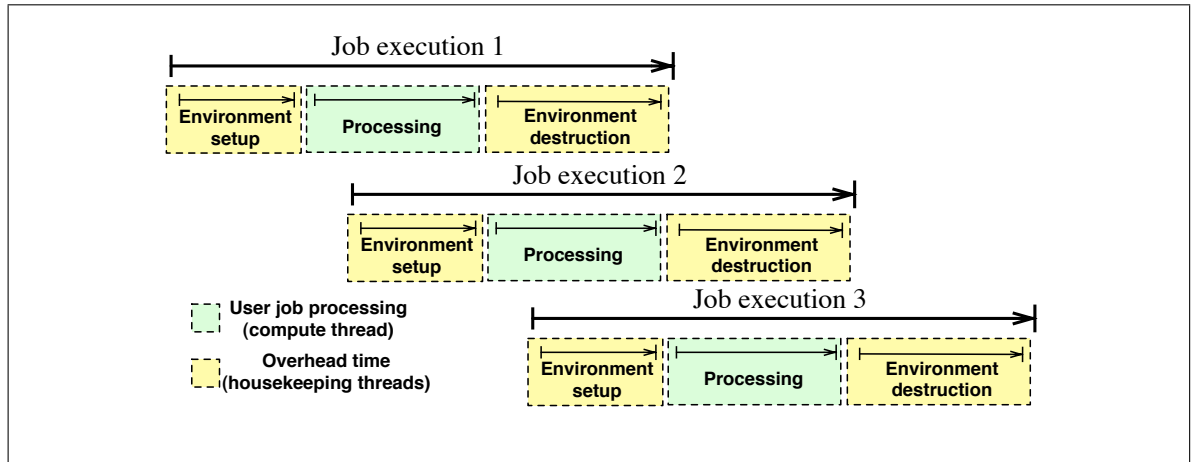


Figure 7.11: Shows a worker node with a single CPU processing three jobs, one after another. For each job, there is an initial environment set-up cost, which includes the acquisition of necessary input data files and the installation of necessary software (jar files and platform-native executable programs). Job processing may begin after the environment has been constructed. The next unit of work is requested, and its independent environment is constructed in parallel with the currently processing job. On completion of the first job, the CPU is allocated to the now read-to-run second job. Meanwhile, the results of the first computation are uploaded to a Microbase archive server, and the inputs of the third job are downloaded.

If the working environment of the second job can be constructed before the first job finishes processing user work, then the second job effectively has no set-up overhead. On the other hand, if the working environment of the second job takes longer to construct than it takes for the first job to complete user work processing, then the set-up overhead is reduced, but not completely eliminated.

by the ‘current’ job are uploaded to a server. The job environment is destroyed while the ‘next’ job starts processing. For jobs where the set-up time is smaller than the processing time, this source of overhead is completely eliminated, allowing an unbroken chain of ‘useful’ processing stages (Figure 7.11), permitting very high efficiencies. .

The resource system instance present on worker nodes may put additional stress on the disk and network hardware components of worker nodes. We make the assumption that network *I/O* transfers and the CPU usage associated with these transfers has a minimal impact on user job processing on worker nodes. Previous work has shown that the use of BitTorrent causes spikes in CPU usage compared with client-server transfers, but has a background level of less than 10% on modern CPUs [62].

7.7.3 Timing results

In a Microbase system, individual job executions can fail due to a multitude of potential software or hardware failures. For the purposes of system benchmarking, job execution failures are permitted on the assumption that the overall job enactment ultimately succeeds within the retry limit. In this case,

a failed job execution attempt would simply have a negative impact on overall system efficiency, since additional effort has been expended by the system with no gain in useful work (see Figure 7.9). When evaluating the efficiency of a job enactment, failed job executions are important to end-users since it impacts the time they must wait for their result sets. System administrators may also wish to know how job failures impact the overall efficiency of a system, whereas developers may be more interested in the efficiency of successful executions only if they regard most types of job failure as a property of the environment.

There are many performance-related considerations that should be taken into account when deciding suitability of Microbase for a particular type of computational work. From the point of view of the task submitter (end-user):

- How long does a typical task take to complete in real time?
- How great is the overhead of running a job within Microbase compared to a single machine? If one worker node was used, how much *slower* does the Microbase system run, as opposed to running the task without Microbase on a single node?
- How does the speedup vary with the number of available worker nodes?
- Does the speedup achieved peak at some point, or continue to increase linearly with the number of worker nodes?

System administrators and developers may be interested in evaluating:

- How much of the elapsed (wall-clock) time is due to Microbase overhead?
- If a job performed no processing at all, how long (elapsed time) would it take to receive a completion notification?
- How much of this inherited overhead due to resource archival, and how much is due to notification message processing and administration tasks?
- How much time do jobs spend in a 'queued' state? Does the addition of more worker nodes reduce this time?
- The overall efficiency of the system: how much useful work is obtained from the effort (system time) provided? what is the ratio of useful work to idle time to environment setup time?

These questions can be addressed by collecting suitable timing information obtained from system benchmarks. However, it is clear that wall-clock elapsed times alone are insufficient to answer these kinds of questions. A detailed breakdown of how much time the system spends on particular types of task is required.

7.7.4 Benchmarking methodology

The benchmarking setup configuration included the following:

- Server 1: Dual 3Ghz Xeon with 2GB RAM
- Server 2: Dual-core 2Ghz Athlon 64 3800+ with 4GB RAM
- Server 3: 3Ghz Pentium 4 with 1GB RAM
- Server 4: Athlon 64 3200+ with 1.5GB RAM
- A varying number of nodes from a pool of 83 dual-core Linux desktop PCs. Each machine is equipped with Intel Core2 6300 CPUs and 2GB of RAM.

It was necessary to spread services over multiple machines for several reasons. Firstly, a single instance of Apache Tomcat had difficulty deploying all the services, even when appropriate Java RAM settings were increased. Secondly, it was necessary to test whether multiple instances of resource system archiver nodes had an impact on system performance. Additionally, this benchmark configuration demonstrates the distributability of Microbase core components, as well as the ability of the resource system to scale by providing multiple Web service instances.

7.8 Results

7.8.1 Performance benchmarks

Minimum feasible job execution time

In order to determine the feasibility for parallelising an application for use with Microbase, it is necessary to understand the latencies and overheads imposed by the system. Of particular interest is the minimum amount of computation time per job required before the system becomes ‘efficient’ — when the amount of useful computational work exceeds system overheads. In order to find the

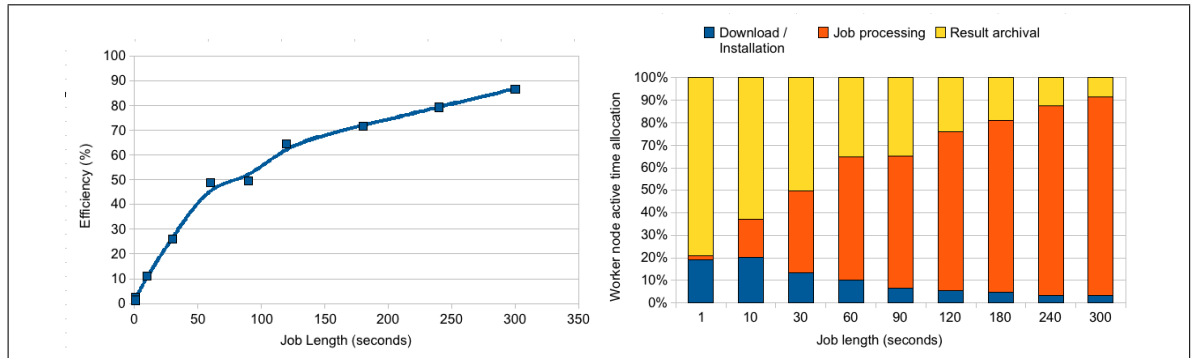


Figure 7.12: A summary of multiple tasks of 20 synthetic compute jobs running on 10 worker nodes. As the job length is increased, overall efficiency increases greatly. To achieve levels of 90% efficiency, compute jobs should last 5 minutes or more.

job computation time threshold at which the system becomes feasible, a number of tasks composed of synthetic jobs of varying length were executed. These tasks were executed with the compute client’s pre-caching feature enabled; while one job is processing, the next is downloaded and prepared concurrently in order to minimise the elapsed time spent performing data transfer operations.

The synthetic job used for this set of benchmarks requires no input files. The only input file(s) required by each node are therefore the job execution jar file and its dependencies. The only output file resulting from execution is a small (1-2kB) log file. The length of simulated compute time was varied to determine the absolute minimum effective job time. Figure 7.12 shows how efficiency increases as the processing time of a job increases with respect to the overhead time.

Distribution of large data files via the resource system

In order to test the effectiveness of the resource system at handling large files with varying numbers of worker nodes, a set of tasks requiring a single 500MB input data file were executed. The time the jobs spent performing ‘computation’ was set at 60 seconds. Each job produces a 1MB ‘result’ file containing randomised bytes. The number of jobs involved in each run was set equal to the number of worker nodes, so that each worker node executed exactly one job. Therefore, in the ideal case, the execution time for each task should be constant. Test runs were performed with 10, 15, 30, 43, and 83 nodes. Since these benchmarks aim to test the overhead of the resource system, job multi-tasking and job pre-caching functionality was disabled since these features result in file operations running concurrently to ‘useful’ work, and therefore mask data transfer overheads to a certain extent. Results are shown in Figure 7.13.

The task completion time clearly increases as the number of worker nodes increases, but it is also clear that the amount of ‘useful’ computational work obtained as a result of adding additional worker

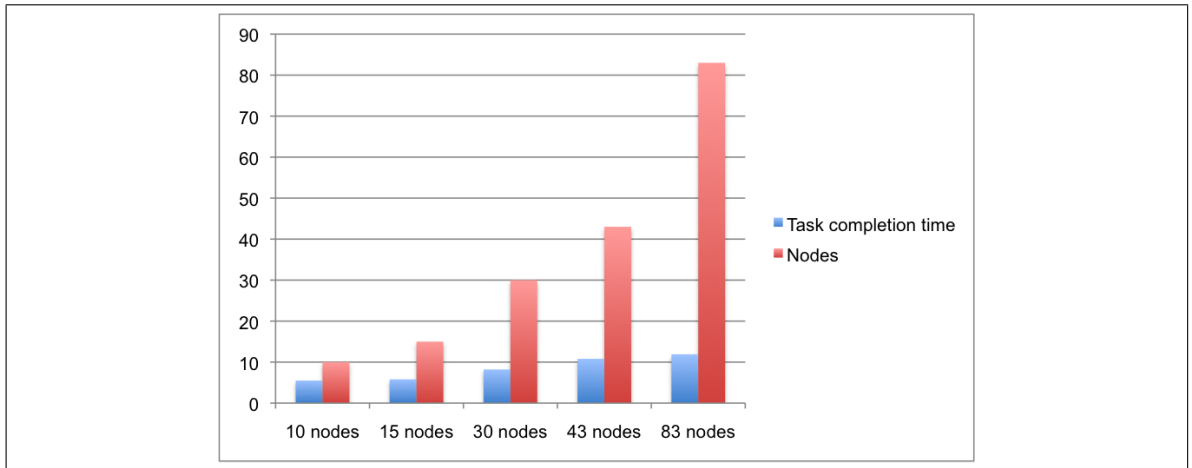


Figure 7.13: File transfer times for a 500MB file with varying numbers of worker nodes. The chart shows task completion time in minutes (blue), and number of worker nodes (red). As the number of worker nodes doubles from 40 to 80, there is only a 10% increase in file transfer times, indicating that worker nodes were transferring data among themselves.

nodes dwarfs the additional data transfer costs. In this test, each job takes exactly one minute to complete. Therefore, 83 minutes of ‘useful’ work were completed in approximately 12 minutes of ‘real’ time. While this is only a 7x overall speedup, this set of tests aimed to show the scalability of the resource system using large files. In the test run involving 10 nodes, approximately 5GB of input data was transferred, while in the 83-node test, 41.5GB of data was transferred, demonstrating the effectiveness of the BitTorrent protocol.

Effect of job pre-caching

In order to determine the effect of job pre-caching on worker node CPU efficiency, the following set of benchmarks were performed. A single worker node was configured to run a set of five jobs, each with a simulated computation time of 90 seconds. Therefore, the theoretical best possible execution time is 450 seconds. A job queue length of 1 means that the worker node will not obtain the ‘next’ job until the ‘current’ job has finished processing. With a queue length of 2, the worker node will process one job while downloading the next. The table below shows the efficiency increases obtained by increasing the job queue length.

Job queue length	Actual duration(s)	Efficiency
1	1588	28.34%
2	882	51.02%
4	590	76.27%

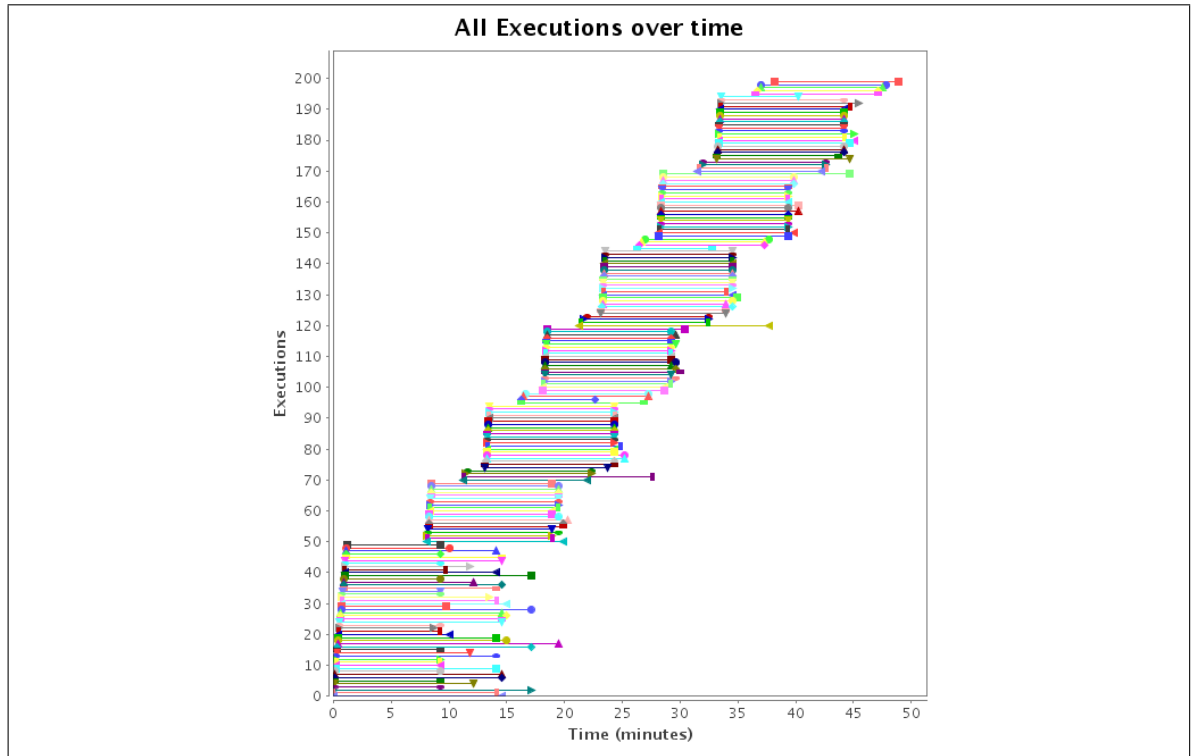


Figure 7.14: The execution timeline of an entire task. Each line represents a job execution attempt. 25 machines were active, so there are approximately 50 jobs active at any time; one executing; one queued. The effect of job caching can clearly be seen, with new jobs becoming active before currently running jobs have finished. Jobs can be seen to complete in roughly blocks of 25, since the synthetic job implementations all have exactly the same ‘user work’ time. However a minority of jobs are shown to take longer to complete than others. The extended execution time of these jobs is likely to be due to a delay in the result archival stage - the job is likely to have finished executing, but for some reason the BitTorrent peer discovery operation to archive the result file has taken longer than usual.

Node utilisation charts

A visualisation of a task consisting of 200 jobs executing on 25 machines is shown in Figure 7.14. The timeline for each job is indicated by a horizontal line. Each horizontal line represents the the start and end timestamp of each job and includes set-up time, processing time, result archiving time and the idle ‘queued’ time that jobs spent cached on a worker node. Each machine pre-caches one job; so each worker node is preparing the ‘next’ job, while the ‘current’ job is processing. Each job requires exactly 5 minutes of raw computation time, although the actual active time will be greater due to system overheads such as job migration. Figure 7.15 represents the same task execution from the point of view of each worker node. A high level of cluster utilisation can be seen. Nodes become idle only at the end of the benchmark, when no unprocessed work remains. In a ‘real’ system, the worker nodes would start processing jobs from another task after at this point.

In this particular benchmark, the actual task running time was 49 minutes. The total ‘useful’ work

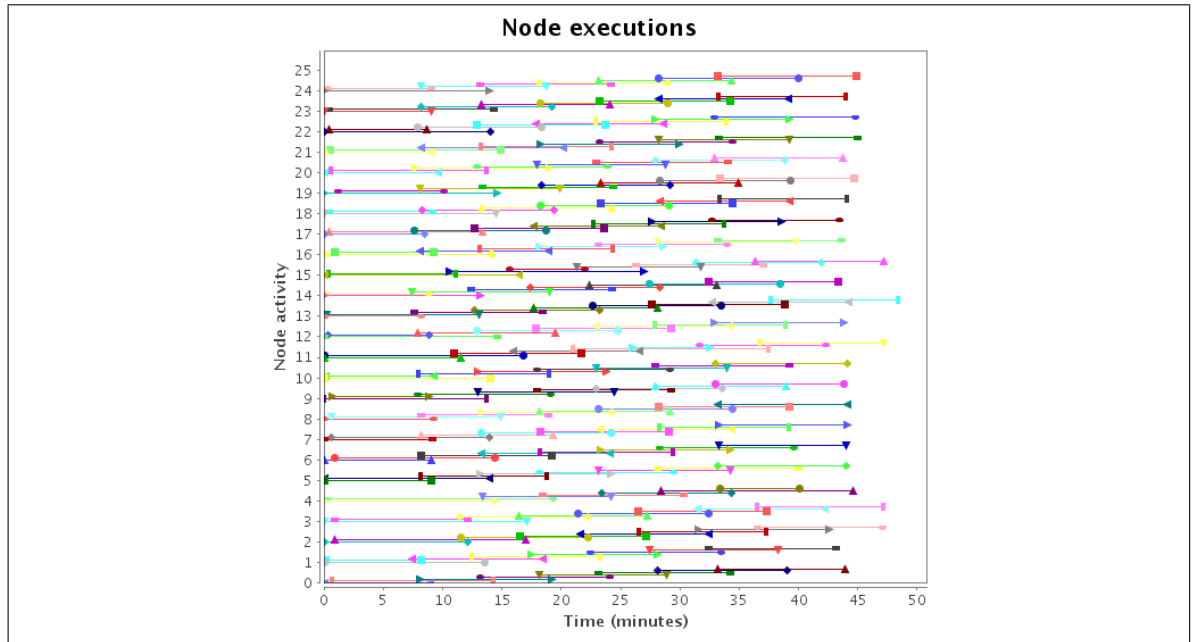


Figure 7.15: Shows the start and end times of each job for each numbered worker node. Overlapping job executions due to pre-caching can be seen clearly. All 25 nodes are shown to be fully utilised for around 45 minutes. Only 5 machines remain active after this time, while they finish the remaining jobs.

time was 1000 CPU minutes, while the time wasted on overheads was 590 minutes. However, because system overheads of job environment preparation and result archival run concurrently with user work, with 25 worker nodes, an overall speedup of 20.2x was achieved. The average node utilisation was 90.4%. The efficiency of each node for its utilised time was 89.5%.

7.8.2 Administration toolkit

In order to monitor a running system, a Web-based monitoring system was developed. The selection of screenshots shown below illustrate the job management system monitoring software while a Microbase system is running: Figures 7.16, 7.17, 7.18, 7.19, and 7.20. The user interface provides facilities for viewing currently executing jobs, and which nodes they are executing on. There is also a facility to start new synthetic benchmark jobs.

7.9 Conclusions

The task enactment system can handle failures with varying degrees of transparency. It is able to almost completely mask job failures caused by environment failures, is able to assist with the handling of corrupt/incomplete/incorrect input data and allows the developer to cope with bugs in the

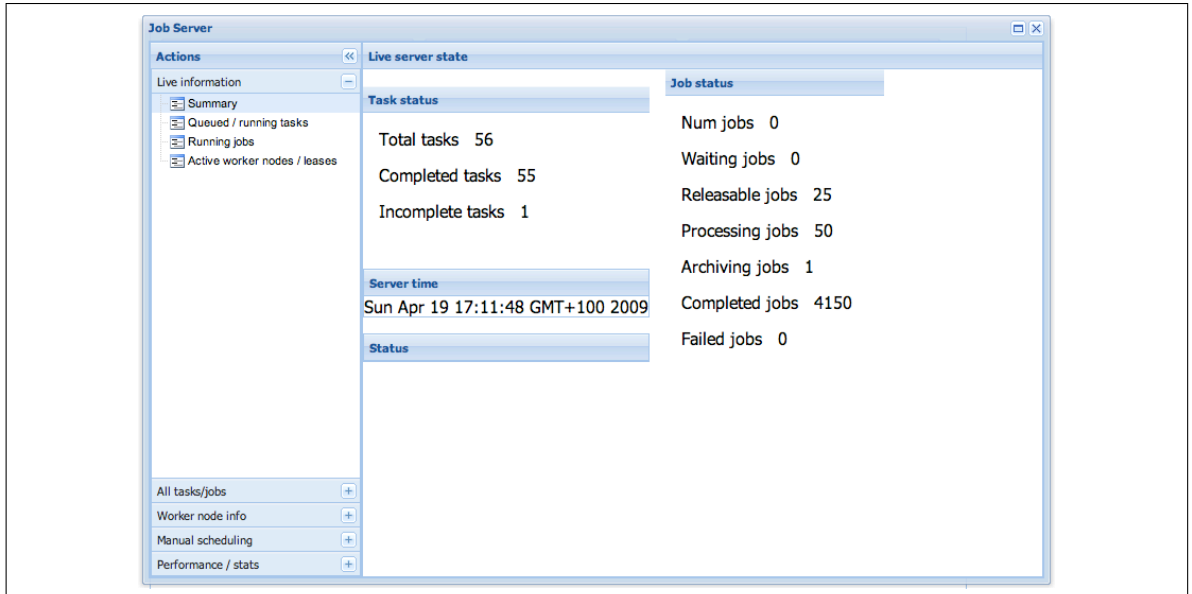


Figure 7.16: A screenshot of the Microbase job server Web interface. An overview of the status of the system is shown. The system shown here is nearing the end of its computational work: 25 jobs remain unprocessed; 50 jobs are currently processing in parallel; 1 job has finished processing, but cannot be marked 'complete' until the Microbase resource system has archived its result files; 4150 jobs have successfully completed.

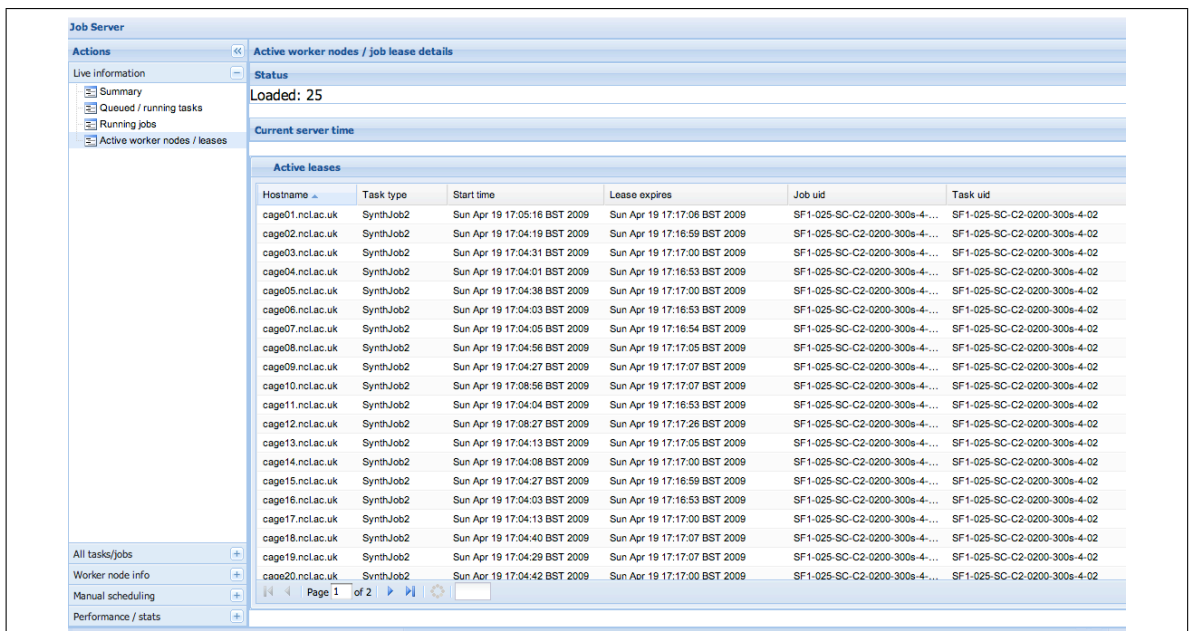


Figure 7.17: This screenshot displays the currently active worker nodes and their associated job lease details. This view allows the user to glance through the active nodes to see what type of job nodes are running, as well as how long the jobs have been running on each node.

The screenshot shows the 'All worker nodes' view in the Job Server interface. The table below lists the details for 33 worker nodes.

Hostname	UID	First seen	Last seen	Archite...	OS	OS ver	CPUs	RAM	Java ver...	JVM vendor	JVM ver
ml14.nclac.uk	7270291e...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:55 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml15.nclac.uk	78641880...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:55 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml16.nclac.uk	402ea3b3...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml17.nclac.uk	aa339f96...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:55 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml18.nclac.uk	9de412ba...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml19.nclac.uk	dcca06b...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:57 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml20.nclac.uk	dd745e0d...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:57 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml21.nclac.uk	aec164ba...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:58 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml22.nclac.uk	69455e4b...	Sat Apr 18 19:16:22 ...	Sat Apr 18 22:47:57 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml23.nclac.uk	58fd1d05...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml24.nclac.uk	bcc96272...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml25.nclac.uk	1e272e1a...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:57 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml26.nclac.uk	500e0e54...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:57 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml27.nclac.uk	e070d8f6...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:57 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml28.nclac.uk	4925010a...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml29.nclac.uk	eeea47c2...	Sat Apr 18 19:16:32 ...	Sat Apr 18 22:48:08 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml30.nclac.uk	3fadf2f7...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:51 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml31.nclac.uk	8e08fa77...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:51 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml32.nclac.uk	75ccdc82...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12
ml33.nclac.uk	dd60eb28...	Sat Apr 18 19:16:21 ...	Sat Apr 18 22:47:56 ...	i386	Linux	2.6.20.11-vs2.2.0	1	775487488	1.6.0_10-...	Sun Micros...	11.0-b12

Figure 7.18: This screenshot shows the types of information stored about each worker node that requests jobs from the system. Nodes are identified by their **UID** which corresponds to a particular set of configuration information, including hostname, operating system version, and various hardware properties. If the software or hardware configuration changes over time, the worker node will be assigned a new **UID**.

The screenshot shows the 'Currently processing jobs' view in the Job Server interface. The table below lists the details for 18 currently running jobs.

UID	Task UID	Task type	Job failures	Job failure limit	Environ. failures	Environ. fr
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25
SF1-025-SC-C2-...	SF1-025-SC-C2-...	SynthJob2	0	100	0	25

Figure 7.19: A summary of the current jobs running within a Microbase system. Various metadata is associated with running jobs, including the number of job execution attempts that have failed. There are two separate failure counts. The first keeps track of failures that occur as a result of an internal error within a job implementation. Another counter records the failures that occur as a result of the enactment environment experiencing a fault.

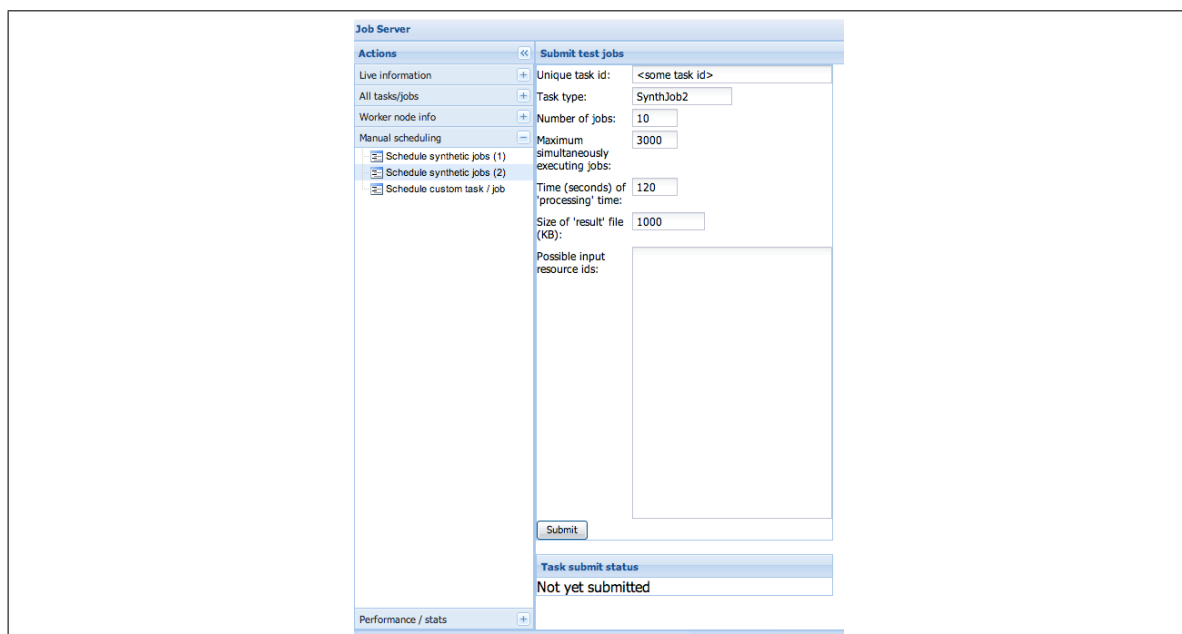


Figure 7.20: Submitting test jobs: the Web interface provides a means for scheduling benchmark test jobs manually.

job implementation projects to some extent. The system has been shown to be tolerant of repeated job failures. Failures result only in increased task completion time.

When assigning job leases to worker nodes, some use of job failure history could be made. If a particular job has been assigned to a specific worker node several times before, with each execution resulting in failure, then perhaps the system should look for another worker node instead. Other statistics could also be used: repeated failures of particular task types on specific nodes, or particular platforms. Or perhaps, a specific worker node is repeatedly failing any job it attempts to execute (potentially indicating a hardware failure, or software configuration problem).

It has been shown that the the minimum job length is around 3-5 minutes due to Microbase overheads associated with job migration. The minimum feasible job length was based on minimal data transfers to the node, so the minimum feasible job length is likely to increase with the length of time taken to acquire data resources. However, data file acquisition has been shown to be efficient with the use of BitTorrent, mitigating resource transfer overheads. Considering many bioinformatics workloads have execution times running from several minutes to several hours, the minimum job time appears to be acceptable.

Cloud computing providers often make use of rapid Virtual Machine (VM) cloning techniques in order to meet the computational demands of consumers [177]. However, such VM image clones all start from an initial shared state. If the software or data required by a remote process is not pre-installed within the VM image, then it must be obtained at runtime, after the VM has booted.

The Microbase compute client, coupled with the resource management system provide a bandwidth efficient method for achieving just-in-time software and data population of worker nodes.

Unlike Condor [192], there is no ‘class-ad’ system, whereby nodes are matched to jobs. There is no need to manually register worker nodes prior to job execution. On startup, a Microbase compute client announces its presence and system properties to an available job server. This approach is advantageous since other properties, such as a list of resource files local to the worker node can be passed, allowing the server to choose jobs for the client based upon which data files or software the client already has.

On receiving a request for work from the client, the job server searches for an appropriate job based on the hardware properties of the worker node. In this respect, the Microbase compute client resembles the BOINC [9] distributed computing client. Also similar to BOINC is the ability of the compute client to download and install necessary software dynamically. Unlike BOINC, however, the Microbase client maintains a local repository of downloaded files which may be re-used by any future job if required.

In addition to computational work, machines running Microbase compute clients also extends the resource file distribution mechanism of by actively seeding a number of resource files they have previously downloaded. Compute clients also respond to broadcast notifications from other clients requesting a particular file for seeding. This cooperative behaviour ensures that files that are in high demand are available from a large number of sources. A number of other desktop Grid systems have demonstrated the advantage of P2P transfers of large files [316, 62, 317]. However, Microbase extends this concept with the addition of dependency links between files. Since compute job implementation projects follow an extended Maven [114] design pattern, the project object model contains rich information regarding project requirements, such as database drivers and other library dependencies. All files downloaded by the compute client are stored in a common directory before being copied to a distinct execution specific temporary directory. Therefore, common resource files required by multiple job implementations will only be downloaded once, thereby reducing network bandwidth and server load. Maven dependencies allow the distinction between different versions of the same library. Since each compute job instance has its own unique Java class-loader, running different compute jobs that require different versions of the same library simultaneously is possible. This functionality essentially comes for no additional developer effort; since responder projects are build with Maven anyway, the compile-time metadata is simply used at runtime by Microbase to prepare a suitable job execution environments.

Chapter 8

Automated Genome Analyser

8.1 Introduction

The availability of complete microbial genome sequences has led to major advances in the understanding of microbial evolution and adaptation as well as a deeper insight into protein function and gene regulation [305, 241, 324, 146]. Many forms of genomic analyses have been developed and new information can be obtained from the examination and comparison of sequences. With the ever increasing number of sequences available, more detailed analyses are becoming possible. However, such analyses are often extremely computationally intensive [205, 256, 195]. The ability of computer systems to scale in parallel to meet this challenge is essential [258]. Several large-scale analysis projects use dedicated clusters of computers for this task, while others are beginning to utilise processing effort ‘donated’ by home or workplace desktop computers [208, 233].

A genome analysis pipeline was constructed partly as a demonstrator application for the Microbase Grid framework, and partly to construct a query-able data source that provides computational access to analysis information, as well as being browsable by bioinformaticians. Microbase was designed to provide an environment suitable for such large-scale, long-running analyses. The [AGA](#) analysis pipeline consists of a set of modules that have been developed using the Microbase responder design pattern described in the previous chapter. The main function of [AGA](#) is to enable existing bioinformatics applications to be executed in a distributed computing, and provide a set of Web services for querying processed data. A separate program, the [AGA](#) browser, consumes data from responder Web services and provides a Web-based Graphical User Interface ([GUI](#)) for browsing integrated result data.

8.2 Motivation

There is a need to demonstrate how Microbase provides a useful resource and is able to utilise real-world bioinformatics data in keeping with the requirements discussed in Chapter 3. Applying a well-known, well-understood analysis application such as `Blast` was carried out to provide a suitable demonstration of the performance of Microbase in all-against-all sequence comparison analyses. The various forms of the `Blast` program make a suitable use-case, since their computational requirements range from the moderately-intensive `BlastN`, to the extremely computationally-intensive `BlastP`.

Several analysis tools were run over available bacterial sequences to construct a data set useful for biologists. The resulting data set must be kept up-to-date by incremental additions as new genome sequences become available. In addition, it is useful for biologists to be able to browse or query generated data sets conveniently, in the similar way to existing visualisation tools. Furthermore, it was also desirable for bioinformaticians to construct ‘canned’ queries that take advantage of the event-driven nature of Microbase. For instance, biologists may not be interested in the entire data set that all against all approaches provide. Instead, they may be only interested in a small subset of the data, such as a set of genes or protein sequences relevant to their research. It must be possible for domain-specific notifications to be sent to notify users when ‘interesting’ data appears.

Suitable access to and presentation of bioinformatics data sets is as important as the efficient generation of data sets. [AGA](#) is intended to provide both a user interface and programmatic access to generated data sets.

8.3 Architecture

The [AGA](#) pipeline is primarily composed of a set of independent domain-specific Microbase responders. Each responder is responsible for a particular analysis type or data storage. The Web service component of each responder provides data set-specific methods, through which it is possible to query result data. For instance, a responder storing genome sequence information would provide methods for retrieving protein sequences belonging to a particular genome entry. An additional [AGA](#) component is a [GUI](#) interface consisting of several applications which draw together the data from each responder into an integrated visualisation tool. [AGA](#) can therefore be conceptualised as consisting of two phases: a data analysis phase that uses Microbase services, and a separate data querying and visualisation phase (see Figure 8.1).

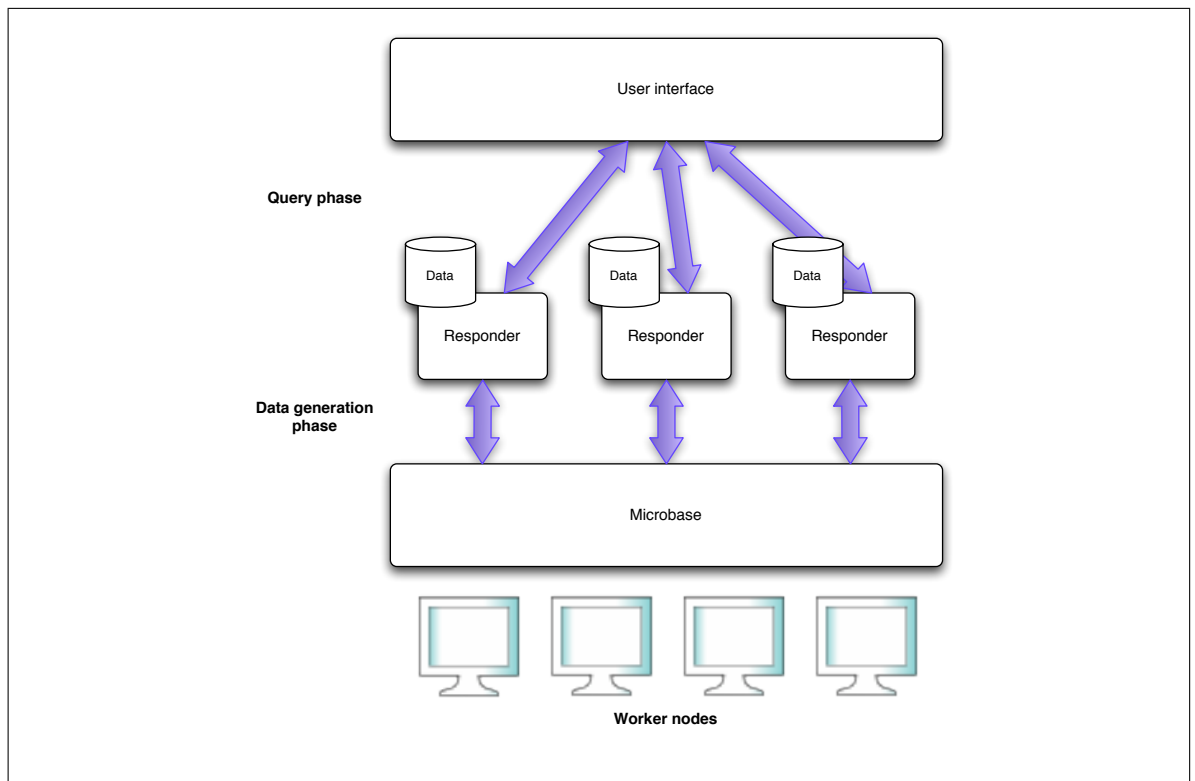


Figure 8.1: Each responder works independently. On receipt of a notification message indicating the presence of a new genome sequence, each responder performs its own assessment of the computational requirements of the task. Responders use the functionality provided by Microbase to schedule and distribute computational work. Each responder typically maintains its own results database. On completion of computational work, the information contained in the separate databases can be re-integrated via the Web service query interface of each responder. In [AGA](#), this kind of integration is performed by the [GUI](#).

The data flow of the [AGA](#) analysis pipeline is similar to other automated analysis systems such as PEDANT [115, 313] in that a list of available sequences must be queried in order to determine if new data is available for processing. New data is then downloaded, ready for processing. An overview of the [AGA](#) pipeline is shown in Figure 8.2. On detection of a new file, the file is first downloaded, and then parsed into a sequence repository by a responder termed the ‘Genome Pool’. The ‘Genome Pool’ then publishes a message indicating that the sequence file is now available in the repository. In [AGA](#), there are several responders that are interested in ‘new genome’ events and more could be added to the system at any time. The current [AGA](#) pipeline contains MUMmer and several forms of the Blast program. These tools populate sequence similarity databases by delegating computationally-intensive operations to Microbase. The resulting data sets can be queried by each responder’s Web service interface. Each responder publishes notification messages to signify a completion event, for example, to indicate the availability of a new Blast report. Currently these notifications are not used within the current [AGA](#) pipeline. They are stored by the notification system for future use, should additional downstream responders need to be added in future.

8.3.1 AGA responders

The work of a responder may be classified into two categories: work that must be performed serially, and work which may be parallelised. Serial processes include the decision of how large computational tasks should be split and database bulk insertions and consistency checking. Processes that can be parallelised include executing analysis applications, and parsing the resulting output files. The server-based component of a responder executes the serial portions of the work, while the compute job component of the responder perform the parallel portions of the work. In [AGA](#), responders work in the following way:

- Responders receive an event notifying them that new data has arrived
- The responder server component performs some initial checks to decide what, if any, computational work must be performed.
- Worker nodes then execute the required jobs. Each job should be as independent as possible from other jobs and from centralised services. Jobs should try not to repeatedly access resources that might become a bottleneck. These characteristics are achievable while running programs such as Blast, since even the output files are deposited in the distributed resource storage system. However some communication with a server is required in order to maintain a

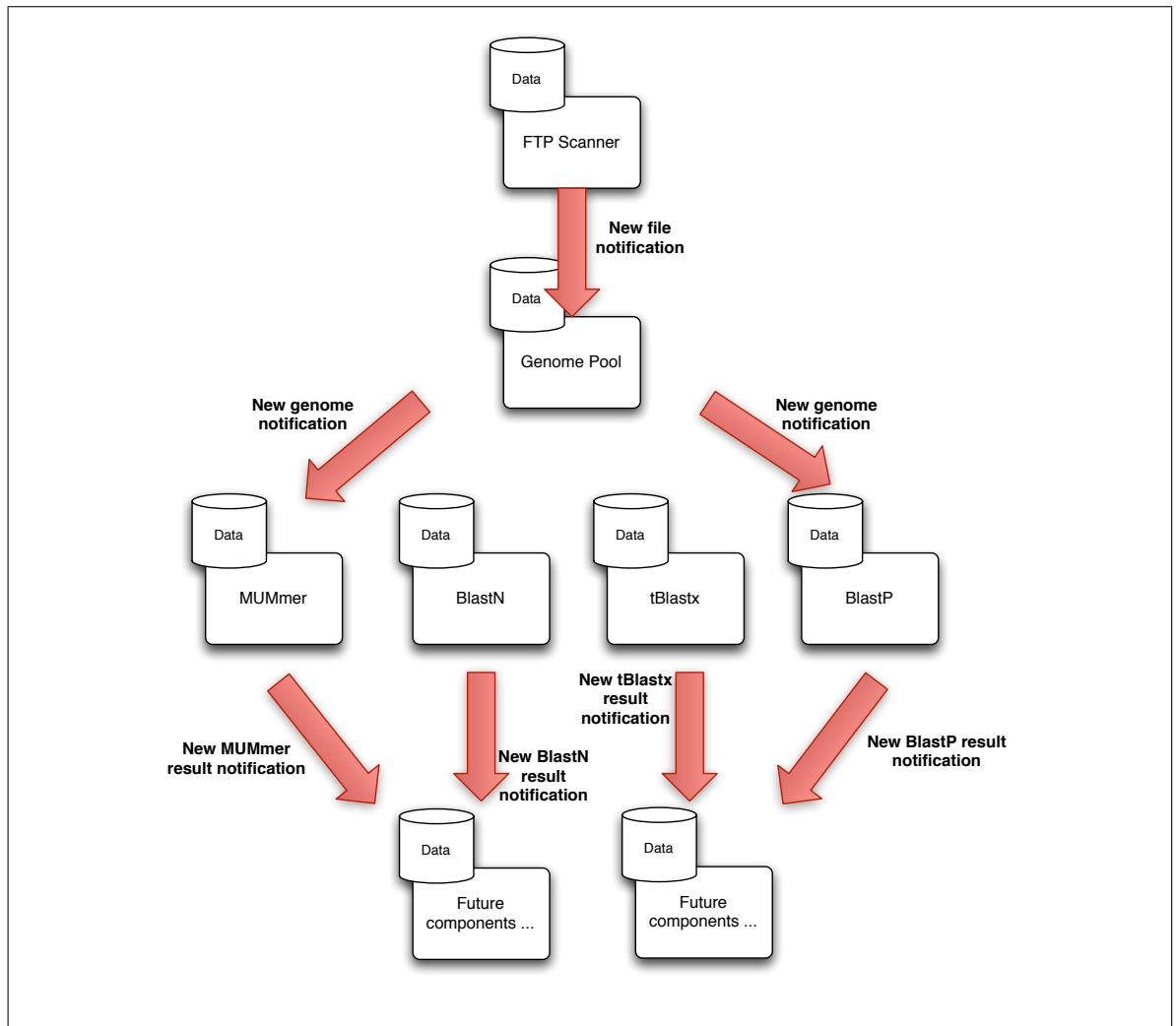


Figure 8.2: Shows the flow of data between responders throughout the [AGA](#) analysis pipeline. The presence of new files is detected by the ‘FTP Scanner’ responder. Other responders can react to the presence of new data files if they are of interest. In this example, the ‘Genome Pool’ responder is interested in ‘new file’ notifications where the file is a GenBank genome sequence. On receipt and successful parsing of the sequence file, a ‘new genome’ notification message is published. Several responders react to the presence of a new genome within the system and schedule their respective analyses to execute within the job management system. On completion of these analyses, each responder publishes its respective analysis completion notification message.

structured data store. For example, after `Blast` has executed, the worker node will parse the output file into a structured form, pass it to a server, which inserts data into a database (several approaches are compared below).

- Meanwhile, the server component awaits a notification indicating that all jobs have finished. On receipt of this notification, the server based component may opt to perform additional work. The types of task commonly performed at this stage include data insertion or data consistency checking.

Different AGA responders have been designed in slightly different ways, partly due to the iterative software development approach used. This chapter evaluates these approaches and concludes with the ‘best’ way to write a responder in order to make the most of available computer resources.

Remote file scanner responder

The remote file scanner is the means by which new primary data can be recognised and automatically imported into the processing pipeline. The file scanner responder could potentially be re-used in other pipelines since it is only concerned with files and has no concept of genome sequences or bioinformatics data formats.

The file scanner responder contains a server-based event handler that may be configured to periodically scan a remote File Transfer Protocol ([FTP](#)) site for particular types of file defined by their file extension. For example, the responder could be configured to find all `.gbk` files located within the NCBI [\[27\]](#) [FTP](#) server. Scans may either be of a single directory level, or recurse an entire directory tree. The first scan of a remote site results in a set of notification messages containing details of every matching file. Subsequent scans result in notifications detailing changes that have occurred since the last scan, such as new files, file deletions, and changes in file properties including length and timestamps.

The compute job implementation of the file scanner responder simply downloads a file from requested Uniform Resource Locator ([URL](#)) and exposes it to the Microbase resource system. The file scanner responder is therefore a convenient means of importing data into the system. Once archived by the resource system, data files are available to any current or future responder that requests them.

Genome Pool

The Genome Pool responder is responsible for maintaining an up-to-date service-oriented repository of currently available genome sequences. The Genome Pool reacts to new genome files being made available via the remote file scanner responder. It must then populate a structured database with the content of these files. Finally, for each successfully parsed file, a notification message is sent in order to inform downstream responders of the availability of a new genome entry.

The Genome Pool responder, like the other major responders, has followed an incremental development cycle. The initial version of the Genome Pool stored genome sequence files directly as binary objects in a database, together with additional indexed metadata for querying purposes. The initial version was adequate for locating genome sequence files and their annotations, but lacked the ability to perform rich querying. This initial version was finally superseded by a more functional replacement database and query service written in collaboration with Nakjang [230] as part of her doctoral dissertation examining high-throughput analyses of surface and extra-cellular proteins. The remainder of this Genome Pool description describes the new version, which is currently unpublished.

The Genome Pool responder was developed to parse genome files in GenBank [27] format into a structured database. The server-side component is responsible for reacting to incoming events, as well as maintaining an [SQL](#) database. The event handler module also provides Web service query methods for accessing the various types of data stored. The Genome Pool database is responsible for providing a query-able repository of genome sequences and their associated annotations. This is achieved via the [SQL](#) database and its associated Web service query interface.

When a new genome entry is added to the database, a message is sent to the notification system to inform other downstream responders of the availability of the data. The notification message sent by the Genome Pool contains the following details:

- The Microbase resource system [ID](#) of the genbank file
- the Microbase resource system [IDs](#) of two FASTA -formatted files generated by the Genome Pool.
- GenBank accession number
- Genome file version
- Taxonomy information
- Organism name and description

- Genome type (chromosome, plasmid, mitochondria)

A Microbase compute job implementation provides the parsing functionality for the Genome Pool responder. The compute job module parses the GenBank file directly into the database via Java Database Connectivity (JDBC) SQL statements. In addition to the database insertions, two FASTA format files are created by the job, containing the nucleotide and amino acid sequences respectively. These FASTA files are uploaded to the Microbase resource system. The purpose of these two files is to assist the scalable distribution of sequence data to worker nodes. Sequence data for an organism is typically a couple of megabytes in size and is potentially required by large numbers of worker nodes simultaneously, so a centralised database repository is not a scalable solution. Therefore, in addition to providing a rich centralised query interface, the Genome Pool responder also publishes sequence data to the Microbase resource system. Worker nodes may then perform efficient bulk data transfers of sequence data via BitTorrent.

The Genome Pool responder has been designed as a generic, reusable component and it has been used within the AGA pipeline to provide downstream responders with genome sequence information. However, the programmatically-accessible Web service query interface of the Genome Pool facilitates its use as an online genome database in its own right.

BLAST-N responder

The BlastN responder runs the well-known Blast tool [6] in a pairwise fashion. The BlastN responder (see Figure 8.3) is responsible for reacting to ‘new genome’ notifications published by the Genome Pool responder. The BlastN responder consists of a Web service component, and a compute job component, as defined by the responder design pattern introduced in the previous chapter. The BlastN responder maintains its own relational database for storing blast result data and status information.

On receipt of a ‘new genome’ notification message, the Web service component of the BlastN responder adds the sequence identifier obtained from the message to a local database table. Next, a set of job descriptions are generated that together represent the pairwise comparison of the new sequence against each existing ‘known’ sequence. The jobs are then scheduled together as a BlastN task and are submitted to the Microbase job management system by sending a ‘new task’ notification message. For the BlastN responder, all of the information required to schedule compute jobs is either obtained from the notification message, or the database owned by the BlastN responder.

If additional metadata about the new sequence was needed, it could be obtained by querying the Genome Pool Web service.

The compute job implementation responder component is relatively straightforward. It takes two FASTA -formatted nucleotide sequence files as inputs. Behind the scenes, the worker nodes executing the compute jobs acquire nucleotide sequence files via the Microbase resource system, rather than querying the Genome Pool responder for sequence data directly. Therefore, worker nodes requiring the same input sequence can potentially obtain the data from another worker node, rather than a central server. The standard `Blast formatdb` command is run with one of the sequences, followed with an appropriate `blastall` command. On completion of a `Blast` analysis, the raw `Blast` report is marked for upload to the resource system. Meanwhile, the content of the report is parsed into an object object model consisting of a 'report' object and set of 'hit' objects. The object model is sent to the Web service component of the responder for insertion into the `BlastN` relational database. Although the raw `Blast` report output file is not used by [AGA](#) beyond parsing its content into the server-based structured database, permanent storage of the raw report file within the resource system is necessary for facilitating future extension of the pipeline. A responder developed in the future may require access to the original raw `Blast` report, rather than the structured data stored by the responder.

On completion of all `BlastN` jobs for a particular 'new genome' notification, a 'new `Blast` result' message is published to the notification system. Currently, this message is not used by any existing [AGA](#) component. However, if a future responder were to be added that consumed `Blast` hits or `Blast` report files, then this message could be used as a suitable hook to which the new responder could be attached.

Pairwise BLAST-P responder

The pairwise `BlastP` responder works in a similar manner to the `BlastN` responder. The proteomes of organisms present in the Genome Pool are compared in a pairwise manner, resulting in n^2 comparisons for each proteome. Each compute job executes a single pairwise comparison.

During the development of the `BlastP` responder, it became clear that improvements had to be made to improve scalability. For a given set of input data, the `BlastP` result database was typically found to be an order of magnitude larger than the `BlastN` database. The database server became overloaded and resulted in many worker nodes remaining idle while they attempted to insert results into the database. To overcome this bottleneck, responder operations were reordered as shown in

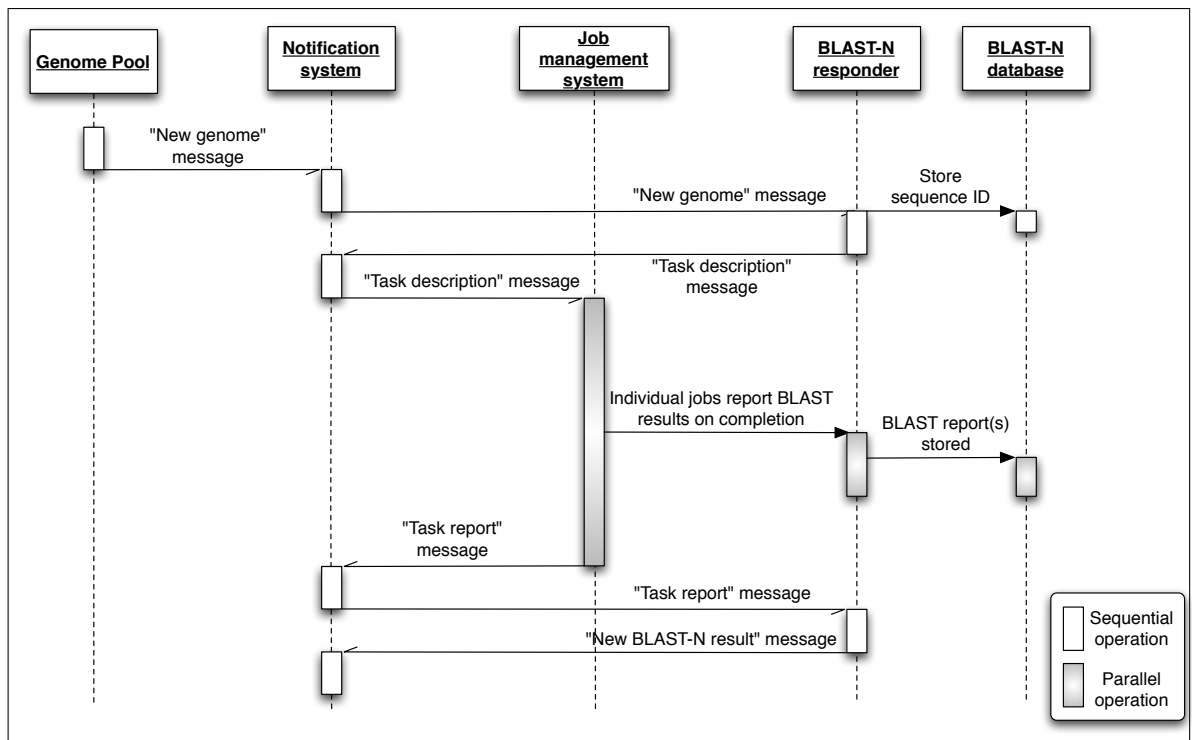


Figure 8.3: Executing a BlastN task in Microbase. 'New genome' messages from the Genome Pool responder are propagated via the notification system to the BlastN responder. The BlastN responder decides what computational work is necessary, and publishes an appropriate 'task description' message which is subsequently delivered to the Microbase job management system. The job management system then executes BlastN jobs on available worker nodes. When a worker node completes a BlastN process, it then parses the resulting text file into an object model and sends this to its parent Web service component. The Web service component then immediately inserts the parsed Blast results into its relational database. Finally, once all jobs have finished executing, a 'task complete' report is published by the job management system and is forwarded to the BlastN responder Web service. The BlastN responder checks the task report for successfully completed jobs and publishes an appropriate 'new BlastN result' message. This message is not currently used, but is stored within the notification system for future extensibility.

Figure 8.4. Instead of persisting result data at the end of each job completion, results are serialised to a temporary file residing on the server. Once all jobs belonging to a task have been completed, the server then performs all the required database insertions.

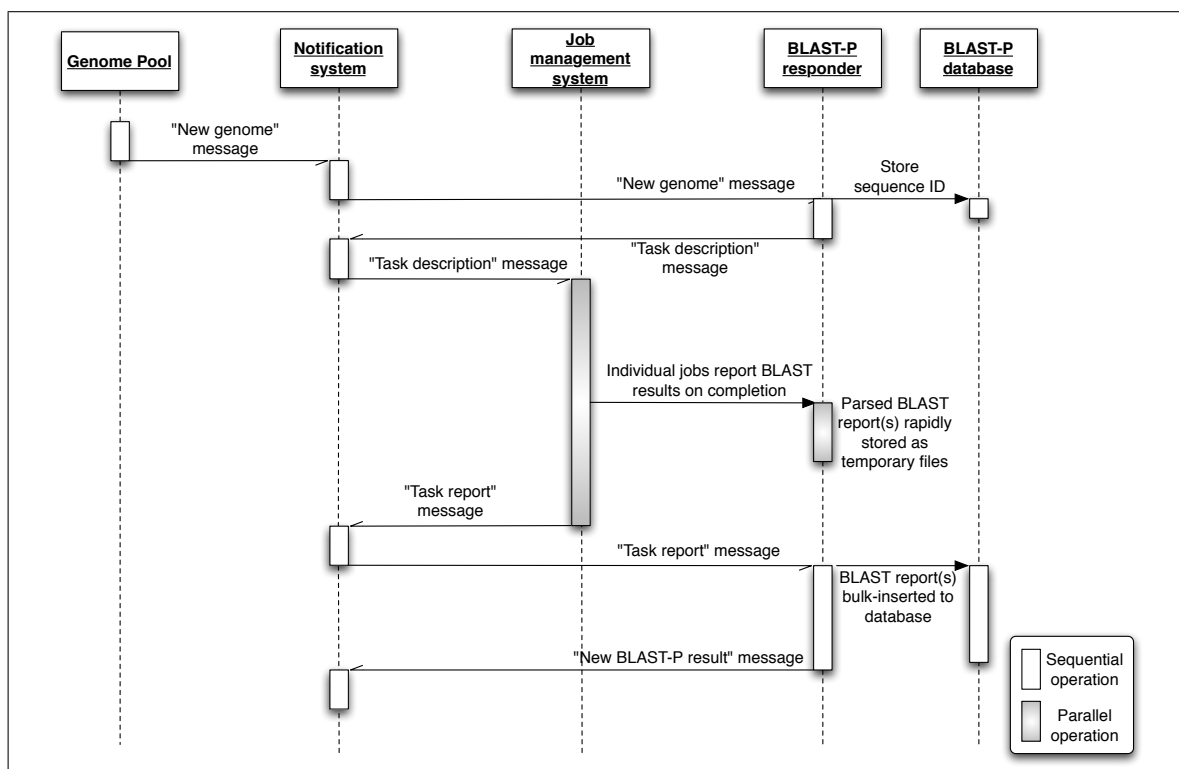


Figure 8.4: Executing a BlastP task in Microbase. The ordering of operations is subtly different to the BlastN responder design, but achieves significant scalability improvements. ‘New genome’ messages from the Genome Pool responder are propagated via the notification system to the BlastP responder. The BlastP responder decides what computational work is necessary, and publishes an appropriate ‘task description’ message which is subsequently delivered to the Microbase job management system. The job management system then executes BlastP jobs on available worker nodes. When a worker node completes a BlastP process, it then parses the resulting text file into an object model and sends this to its parent Web service component. The Web service component writes the object model to a temporary disk file as quickly as possible. Once all jobs have finished executing, a ‘task complete’ report is published by the job management system and is forwarded to the BlastP responder Web service. The BlastP responder performs bulk result insertion operations by reading the Blast reports stored in temporary files into the database. This approach is significantly more efficient since the worker nodes do not block on database operations and can be processing jobs from another task instead (data not shown). Finally, the BlastP responder publishes an appropriate ‘new BlastP result’ message which is stored for by the notification system for future extensibility.

BLAST-P with the NCBI Non-redundant database

Another responder was implemented that executes BlastP with a single large database, rather than comparing sequences against each other in a pairwise fashion. In common with the other Blast responders, this responder also reacts to ‘new genome’ notification messages published by the Genome

Pool. `Blast` searches are performed with each proteome against the NCBI non-redundant protein database [250], which has a compressed size of 2.4GB. The large file transfers required by this responder make it a good use-case for benchmarking the performance of the Microbase resource transfer system with large numbers of worker nodes.

Comparing the sequences of an entire bacterial proteome against the NCBI non-redundant database takes in the order of 10-12 hours on a single modern desktop machine (Intel Core2 duo, 2GB RAM). Therefore, for each 'new genome' event received, `BlastP -nr` responder schedules jobs that analyse a block of 100 proteins each.

8.3.2 AGA Viewer

The `AGA` visualisation tool is a browser-based set of applications for monitoring the progress of analyses, browsing genome sequences and annotations, and visualising pairwise sequence comparisons. The user interface was written using the Google Web Toolkit [125]. The interface draws its data from the public Web service query methods provided by each responder (see Figure 8.5).

The genome browser application displays glyphs representing Coding Sequence (`CDS`) regions and other annotations provided by the Genome Pool responder Web service interface. Genome sequences can be chosen based on searchable properties, such as their accession number, organism name, and so on. It is possible to zoom and pan the view in order to display the required region.

The comparison viewer application displays two parallel genome browser tracks described above, together with similarity information shown as linking regions between the two browser tracks (Figures 8.6 and 8.7). The result is comparable to other visualisation tools. However, unlike existing tools, it is possible to switch between, or select multiple comparison data sets simultaneously to determine if different analysis methods correlate with one another. Comparison data sets are colour-coded in order to differentiate them. Different comparison data is loaded on-demand from the appropriate responders.

8.4 Results

8.4.1 System configuration

The following table shows the hardware specifications of the worker nodes that took part in the benchmarks described in this section. The benchmarks were performed using a cluster of Linux computers

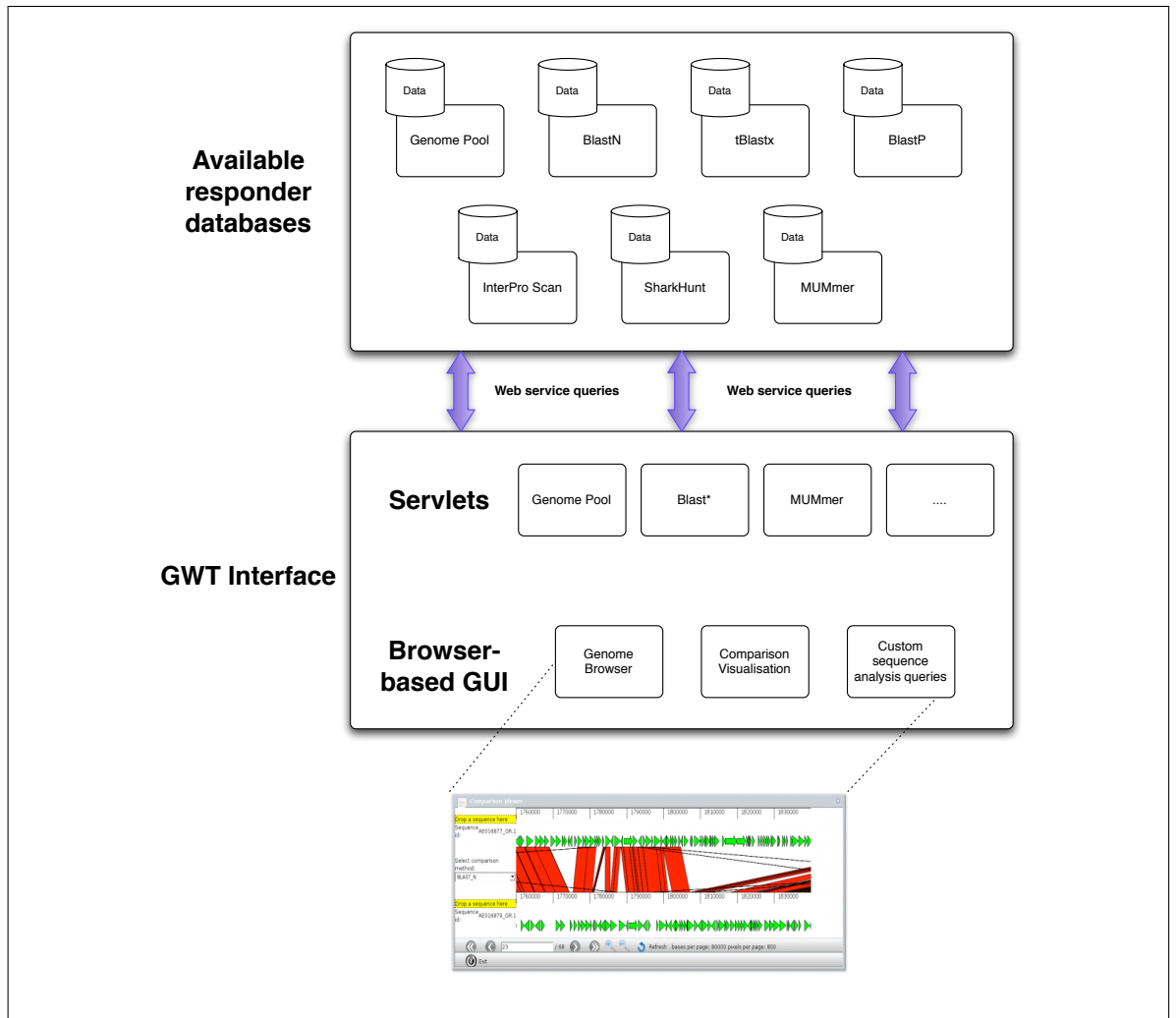


Figure 8.5: Shows the architecture of the **AGA GUI** interface. The user interface is implemented using the **GWT** framework. The **GUI** is entirely separate from Microbase — no contact is made with core Microbase services. Data is retrieved from several responders via standard Web service queries. The structure of the **GWT** interface is divided into two layers: the first layer is a set of server-based **GWT** servlets , one for each responder. The second layer is a set of browser-based applications that communicate with one or more **GWT** servlets to obtain their data.

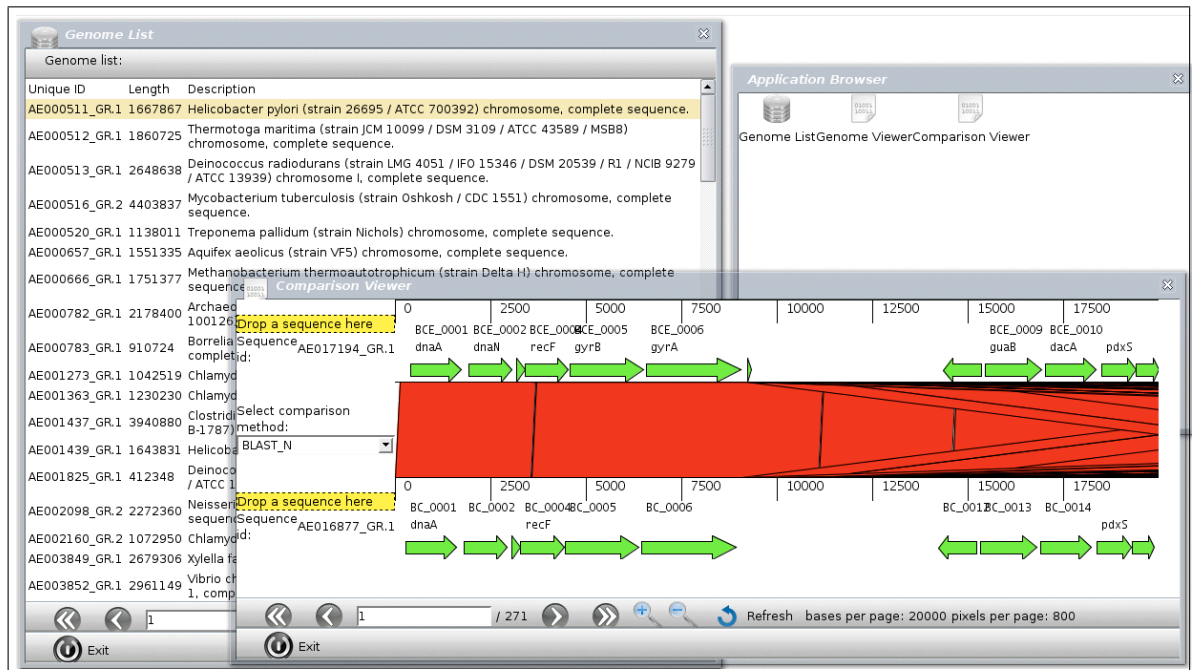


Figure 8.6: The AGA viewer. A list of available genome sequences is obtained via the Genome Pool Web service. A genome comparison window can be opened, and two sequences selected. Genome comparison data is then obtained from appropriate responder Web service interfaces and integrated with the genome information data.

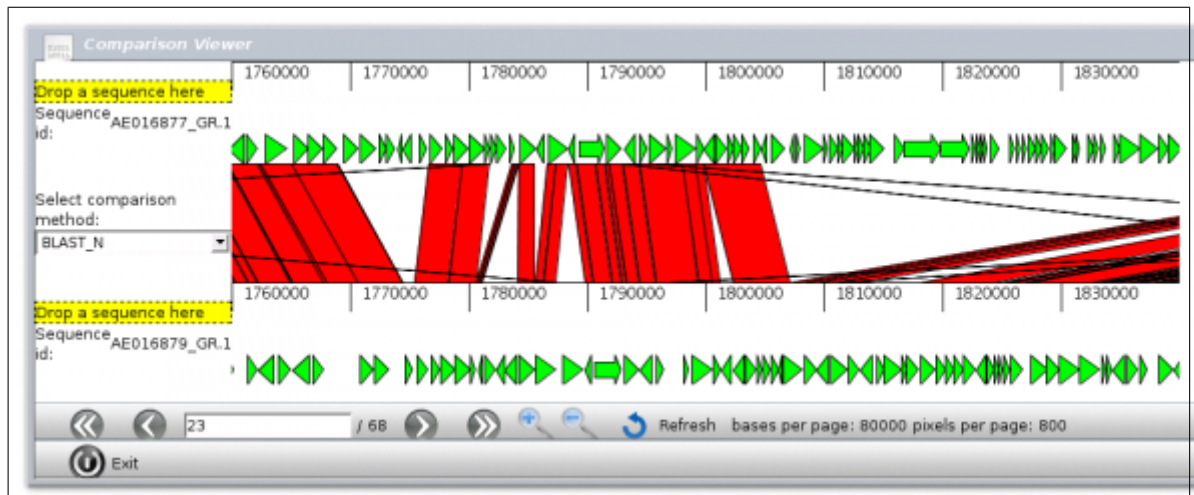


Figure 8.7: A visualisation constructed from the integration of several data sources. The view depicts two bacterial genome sequences. Red bars highlight regions of similarity between the two sequences. The data source providing information for the comparison track can be switched between BlastN and BlastP. The view can be panned left and right as well as zoomed in order to locate features of interest. The region shown suggests that top sequence has several genes not present in the bottom sequence.

at Newcastle University, as well as a set of pay-per-use machines from the Amazon Compute Cloud (Amazon EC2) [1].

	Newcastle desktop PC	Amazon 'small'	Amazon 'medium'
CPU	Intel Core2 duo, 2Ghz	1x 1EC2	2x 2.5EC2
RAM	2GB	1.7GB	1.7GB
Disk space	25GB	160GB	350GB

The CPUs provided by Amazon EC2 are not guaranteed to be of a particular vendor or generation. Instead, a leased machine has a CPU capable of a certain measure of performance measured in 'EC2' units that are a measure of relative performance to other CPUs provided by Amazon. For example, the 'medium' Amazon node has two CPUs, each of which is 2.5 times faster than the single CPU present in the 'small' Amazon node configuration.

The Amazon instances executed custom virtual machine image generated via the following steps:

- An existing public 'barebones' Debian 5.0 image was obtained from <http://alestic.com/> [accessed 2009/10/08].
- Various system configurations were performed, including the creation of a 'microbase' user. A security group was created to permit Azureus BitTorrent traffic to reach the node.
- Java was installed, required by the Microbase compute client.
- Finally, a snapshot of the running instance was taken using the Amazon tools. This snapshot was used for the experiments described in this section.

A small configuration script can be passed to Amazon instances that execute at boot time. A script was written to download the Microbase compute client from a web server and execute it. The compute client was configured to communicate with a Microbase job server Web service located at Newcastle University.

8.4.1.1 BLAST-P NR responder using Amazon EC2 and Newcastle nodes

This experiment is of interest because of its global nature. The experiment involved pre-loading a Microbase/AGA installation with two bacterial genome sequences: *Staphylococcus aureus* USA300_TCH1516, and *Staphylococcus aureus* COL. The BlastP -nr responder divided the computational work into blocks of up to 100 proteins, resulting in 54 jobs. The 2.4GB NCBI non-redundant database Blast database was used for this benchmark. 34 worker nodes at Newcastle University and a further 20

nodes at Amazon’s European data centre were started simultaneously. Each node was configured to execute one job at a time, and to allocate all local CPUs to that job. Therefore in this experiment, each worker node executed exactly one job and both available cores were allocated to the BlastP process. Timing information is summarised in the table below:

Duration (jobs) (mins)	Durations total (mins)	Useful work (mins)	Job processing speedup	Job efficiency (%)	Overall speedup	Overall efficiency (%)
202.38	239.3	4948.4	24.45	45.28%	20.68	38.29%

The parallel portion of the task (job execution) took just over 200 minutes to complete. The entire task, including database insertion operations took 240 minutes. Using 54 worker nodes resulted in an overall task speedup of approximately 21x. The reason for this low efficiency is mostly due to the worker nodes executing a single job, and the time taken to transfer the 2.4GB Blast database to each node. In a more realistic scenario, each node would execute multiple jobs, thereby reducing the impact of the one-off file transfer operation. However, the main purpose of this benchmark was to demonstrate the ability of the resource system to distribute large files to multiple nodes simultaneously.

Figure 8.8 shows a BitTorrent client monitoring the Blast database file as it transferred to the worker nodes. The machine serving the Blast database is connected to the network via a standard 100Mbps network adaptor. However, the total speed for the BitTorrent ‘swarm’ reached a peak of 900Mbps, demonstrating the the worker nodes were transferring data among themselves.

The ‘CloudWatch’¹ facility of Amazon EC2 permits the collection of a number of statistics such as CPU and disk utilisation. At the time of the experiment, network bandwidth monitoring of instances did not appear to function correctly. However, the total amount of bandwidth sent to and received from the Amazon data centre was available.

During the benchmark, 5.82GB of data was transferred to the Amazon data center, costing \$0.58. 1.79GB of traffic was sent back to Newcastle University, costing \$0.30. If the entire 2.4GB Blast database had been transferred to each node via a centralised protocol such as FTP, 48GB would have needed to be transferred to the Amazon data center. This would have cost \$4.80. Therefore, the resource distribution via BitTorrent was 8.25 times cheaper for this benchmark than using a centralised protocol.

CPU analysis of the 20 Amazon worker nodes confirms the length of time taken to download the 2.4GB database. The CPU utilisation graph in Figure 8.9 shows the processor usage of 10 Amazon

¹<http://aws.amazon.com/cloudwatch/> [accessed 2009/09/26]

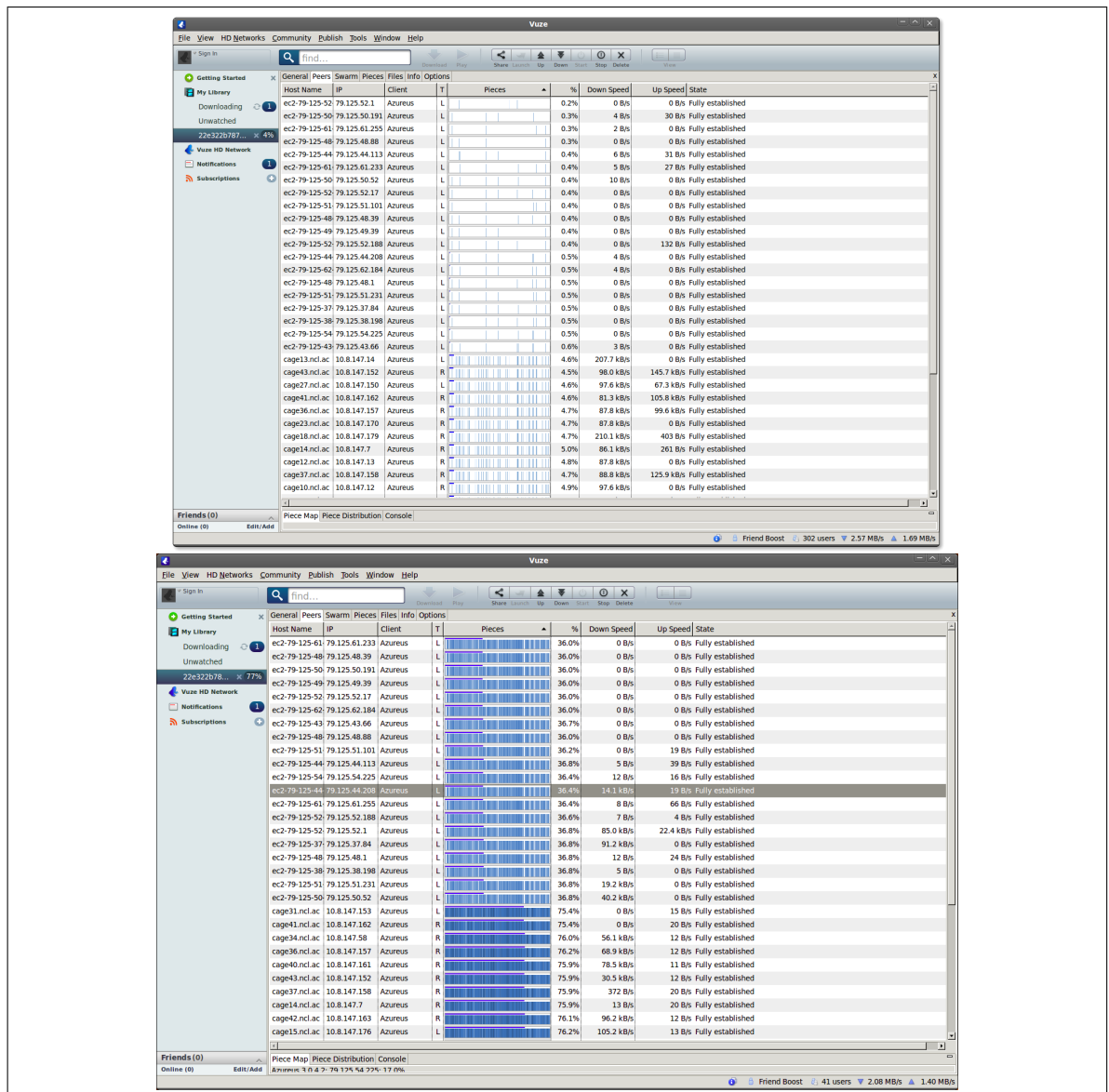


Figure 8.8: Monitoring the progress of a BitTorrent file transfer of a 2.4GB Blast database file using a standard Azureus client [249]. Machines local to the resource archiver node serving the file (hostnames: (cage*.ncl.ac.uk) manage to acquire the file at a much faster rate than remote machines (hostnames: ec2-79-*) since the local network is much faster than the Internet connection. However, as soon as one of the remote 'ec2' machines acquires a file chunk, it is immediately shared with the other 'ec2' machines via Amazon's internal network.

nodes². Processor utilisation is shown to be very low for approximately 25 minutes. This time corresponds to the length of time estimated for a BitTorrent transfer of 2.4GB. After 25 minutes, CPU utilisation rises to 10% for approximately 10 minutes, presumably due to the worker node making a copy of the downloaded file to an isolated job execution temporary directory, and decompressing the file. After this, CPU usage rises to between 20-30% while the machines start the ‘blastall’ process and begins to read the database file. After a short period, CPU usage reaches 100%. Also of interest on this graph is the apparent failure of one of the nodes. The node represented by the purple line appears to download the file correctly, but then appears not to progress, with CPU utilisation remaining at close to 0%. Analysis of the log file produced for that worker node suggests that one of the timeout values for detecting job inactivity had expired, causing the job to be reported as a failure.

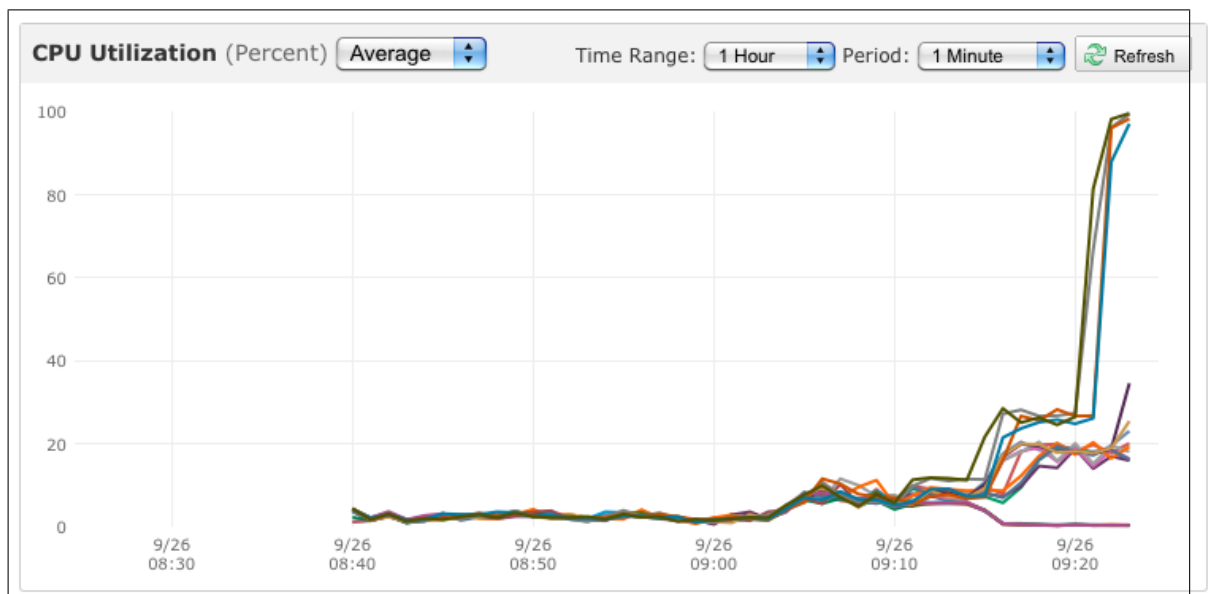


Figure 8.9: CPU usage graph for 10 Amazon nodes for approximately the first 40 minutes of the experiment. The X axis represents time. The Y axis represents CPU utilisation. The Amazon instances used for this experiment have two CPUs, so a value of 100% represents indicates both CPUs are fully utilised. Note the CPU usage of all nodes remains low during the data staging time, rising only once all nodes have downloaded the necessary executable files and Blast 2.4GB database.

While downloading a large file via BitTorrent the CPU utilisation is negligible. Each node is presumably in contact with a significant number other nodes — perhaps 20 to 30 — yet maintaining these connections does not appear place a significant burden on the CPU. This observation provides evidence that the parallelised job caching and result archival operations (as described in Chapter 7) do not have a high impact on concurrently running processes, and are therefore a good approach to reducing the amount CPU time that is ‘wasted’ while waiting for resource transfers to complete.

²the CloudWatch graph is limited to 10 nodes. Another graph was generated of the remaining 10 nodes and showed a very similar layout.

8.4.1.2 BLAST-P Pairwise responder using Amazon EC2

The BLASTP-NR responder was a good test case for Amazon since it highlights the advantages of BitTorrent transfers to a large number of remote nodes. The BLASTP-pairwise responder used in this benchmark poses a different challenge to the resource transfer system. Instead of one extremely large database file, each proteome is compared with each other proteome. Each job will therefore require two FASTA -formatted files as input. Since an exhaustive pairwise analysis is performed, more than one worker node will require the same files at some point. However, this is likely to be at different times. Also, not all worker nodes will require all files.

The benchmark was performed using 40 bacterial sequences comprising the *Staphylococcus* and *Bacillus* genome sequences available from the GenBank FTP site.

The Genome Pool database was populated using 10 worker nodes local to Newcastle since the Genome Pool responder cannot yet function in the Cloud. The Newcastle nodes were then shut down, and 20 'medium' Amazon instances were started with a job queue size of 2. Therefore, there were a maximum of 80 jobs active at any one time: 20 worker nodes, with 2 CPUs per node with 2 jobs in a queue (one processing, and one downloading/installing).

The CPU and disk utilisation graphs from CloudWatch are shown in Figures [8.10](#), [8.11](#), and [8.12](#).

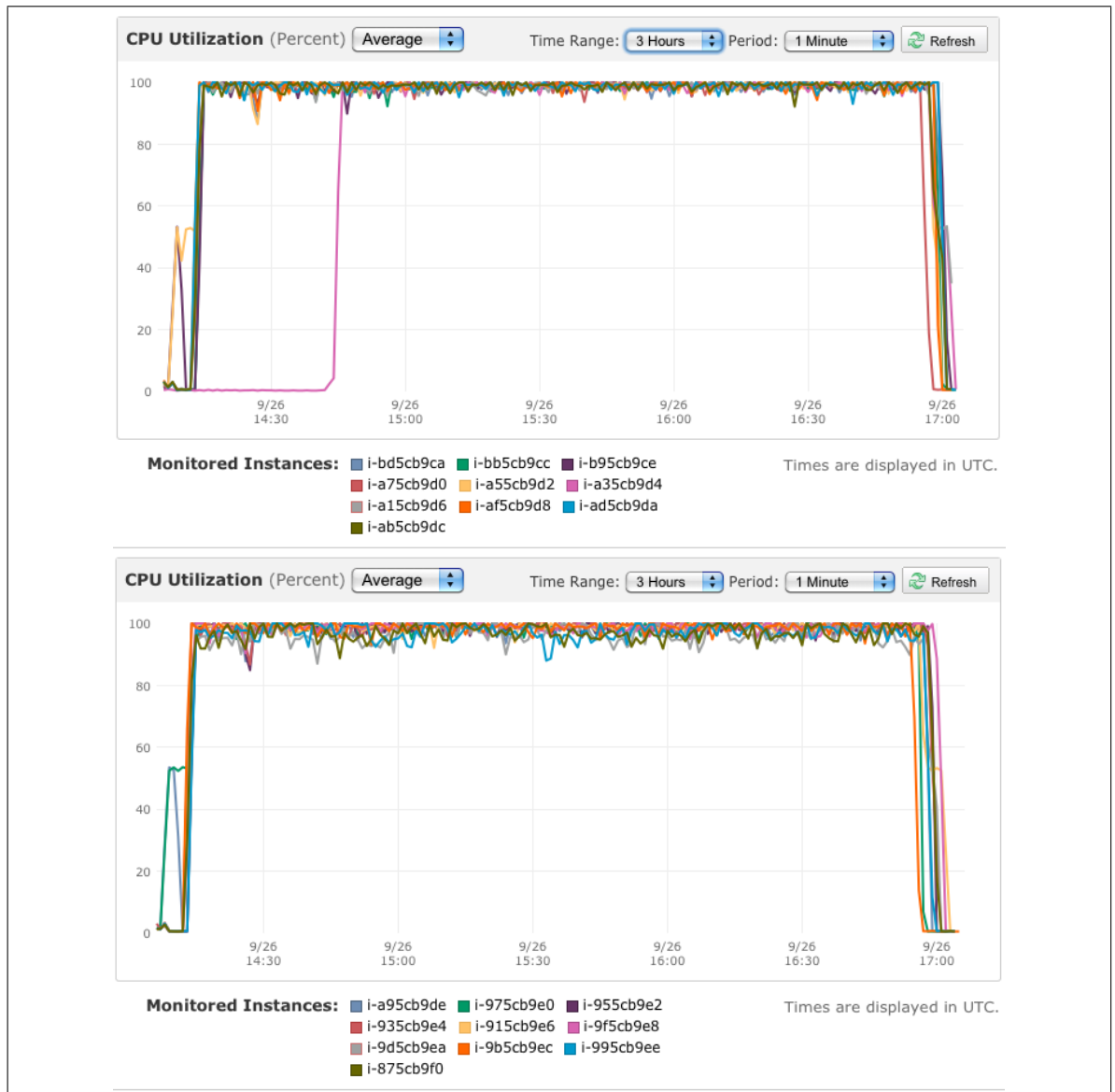


Figure 8.10: CPU usage for 20 Amazon EC2 nodes over the duration of the experiment. It is interesting to note that CPU utilisation remains consistently above 90% even though each node is executing multiple jobs that require different data files. The Microbase compute client is successfully preparing the ‘next’ job while the current job is processing. All worker nodes were started at the same time, however, node ‘ia-35cb9db’ initially failed to connect to the Microbase job server running at Newcastle. Following a manual restart, the node worked correctly.



Figure 8.11: Disk reads for 20 Amazon EC2 nodes over the duration of the experiment. Disk reads were surprisingly light, indicating that once obtained via BitTorrent network transfer, most files were cached in memory.

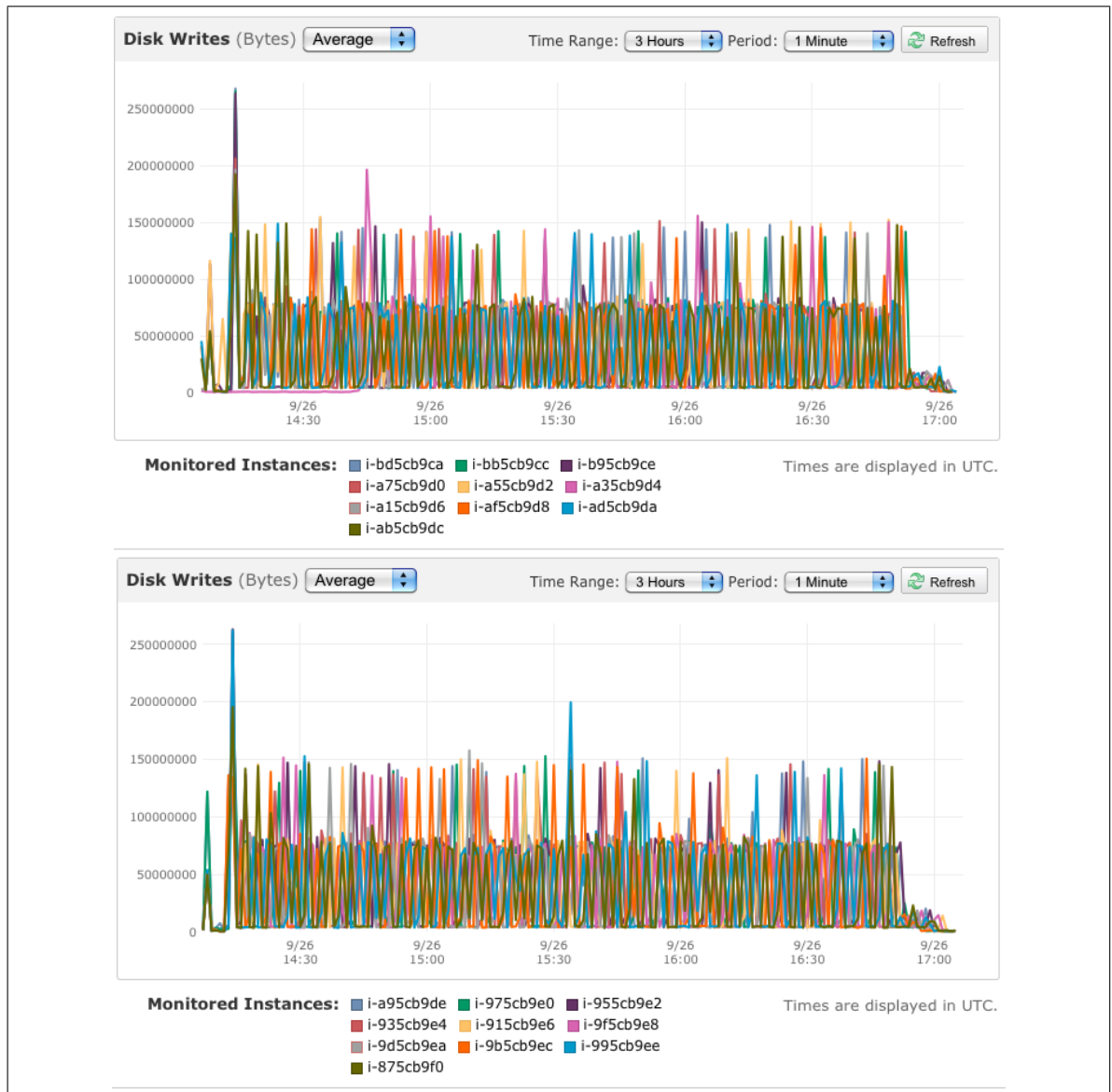


Figure 8.12: Disk writes for 20 Amazon EC2 nodes over the duration of the experiment. A large number of disk writes indicate a) many output files being written as the 1600 jobs were completed, and b) FASTA sequence files being written to disk as they are transferred among the nodes.

The CPU utilisation graphs show that the Amazon nodes are almost continuously at 100% utilisation (Figure 8.10), indicating that the job caching mechanism used by the compute client was successfully preparing jobs ahead of time to ensure uninterrupted computation. There is a node that appears to be continuously idle: 'i-a35cb9d4'. On inspection of the log files, it appears that the compute client failed early in its initialisation phase. On manually restarting the Microbase client on node 'i-a35cb9d4', the node correctly contacted the Web service and proceeded as normal. This is represented on the graph as the pink line that jumps to 100% much later than the other nodes. The failure of the node did not affect the outcome of the experiment in terms of the expected number of database results. In fact, it demonstrates the failure handling mechanisms that regularly deal with

individual worker node failures. Although the Amazon nodes tend to be reliable, desktop computers at Newcastle regularly fail without warning, either due to hardware problems or user interruptions.

The larger number of jobs in this run means that job life is much shorter than with the BLAST-NR jobs. The disk statistics reflect this, with much greater fluctuations. Disk usage typically occurs at the beginning of a BLAST-P job when reading the sequences and at the end, while writing the alignment. Other possible disk access include: preparing the temporary directory for the next job, while the current job is still executing; uploading the previous job's alignment file to the Microbase resource system; and sharing sequence data files with other nodes, again via BitTorrent. Disk access tends to be mostly write-access, with the only significant disk reads occurring at the start of job processing. This pattern is consistent with worker nodes having ample RAM in which files are cached. Disk reads are infrequent because the worker nodes have enough memory to almost entirely cache software and data files. Disk writes are fairly continuous since output files are generated from the BLAST-P process, and new sequence files will be arriving via BitTorrent.

Regarding server capacity for the distributed job processing phase, the current server configuration appears to be adequate. The number of completed jobs waiting to have their files archived is a good measure of server stress. With 20 worker nodes executing 40 jobs concurrently, there were between 10 and 20 jobs with outstanding archive operations. The system should therefore be able to cope with much greater numbers of worker nodes.

The parallel job processing of 40 proteomes executed on 20 nodes, with a combined total of 40 CPUs completed expended 6521 'useful' CPU minutes in just over 3 hours of wall clock time. The parallel phase therefore completed with a 36x speedup. However, a very slow database configuration prevented a reasonable speedup from being attained. It took a further 18 hours to populate the structured database with the Blast hits, resulting in an overall speedup of just 5x.

8.4.2 Benchmarking an entire pipeline of responders

A final set of benchmarks show how the system scales when a number of responders are executed together as a pipeline, rather than individually. For this set of benchmarks, 18 *Staphylococcus* genome sequences were used. The BlastN responder has been omitted from these tests, since it did not scale beyond 10 nodes.

The following table shows the speedups achieved when executing the pipeline on various numbers of worker nodes. In this case, all worker nodes are local to Newcastle University.

No. workers	Job processing duration (mins)	Total responder duration (mins)	Useful job time (mins)	Speedup of job processing	Efficiency of job processing (%)	Overall speedup	Overall efficiency (%)
36	1720.93	1727.35	54659.89	31.76	88.23%	31.64	87.90%
71	1000.7	1024.02	53432.42	53.4	75.20%	52.18	73.49%
75	920.82	935.52	53636.55	58.25	77.67%	57.33	76.44%

A faster database server machine was used for this benchmark that obtained approximately 2.5x faster raw row insert performance compared to the database machine used in the previous experiment. The Blast responder Web services were also installed to the same physical machine as the database software, eliminating network transfers between client and server. The results show that the database insertion phase did not have a large effect on the overall speedup value, although at roughly 9 million rows each, BlastP -pairwise and BlastP -nr responders had much smaller result sets than those generated by the benchmark in Section 8.4.1.2. Therefore, further work with larger data sets would be required to confirm that the database bottleneck issue has been resolved.

During this experiment it was noticed that before the end of the parallel job processing stage, the database was being populated by both BlastP -pairwise and BlastP -nr responder results from completed tasks. When responders are in ‘data insertion mode’, even a single bulk insertion thread put significant load on the database server. An initial concern from the previous experiments (Section 8.4.1.2) had been that database insertion took far too long in relation to the parallelised job processing. However, it appears that for at least some of the time, there is an overlap between parallel job processing and result insertion. Because the ‘serial’ step of inserting the result from each job is being started before all tasks have completed, it appears that the best possible use is being made of all hardware.

Two conclusions can be drawn. When scheduling jobs, the Microbase job scheduler should attempt to prioritise tasks with only a small number of remaining incomplete jobs. Prioritising those jobs would allow tasks to be completed sooner, and therefore database insertion operations to begin sooner. Secondly, splitting large sets of jobs into multiple tasks has proven to make better use of server-based resources, again because data insertion operations are started earlier. The earlier the database insertion process commence, the shorter time is required to complete the entire workload of a responder. Bulk-insertion of large results sets into a centralised database can only feasibly be performed by serialised transactions, so the overall performance of the system can be increased if as many results as possible are inserted while distributed jobs are being processed.

8.5 Conclusions

[AGA](#) has demonstrated the ability to construct bioinformatics pipelines using the Microbase system. It also provides a useful tool for biologists in the form of a browser-based visualisation tool. A large data set has been constructed executing multiple unmodified bioinformatics applications operating within a distributed processing framework. The data set can be kept up-to-date automatically by periodically scanning for newly-released sequence data. The data set may be browsed via a [GUI](#) interface, or queried programmatically via a number of Web service interfaces.

Responders are extensions of notification system `push subscribers`. Processing pipelines composed of responders are inherently extensible. For example, it would be possible to attach new responders to react to the events from the Genome Pool. In this case, the entire history of ‘new genome’ events would be passed to a newly attached responder, enabling it to bring its result data up-to-date. In fact, the pipeline could be extended at any point connected with a notification event.

8.5.1 Responder development experience and data flow

Whilst variation in development occurred throughout the duration of this project, all AGA responders follow the same general pattern as discussed in Chapter 6: the server-based component is responsible for notification message handling, task splitting and management of a structured data store, while the worker nodes are responsible for operations that can be parallelised. With each AGA responder there are minor differences with the way in which a compute job communicates with its management Web service, that have considerable consequences for scalability. This section presents some refinements to the design pattern described in Chapter 6 that take into account our real-world experiences of implementing an analysis pipeline using Microbase.

The Genome Pool responder compute job implementation communicates directly with its parent [SQL](#) database. As such, we found that its scalability was impaired as a result of the database becoming a bottleneck. The direct connection to the database also limited the potential of this responder to run in a Cloud environment since the relational database was located behind a firewall.

The `BlastN` responder implementation offered a slight improvement in design. Instead of communicating directly with the relational database, compute jobs construct an object model representation of analysis result data and transmit this to the Web service interface of the responder. The Web service implementation then immediately inserts the results. The `BlastN` database schema is relatively simple in comparison to the Genome Pool database. Even so, scalability was poor, with around 10

worker nodes able to saturate the database server. While monitoring a running system, it was discovered that the CPUs of worker nodes remained idle for long periods while they awaited the completion of database transactions.

The `BlastP -pairwise` and `Blast -nr` responders improved on earlier designs with by implementing a slightly different approach. Instead of inserting data immediately upon completion, the Web service component simply serialises the result data object model to a temporary file as shown in Figure 8.4. This operation is fast since no database transactions need to be executed. The worker node can also immediately start processing the next job. On completion of all the jobs in a task, the Web serviceresponder component bulk-inserts the results from all successfully-completed jobs. This approach offers a number of advantages. The parallel execution phase of a responder is vastly more scalable than the Genome Pool and `BlastN` responders. If computational tasks are suitably split into a number of `Microbase tasks`, then the database insertion stage also runs in parallel to job processing, for at least some of the time. Evidence for parallel job processing and database insertion was presented in the whole-pipeline benchmark results, where database insertions were observed to proceed concurrently with job executions towards the end of the job processing stage.

8.5.2 Future work

The `Microbase` notification system is used for responder co-ordination operations within AGA. In future, it would be possible to make use of event notifications for high-level application purposes. For instance, one possible extension would be to allow users to register an interest in the completion of particular analysis type or data type, and be notified when a significant event occurs. The AGA browser application could be extended to display such event notifications. This kind of functionality could be used by biologists who might be interested in a specific organism or gene name. When an event involving a specified gene occurs, such as a significant `Blast` hit, the user could be informed immediately instead of having to periodically browse or query the result data themselves.

Chapter 9

Discussion and conclusions

9.1 The Microbase System

9.1.1 Architecture choices

Microbase is a robust and scalable system that permits the parallel execution of numerous bioinformatics analysis tools in a flexible way. A variety of distributed systems architectures have been utilised in the Microbase framework. When viewed as a whole, Microbase appears as a collection of interconnected distributed components, providing a broad range of services. Each service provides a particular unit of functionality, whether this is ‘core’ functionality such as task scheduling or file management, or application domain-specific functionality. Within a functional unit, interactions are typically client-server based, since this is a practical approach for a) collecting results in a central, coherent location after distributed processing and; b) servicing data requests and queries from 3rd party clients. Finally, Microbase makes use of a decentralised [P2P](#) architecture for large-scale file transfers among participating computers.

The choice of architecture for different areas of Microbase contribute to its overall properties and suitability for meeting the requirements described in [Chapter 3](#). For instance, the use of Service Oriented Architectures ([SOAs](#)) facilitate straightforward access to application data via standard Simple Object Access Protocol ([SOAP](#)) [[59](#)]. Although Web services introduce an element of centralisation, bottlenecks may be overcome with standard techniques such as service mirroring or database clustering [[333](#), [166](#), [320](#)]. In addition, the server-resident portions of `responders` are movable by a system administrator.

A collection of distributed components termed `responders` form the core Microbase system. Most

of the time responders are independent of each-other, with occasional synchronisation operations being performed via the Microbase notification system. Responders may also optionally communicate directly with each other via their Web service query interface. The decision to distribute the Microbase core functionality as well as application workflow steps across a number of Web services was made to facilitate flexible deployment options. For example, a system administrator is able to choose freely which components are installed to which physical server hardware. It is possible to install multiple instances of some responder services, such as the resource storage system, in order to support larger numbers of worker nodes.

Microbase has a partly centralised, partly P2P and part-mobile Grid system. It could be argued that there is no need for a central server of any kind; that all data storage, querying and logging could be stored in an entirely distributed environment. Several such P2P computational Grid systems exist. However, the focus of Microbase is to provide an environment amenable to all aspects of large-scale data analyses, covering not just the computational phase, but also the data querying and integration phases associated with an analysis pipeline. For instance, SQL databases are a proven means of large-scale data storage and retrieval, while Web service technologies a convenient means of providing programatic and well-defined access to such data stores [244]. These technologies are essentially centralised in nature, but are widely-used, well-understood industry standards that work well over Internet connections [52].

Long-term analysis pipelines require an operating environment that is extendable with the addition of both new data and new software, is tolerant of failures, and is maintainable through patchable software components. The emphasis in the development of Microbase was therefore not only on computational efficiency of the immediate-term, but also on the flexibility of the system as a whole over long periods of time. Also important was the issue of responder development, which must be straightforward in order to encourage uptake and use of the system by the community. Finally, the practical issues associated with the availability of non-dedicated, ‘donated’ hardware resources at university campus, as well as transient hardware rented from a commodity ‘cloud computing’ provider was addressed.

The Microbase system was developed with these concerns in mind. The approach provided by Microbase offers a good compromise between the requirement for well-defined, accessible data sources, and that for dynamic, flexible and scalable data distribution. Centralised server-based components have been used where reliable access to data via complex queries is required. Where significantly large amounts of computational power are available, but with sporadic availability and reliability, mobile agent approaches have been used to mitigate the effects of node failure on the system. Where

file-level access is required by a large group of sporadically-available nodes, a P2P distribution network has been used. This mix of technologies and the varying types of available hardware have been used to its best potential. In addition, in the campus-style environments in which Microbase was designed to run, there will always be a need for a central server in the system since computational results and logs must be stored reliably. In an environment such as Newcastle University most, if not all cluster machines are switched off at various times of the year, and are frequently being powered off at night in an effort to save energy. Therefore, cluster nodes cannot be relied upon to store information permanently.

Types of functionality that are best provided by centralised infrastructure are hosted on dedicated servers. In order to minimise the risk of these services becoming bottlenecks for the rest of the system the server-based components that are subjected to high loads are distributable over multiple server machines. The best example of this approach is the resource system, where it is possible for multiple archiver nodes to pool their network bandwidth and disk capacity.

9.1.2 Scalability

In this project it has been demonstrated that by dividing a scientific workflow into multiple modular stages, applications can be flexibly upgraded and application extensibility is easily achieved. This work has also shown that it is possible to extend the pipeline indefinitely with the addition of new ‘responder’ components, which may be located on their own physical hardware. However, with these advantages comes increased latency between the analysis stages. Following the processing of a single data unit, such as an individual genome sequence, through all the required processing stages from start to finish results in a processing time that is far slower than would be achieved by a custom batch script running on a single machine. However, given enough simultaneous work units and a suitably large number of worker nodes, large amounts of hardware can be used in parallel to achieve vast performance increases.

It has also been shown that the number of computers used as worker nodes can be scaled effectively as long as there is adequate server capacity. The ‘rate limiting step’ appears to be the result archival stage, in which many worker nodes require result data to be reliably stored. Here, however, the server-side support can also be extended extensively through the addition of potentially hundreds of ‘archiver nodes’. The addition of archiver nodes requires no reconfiguration other than the deployment of an additional instance of a Web service to a new computer. This form of expansion is also the preferred means of disk-space expansion, since in addition to additional storage capacity, net-

work and CPU resources can also be used. Again, the disadvantage to this approach is the increased latency seen during the peer resolution phase of torrent acquisition.

9.1.3 Responder development framework

Microbase is a distributed computing framework that provides a number of services, such as publish-subscribe event notification, [P2P](#) file transfers and job scheduling abilities. Together these form a novel platform on which distributed processing pipelines can be built. However, building applications directly using Microbase Web services is complex, since it requires knowledge of several services and how they interoperate. This complexity was the motivation for the `responder` design pattern and associated abstraction layer described in [Chapter 6](#).

The responder development architecture assists with a number of challenges commonly associated with distributed application development such as difficulties in using Grid technologies, difficulties in deployment and maintenance of application components, and difficulties in structuring applications to make the most of available hardware and to avoid implementations that may result in performance bottlenecks [[153](#), [265](#), [255](#)].

For the application developer, the responder development framework provides an [API](#) that is decoupled from Microbase itself. Microbase services and implementation details are hidden behind a layer of abstraction. For example, the application developer would construct a compute job by starting with a standard Java data bean. Bean properties would then be annotated with metadata that informs Microbase whether a particular property is an input or output from the job. At no point is the developer exposed to BitTorrent or resource system lookup queries.

The responder design pattern also provides clearly defined guidelines about which component should host particular types of functionality: management and query functionality should be placed within the Web service component, while computationally-intensive operations should be placed into the compute job component. The rigid structure of a responder means that applications can be designed for scalability since the developer knows which parts of the system will take advantage of parallelism and [P2P](#) data transfers, and which parts of the system will provide management of centralised structured storage.

Third party developers will also be able understand the structure of a responder more easily because it follows a common pattern, thereby facilitating re-usable components. For example, they will be able to easily adapt an existing responder to fit their pipeline by modifying only the Web service component to respond to event topic names that are present within their pipeline.

For system administrators, responders provide modular units of functionality that can be deployed or moved to another application container or server at will. New responders can be added to a Microbase installation by deploying them to an application container - existing responders and Microbase services do not need to be shut down. Because server and client components are contained in separate projects, and the directory structure and project descriptions (Maven project object models) are machine-interpretable, tool support for deployment management can be provided (see Appendix A). The installer application aids system administrators by ensuring that Web service components of responders are deployed to an application container of their choosing. It also parses project dependencies of compute job components, including `jar` libraries, command line software packages, and data files and ensures that these files are published to the Microbase resource system so that they are available to install on worker nodes.

9.1.4 Comparisons with other frameworks

To my knowledge there are no other systems that are similar to Microbase in terms of complete functionality and overall architecture. However, a number of other systems have been described that have been developed to tackle high-throughput data analysis that share similarities at the component or architectural level.

The Microbase job management component provides functionality that is broadly similar to Condor. Both the Microbase job manager and Condor [194] maintain a queue of job descriptions containing details of the programs and data to be used. However, Condor uses a ‘push’ model, where jobs are actively sent to registered worker nodes. In contrast, the Microbase compute client pulls work from a remote server. The ‘pull’ model reduces the amount of system configuration that is necessary, since the server does not need to know the location of every worker node. On startup, a Microbase compute client will announce the system configuration and hardware specifications of the worker node. The Microbase job server uses these details to find suitable jobs that fit the capabilities of the worker node. Condor has a similar ‘ClassAd’ system.

Although Condor has been used in a number of large deployments with hundreds of worker nodes it is not ideally suited to the kinds of data-intensive work that pipelines such as AGA perform [302]. The reliance on a centralised infrastructure for data distribution is the limiting factor in terms of scalability for applications requiring large data files [303]. For example, attempting to transfer large files to even a modest number of local worker nodes using Condor quickly results in the file server becoming overloaded.

While Condor has been used in Grid applications spanning multiple sites, it is most often used to aggregate resources local to an institution, while a higher-level framework handles cross-site communication and coordination operations [106].

BioAgent [215] applies mobile agents to bioinformatics data processing. Different agents in a BioAgent system perform different types of processing. In some respects the architecture of BioAgent resembles the responder architecture in Microbase. Microbase is different to this because although Microbase compute jobs are fully mobile, they are always tethered to the Web service component of a responder. An entirely agent-oriented approach would not be the most optimum solution for types of computational task Microbase is targeted for. Pipelines developed with Microbase are likely to have large, typically immobile dependencies such as a multi-gigabyte Relational Database Management System (RDBMS) attached to them. Having the Web service responder components local to the database storage which is faster than having to remotely connect via a network, which is the case for a mobile agent. For Microbase, combining a semi-permanent but nonetheless moveable server component with a highly mobile lightweight ‘job’ component better reflects our intended application use cases. Moreover, making this architectural split encourages better use of available hardware resources; worker nodes execute in isolation on discrete units of highly computationally intensive work, while highly capable server machines are used for the structured storage and complex querying of large datasets.

The system presented by Elmroth et al. [85] integrates a workflow editor with Grid services. Their system defines a separation between workflow processing and performing computational tasks. The workflow components present in the system by Elmroth et al. are similar in purpose to the Microbase notification system and responders’ Web service components. The motivations principles behind the Grid toolkit appear to be similar to those of Microbase. Elmroth et al. argue that the definition of a workflow should be decoupled from the way in which it is enacted, and that domain-specific functionality should be added via plugins. However, the emphasis in [85] appears to be centered around the workflow, and in particular how to import workflows from different representation languages in order to determine the order in which to execute Gridtasks. In Microbase, there is no specific workflow as such, but the notification system and responders with appropriate subscription interests provide equivalent functionality.

Microbase currently makes use of a P2P architecture for the sole purpose of transferring bulk data items. A number of other works use P2P for resource discovery and matching of compute providers with compute consumers. Cao et al. [45] have demonstrated such a system that works across administrative domains. A small number of machines on different private addressing schemes were linked

as a P2P Grid, with messages successfully traversing routers.

For data bulk data transfers, Microbase utilises the BitTorrent protocol. Other P2P protocols, such as Gnutella [103] were also investigated. However BitTorrent was found to best suit our needs because peers involved in a transfer started sharing content with each other long before they had a complete copy themselves. Although the Limewire client [198] was observed to be capable of downloading a file from multiple sources simultaneously, this only occurred once each source had a complete copy of the file (data not shown). Nodes with partially-complete copies of the file would not share to other peers. BitTorrent therefore has the clear advantage for distributing large data items to multiple remote worker nodes via a ‘slow’ Internet connection, since as soon as one chunk has been transferred to one of the remote hosts, all hosts at the remote site will effectively have that chunk soon afterwards (see Chapter 8). In some respects, the Gnutella protocol, and in particular the Limewire implementation are more advanced than BitTorrent. For example, Limewire supports advanced decentralised search capabilities that can make use of keywords, file types and other properties such as file size. In Microbase, these operations are performed in a centralised way via a Web service. Distributed resource lookup for Microbase is a feature that is worth investigating as an area for future work.

Machida et al. recently proposed an approach that overlaps the data staging and execution phases of job execution [204]. Machida et al. state that most current Grid systems perform ‘simple staging’, where data is stored centrally and is staged to each machine in turn which often results in high data transfer overheads and data starvation of worker nodes. Their system implements a file replication system that works in a similar fashion to the Microbase resource storage system. A central service handles lightweight requests for files and matches peers with each other, while the bulk data transfers are handled in a decentralised manner. Their approach utilises an application-level multicast protocol that takes advantage of the routing features of modern network hardware. A machine transmitting a file can essentially transfer the data to any number of peers at an $O(1)$ cost for machines connected to the same router. Since no complex P2P connections need to be maintained, this approach is potentially more efficient than BitTorrent transfers *if* all machines require the same file at exactly the same time. However, traffic between multiple geographic sites is HTTP-based. Therefore, the entire file must be completely transferred to the remote site before it can start to be distributed to remote worker nodes; in effect, the file must be transferred twice. In contrast, BitTorrent provides a much more efficient distribution method in this case since worker nodes at the remote site start to receive the file as soon as the first file chunk arrives, as shown by the AGA use case.

9.1.4.1 Programming models

In terms of design paradigms and programmer toolkits, the Microbase responder development framework compares favourably with similar abstractions for other systems. Having a design pattern or abstraction layer sitting above core services typically reduces application development time by guiding developers in good programming practise and masking underlying complexities of distributed systems [90].

The JaSkel project [90] provides a set of abstract Java classes for writing multi-threaded parallel applications. The classes developed by Ferreira et al. provide a set of templates with a range of hooks into which application developers insert their application-specific code. The templates are not suitable for *every* situation, but makes application development much easier when the programming problem can be made to fit the design pattern. The Microbase responder architecture is very similar in its approach in that classes for forming basic Web service and compute job components of a responder are provided. The Web service abstract class provides programmer hooks for implementing notification message handling, and provides assistance with message serialisation and publication. Initial registration assistance is also provided that includes automated registration of the responder with the notification system. Another abstract class is provided for the implementation of compute jobs. A hook with a well-defined contract permits the developer to place computationally-intensive operations in the appropriate place. Input and output resource file requirements of the job are facilitated through a set of standard Java bean properties, extended via Microbase annotations.

MapReduce [69] is a programming paradigm developed by Google for the parallelisation and distribution of large computational tasks. The MapReduce design pattern consists of two stages. In the ‘map’ phase, a large input data set is split into multiple chunks. A distributed set of processes then perform computationally intensive operations over each chunk in parallel, resulting in a list of outputs associated with each input. These operations must be independent of one another in order to achieve a high scalability. The output of each . The ‘reduce’ phase, consists of a set of distributed processes that combine the separate outputs from the ‘map’ operation to form a coherent result.

For example, a parallelised search application over a large document might first split the large document into sub-documents. the ‘map’ function for each sub-document would count the number of occurrences of the specified regular expression. For each match it would emit an output value. The ‘reduce’ function would then iterate over each output of the ‘map’ function, forming a running total of matches for a particular sub-document. The advantage of this approach is that the task splitting and reduction operations can be recursive, for example a sub-document could in turn be split into

sub-sub-documents.

An application of the MapReduce programming model to bioinformatics is the recently published CloudBLAST system [209]. A large list of sequences in FASTA format is split into multiple chunks. For the distributed ‘map’ stage, the `Blast` application is executed over each chunk. The result of each execution is then merged back into a single file. The ‘reduce’ phase is not used in CloudBLAST, but Matsunaga et al. suggest that a ‘reduce’ phase could be used to filter or classify the results in some way. Matsunaga et al. also demonstrate a multi-site system where clusters at two universities are connected via a Virtual Private Network (VPN). However, the system still requires a manual staging step in that necessary data and applications must be distributed in a virtual machine image to the remote location.

For bioinformatics applications such as `Blast` or other applications that follow the model of input file → processing → text output, it is difficult to see the advantage of the MapReduce approach over other methods if the results must always be parsed back to a structured database for stringent consistency and completeness checking via RDBMS constraints. If the result data is to be exposed for browsing purposes, or for programmatic querying via Web services and consistency is to be maintained, then a central database or database cluster is the most convenient means of achieving this. The database insertion stage will be the rate-limiting step since it is necessarily centralised in nature. On the other hand, if no rich database structure is required and a storage solution such as BigTable [51] is sufficient, then the MapReduce technique offers an elegant solution that is inherently scalable.

It would be possible to implement MapReduce-style computation using Microbase. Instead of a single compute job implementation per responder, two such compute job components would be required: one for the ‘map’ stage, the second for the ‘reduce’ stage. The Web service responder component would schedule a pair of these jobs for each block of computational work. Through the notification system, the MapReduce-style responders could be integrated with multiple other MapReduce responders, or indeed responders written using other parallel programming paradigms.

9.2 Use cases

A number of projects are using Microbase to provide access to computational resources. These projects are briefly introduced in the following subsections. These use-cases are important for a number of reasons. They demonstrate the utility of Microbase, and provide valuable insight into its

usefulness and limitations, and have therefore provided a measure of whether the original requirements (see Chapter 3) were sufficient. The use-cases have also demonstrated the re-use of responders. For example, the ‘genome pool’ responder developed in collaboration with Nakjang [230] has been subsequently re-used without modification for the metaSHARK parallelisation project. Two of use-cases have also contributed directly and indirectly to AGA. Although not part of the AGA analysis pipeline, metaSHARK results are accessible from the metaSHARK responder query Web service. Support has been added to the AGA genome browser interface to display metaSHARK results alongside annotations derived from the AGA pipeline.

Perhaps most importantly these use cases have demonstrated the accessibility of the responder development framework to bioinformaticians with some experience with programming, but whose primary experience is not that of software development.

9.2.1 AGA

The AGA analysis pipeline described in Chapter 8 has shown that it is possible to split up several large-scale bioinformatics computational tasks for distribution to many worker nodes. The individual results from each job were then successfully recombined into a coherent set of independent databases. Several such workloads were completed, with each task type having its own individual database. The set of result databases were then successfully integrated for use by a Web-based genome browser and comparison visualisation tool.

AGA is not so much a tightly-integrated pipeline as a set of independent modules that happen to react to particular types of event. The only stipulation is that a responder must be able to interpret the notification messages that it receives. The development of AGA demonstrated the modularity and flexibility of responders. A prime example is the replacement of the genome pool responder.

AGA is a proof-of-concept analysis pipeline. AGA has shown that Microbase can execute an event-driven set of processes. As new data arrives, it is forwarded to interested responders, which update appropriate data sets incrementally. AGA has also been used to demonstrate that new responders can be added to an existing pipeline without affecting existing data sets.

AGA viewer

The AGA viewer is not part of the processing pipeline. It is a separate project that has demonstrated the ability to re-integrate the data generated by several distinct responders. Because the AGA viewer

application queries the Web service components of responders directly, the data available to the viewer is updated the moment new jobs completed. Whilst the AGA viewer is currently a proof-of-concept Web application it can still provide a valuable visual genomic comparison resource for biologists.

9.2.2 Mucosa project

“Comparative and evolutionary genomics of the surface proteome of mucosal microorganisms” [230] is an ongoing PhD project aiming to identify surface proteins associated with microbes thriving in a mucosal habitat. The project requires an extra-cellular protein identification pipeline that involves multiple protein analysis tools, as well as comparative genomics using sequence similarity data. The project has developed a distributed processing pipeline including novel databases and statistical analysis steps, as well as re-using many existing bioinformatics software applications. The bioinformatics tools required by the ‘mucosa’ project include: SignalP [26], TMHMM [173], InterProScan [337], LipoP [163], and Blast [6]. These tools have all been successfully run within the Microbase system.

The genome pool database used in the Mucosa project was co-developed by myself and Nakjang [230] and was later incorporated into the AGA pipeline.

Whilst the system is still in development preliminary results indicate promising values for system throughput:

- Approximately 1300 genome files were parsed and added to the genome pool. This was performed on a single desktop worker node in approximately six hours.
- 2.5 million proteins were analysed on cluster of between 40 and 50 machines over a 4 week period. Each Microbase job contained 100 proteins and took between 45 and 60 minutes to execute on a dual-core desktop machine.

The InterPro Scan work was a useful example of a relatively long-lived analysis task. The machines were available to Microbase almost un-interrupted during this time since most undergraduate students were away. During this time, several Microbase server restarts were required due to various problems associated with software that was still under active development at the time. The worker nodes were also forcibly rebooted once a week for routine system updates, causing jobs to fail and require subsequent re-runs.

The analysis pipeline developed by the Mucosa project is large and complex (Figure 9.1). At the current time, it has executed almost 200,000 jobs, using mainly the Linux desktop cluster machines available at Newcastle University. The pipeline developed for the Mucosa project was an incremental design, with responders being added to the system as research needs dictated. The design allowed for new tools to be added incrementally to the existing system. The Microbase notification system automatically informed the new responders of the existing message history so that the new responders could update themselves to the current system state. Microbase job descriptions were only generated for newly-added tools during this time. When new genome files are added to the system, jobs for all responders are generated as expected.

9.2.3 Parallel metaSHARK

The second project, undertaken by Illiasova [155] demonstrates the ability to parallelise the execution of metaSHARK. metaSHARK is a software package that identifies genes using solely unannotated DNA sequences as input data. However, it is extremely computationally intensive; a typical bacterial sequence can take in excess of 48 hours on a typical desktop machine.

In effect, metaSHARK is itself an entire pipeline of tools including PSI-BLAST [7], HMMER [80], MUSCLE [81] and GeneWise [34]. A parallelised version of metaSHARK using Condor already exists. It has required substantial modification to the original metaSHARK implementation. Each stage of the metaSHARK pipeline has been parallelised. Modification to the metaSHARK program itself has meant that any future version of metaSHARK must also be similarly modified in order to execute in a parallelised fashion. The aim of parallel Sharkhunt is to parallelise an unmodified metaSHARK distribution by dividing the input into manageable blocks, rather than dividing each execution stage. For example, for each DNA sequence to be analysed, several Microbase jobs are produced, each assigned a different set of PRIAM profiles [56].

The parallel Sharkhunt project is still under active development and is yet to be published. However, initial results have shown an almost linear speedup as the number of CPUs was increased. Currently, the project has been tested with 12 dual-core nodes. The high level of efficiency is probably due to the extremely CPU intensive nature of the work; analysis of a single genome sequence takes in the region of 24-48 hours to complete on a single machine.

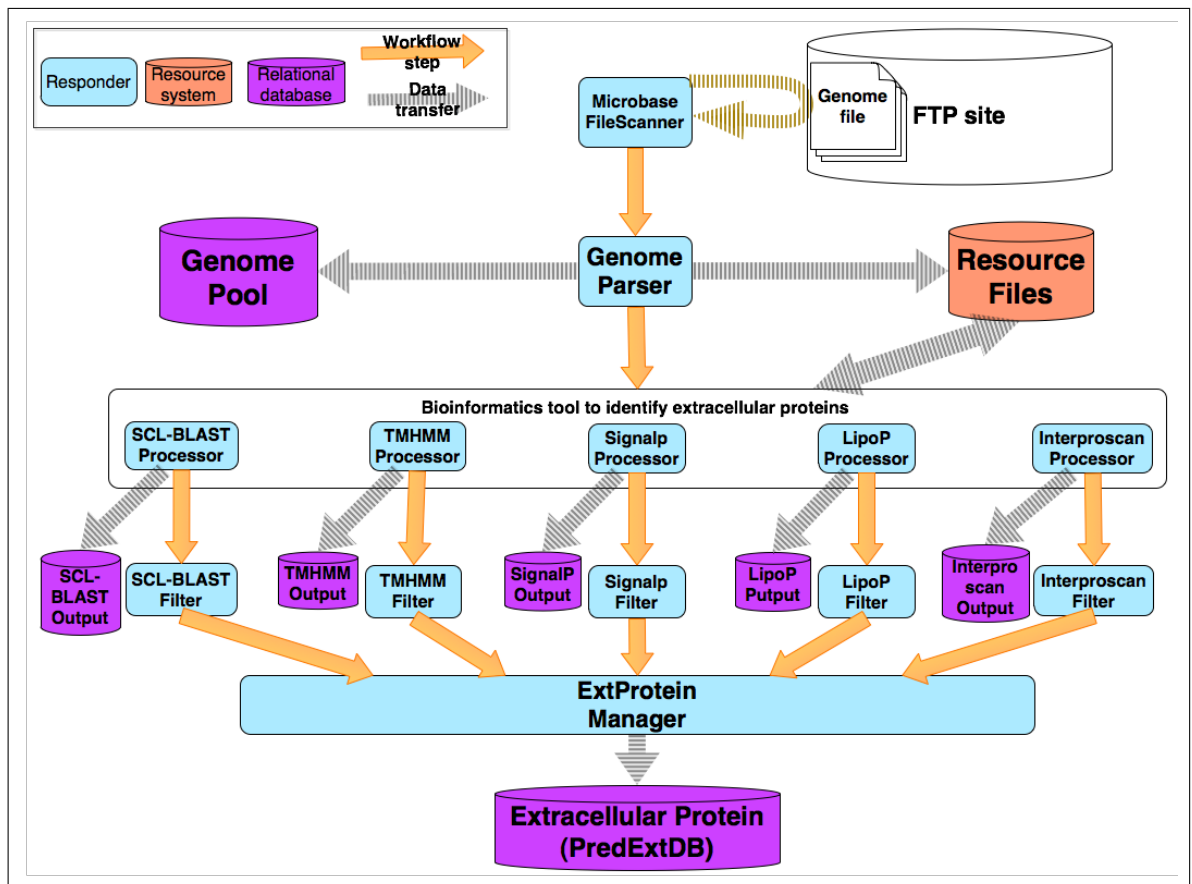


Figure 9.1: The Mucosa pipeline: diagram by Nakjang [229]. New genome files are detected and collected by the AGA ‘file scanner’ responder. Following an event notification, the ‘genome pool’ responder schedules a job for each newly-acquired genome file that simultaneously parses the file into a structured database, and generates FASTA sequence files that are archived in the Microbase resource system. On successful completion of these jobs, the ‘genome pool’ responder fires a ‘new genome available’ message. The Microbase notification system propagates this message to the five other responders shown to be interested in receiving ‘new genome’ events. The ‘Blast’, ‘TmHmm’, ‘SignalP’, ‘LipoP’ and ‘InterPro Scan’ responders then schedule their own jobs independently. After these jobs complete, another set of responders perform a filtering over the generated data sets, which flag the data items that are relevant to the Mucosa project using various heuristics. The filtering stage employs responders with server-based components only. The filtering stage is database-intensive so Nakjang decided the best solution was to perform the processing locally with respect to the data. Finally, a manual snapshot of each database is made, which pools all relevant results into an integrated database used for predicting extracellular proteins.

9.2.4 AptaMEMS-ID

Microbase provides a framework ideally suited for executing exhaustive analyses using a variety of tools. It has the ability to keep datasets updated, and the flexibility to add new tools without impacting the wider system.

The AptaMEMS-ID project [211] aims to identify unique surface proteins of infectious organisms, and therefore requires exhaustive analysis of bacterial protein sequences by using a number of software tools. The aim is then to apply pattern matching and machine learning techniques over the generated data set. Several relevant software components developed by the AGA and Mucosa projects are being successfully re-used with minor modifications to build the initial data set. Development is on-going and additional tools such as OrthoMCL [188] will be added to the pipeline in the near future. The AptaMEMS project will ultimately require the analysis of all available bacterial sequences.

9.2.5 iGem 2009

iGEM is an annual synthetic biology competition where students compete to build novel organisms from standardised building blocks. The Newcastle University team aims to simulate a population of cells living within an environment [325]. Each cell may perform various operations, such as consuming food resources, or spawning a daughter cell to an adjacent location within the environment. The purpose of the simulations are to determine how their modifications to an organism's DNA affects the growth rates of the population as a whole ¹.

The simulation project is slightly different to the other use-cases discussed so far. Instead of a traditional processing pipeline, where data flows from top to bottom, the cell simulation is effectively self-sustaining until terminated by the operator. The simulation of a cell is CPU intensive, so each cell executes within a Microbase job. When a cell 'spawns' another cell, a new Microbase job description is created for the new cell. The daughter cell process will then be initialised on its own worker node. A job only ends when a cell eventually dies, so the simulation as a whole tends to become more CPU intensive the longer it remains running. The iGEM team plan to utilise the Amazon EC2 service [1]. It took the software developers on the team about a week to learn Microbase, deploy their own Microbase installation and implement a responder for executing their simulation application [297].

¹<http://2009.igem.org/Team:Newcastle/Modeling/Population> [accessed 2009/09/27]

9.3 Evaluation

One of the benefits of Microbase for large organisations such as universities is to utilise their existing desktop machines for computationally-intensive tasks. The effects on the primary users of the desktop computers can be minimised by running Microbase on top of a system such as Condor, which can be configured to shut down jobs when a local user is detected. Microbase then re-queues the abandoned job and migrates it to another node, installing necessary software if necessary. Of course, powerful dedicated cluster nodes can also be used for processing jobs. It was concluded in Chapter 8 that Microbase jobs could be migrated around the world if necessary.

In order to maximise the potential throughput, the application developer must use the Microbase functionality appropriately. The developer is responsible for ensuring that tasks are divided into appropriately-sized units. If too many jobs complete too quickly, the resource storage system becomes strained and worker nodes either spend their available time performing BitTorrent peer discovery operations required for file transfers, or sit idle due to a fail-safe throttling mechanism designed to temporarily suspend new jobs from starting until the resource system load reduces to normal levels.

Microbase itself is cross-platform and is able to run on any platform with an appropriate [JVM](#). However, the lack of availability of binary executables for some platforms, particularly for bioinformatics applications reduces the number of worker nodes that can be harnessed. For instance, although there are many thousands of CPUs available within the Newcastle Condor pool, in practice bioinformatics analysis work is performed on the Linux clusters — roughly 10% of the available CPUs.

9.3.1 System efficiency and job design considerations

The suitability of Microbase for a particular workload must be evaluated prior to writing a responder. When designing and implementing a responder, it is important to understand the advantages and limitations of the system. To a large extent, the suitability of Microbase for a particular class of work depends on the ability of jobs to work in isolation from one another and to transfer data in large blocks, rather than in smaller fragments. Microbase performs much more efficiently when the data workloads are amenable to BitTorrent transfer.

Efficiency analysis of the [AGA](#) pipeline has shown that the less CPU-intensive and the more data intensive a workload is, the less efficiently the work will be completed. This is to be expected, given the nature of the Microbase resource system that relies on the BitTorrent protocol. The minimum amount of computation per job for a Microbase implementation of an application to be worthwhile was calculated at around minutes (see Chapter 7).

The scalability of the resource system may be undermined by compute job implementations that require continual access to other shared resources (such as relational databases) throughout the course of their execution. When running computational tasks on a single computer, the emphasis is on the developer to write an implementation that is efficient as possible. This task includes ensuring that no superfluous computation or data copying is performed. In the case of an application running on a single machine or on a small set of nodes, the most efficient means of obtaining a data set may be to perform a highly specific query to a database that returns exactly the data required for a specific unit of computation. Given appropriate indexing, such a query should be reasonably fast even for large datasets, and has the benefit that only the exact amount of data that is required for the computation is transferred, keeping network traffic and server load to a minimum. However, in a Grid environment, the ability of a system to scale to many hundreds or thousands of nodes outweighs the efficiency concerns of an individual node. The database would be swamped with requests if many hundreds of nodes attempted to connect simultaneously.

Therefore, in the Microbase resource system the level of granularity at which data can be requested is not that of an [SQL](#) table row, but that of a block of data represented by a torrent. The actual size of a torrent depends entirely on the publisher and may range from several kilobytes to several gigabytes. For jobs that need to work on a portion of this block, rather than the entire file, this means that significantly more data may be transferred than is actually used by the worker node. While this seems a waste of bandwidth from the perspective of the worker node, it does have a number of advantages for the system as a whole. The major benefit is the ability to use BitTorrent as the basis for large-scale data transfers; the compromise is the tradeoff of ability to support large numbers of nodes against the possibility that some workloads may not use all of the data they transfer. On the other hand, if multiple worker nodes are working on different parts of the same data file, the extra work involved in transferring the whole file is not necessarily wasted, effort since other worker nodes can benefit from increased resource availability and network bandwidth.

9.3.2 Service and data security in a Microbase system

At present, Microbase is suited to a semi-secure environment. At Newcastle University, desktop computers are trustworthy since administrator access is restricted to support staff, and there are adequate file permissions to prevent tampering with Microbase-installed software and data files. Likewise, Amazon provides suitably private virtual machines instances [1]. Presently, the Microbase Web services provide the greatest potential security threat. There is some protection against bugs or malicious attempts to call Web servicemethods. For example, one worker node may not interfere with or post

results for a job assigned to another worker node. However, no communications are currently encrypted and there is no strong authentication. Although security is less of a concern when operating Microbase across a set of privately-addressed machines, it is potentially one of the biggest barriers to the use of Microbase when using global compute resources. Nevertheless, this Web service security should not be a long-term problem, since there are several well-known and readily-useable methods that could be adopted, including WS-Security [228] and signed certificates, coupled with hosting Web services on Secure Sockets Layer (SSL)-enabled application containers.

9.3.3 Achievements

Although the potential of AGA has not yet been reached, it has provided valuable knowledge and insights into developing high throughput applications that can be executed across multiple geographical sites. The insights gained while experimenting with the AGA pipeline have allowed other developers to construct their own analysis pipelines using efficient, distributed components. The use-cases described in the previous section have shown that highly data-intensive and long-term analysis pipelines can be successfully enacted by Microbase. The system has been successful in overcoming both worker node and server failures. While the system does periodically fail and requires user intervention to restart Web services, it fails ‘safe’.

The developers of the pipelines described in the previous section have some experience in computing science and Java programming, but all have very varied backgrounds including bioinformatics, medical science and mathematics. I believe that the responder architecture of Microbase enables developers to parallelise these kinds of applications in a relatively straightforward manner, without having to consider the usual difficulties associated with parallel systems, such as locking shared resources to prevent simultaneous updates. In part, this can be attributed to the way Microbase operates, the restrictions it places on the way responders operate, as well as the overall ‘best working practices’ of responder design:

- The resource storage system supports only immutable data items. This allows for highly-scalable file distribution via BitTorrent. However, a side effect is that no conflicting updates to files can be made by compute jobs. Any changes to data require a new ID to be assigned.
- The notification ensures that messages are delivered to responders reliably, and in the correct order. The responder architecture then enforces serial the processing of notification messages.
- Developers are encouraged to ensure that each responder they write maintains its own independent result database. Each responder is should also perform all database write operations

within the server-based component of the responder. The server component is forced to process notification messages serially, reducing the risk of competing or conflicting updates to data sets.

For many research applications, distributed sets of processes can be executed largely independent of each other. We have found that most bioinformatics applications can be parallelised effectively within the above restrictions. Meanwhile, the way in which responders operate permit convenient and straightforward implementation options for the application developer.

Another major achievement has been the ability to use existing, well-understood protocols such as [SOAP](#) and BitTorrent to permit cross-site operations with relative ease. Both technologies have proven themselves to be amenable to Internet use. Web services are not typically blocked by corporate firewalls, enabling convenient cross-site communication. Although BitTorrent appears to be a somewhat less “socially acceptable” form of content distribution — there have been numerous queries from the network security staff at Newcastle requesting details of what data had been transferred to remote nodes — the sheer efficiency and scalability advantages of BitTorrent, as well as its decentralised nature are indispensable to the scalability of the Microbase resource storage system. The resource system has the potential (see [Chapter 8](#)) to save an organisation large amounts of their own institution’s bandwidth, and also reduce costs when using rented commodity hardware, such as Amazon’s EC2 system. Although it would be possible to manually stage large files at remote locations, the resource system described in [Chapter 5](#) ensures that worker nodes themselves provide the necessary file mirroring capacity dynamically.

Finally, the Microbase system and the AGA analysis pipeline has been made available as SourceForge project², where it will continue to be developed.

9.4 Future work

There are a number of specific changes and investigations that could be made to enhance Microbase. One future development would be the inclusion of a number of other data transfer protocols to the resource system, in addition to BitTorrent. These protocols would be selectable at runtime by the requesting worker node. For example, BitTorrent could be selected as the transfer mechanism where large (e.g., > 15Mb) files are to be transferred. A simpler, lower-latency protocol such as point-to-point HTTP or FTP [[92](#), [245](#)] among would be preferred where ‘small’ files must be transferred

²<http://sourceforge.net/projects/microbase/> [accessed 2009/09/28]

since in these cases, a large proportion of the time taken to copy such a file is due to BitTorrent peer discovery, rather than actual data transfer [317].

It would also be beneficial to investigate the use of P2P technology for more than file transfers. Currently, the server-based components of responders may become overloaded if too many requests are made. For example, a Microbase job server instance may become slow if too many worker nodes connect to it simultaneously. In order to alleviate a highly-loaded server a system administrator would need to deploy more instances of the service to another physical host. One potential area for future work would be to permit starting temporary responder service instances as jobs that run on worker nodes. The idea being that these temporary instances would deal with the surge of requests, and eventually trickle data back to the central job server instance. A number of reliability and security aspects would need to be investigated to achieve this aim. Another possibility would be to use a Distributed Hash Table (DHT) in order to find other worker nodes processing similar types of job. Finding other nodes running the same applications may be useful if P2P IPC is required among distributed processes.

The Microbase administration user interface could be improved significantly. Currently, a Web-based interface provides live monitoring of several aspects of the system such as: job queues, the ability to browse notification messages and resource system file metadata. A Taverna-like workflow viewer and editor would make complex pipelines easier to visualise for application developers [152]. The integration of Microbase with Taverna workflows would be of benefit to the bioinformatics community. A large number of existing workflows are available for download [122]. Re-using existing Taverna workflows by incorporating them into a Microbase responder may speed up development time, and allow multiple instances of such workflows to be executed in parallel. A Taverna workflow enactor could be packaged within a Microbase job, allowing application developers to incorporate existing workflows within a Microbase pipeline. The workflow editor could also be used to design jobs for Microbase if the complexity rises to the point where a plain Java implementation becomes difficult to maintain or too verbose. The use of a graphical editor may also reduce the barrier to developing Grid applications, particularly if developers are not fluent in Java.

AGA could be extended to make use of the amassed analysis data to further research in comparative genomics. Several evolutionary pressures act on bacterial genomes, resulting in continual flux from biological processes such as deletion and lateral gene transfer events [220, 226, 200]. Artefacts of these biological processes can be observed as features such as insertion, deletion, translocation, and inversion events. Currently, these rearrangements can be viewed graphically with tools such as the Artemis Comparison Tool [48] and GenomeComp [332]. These rearrangement features can reveal

important aspects of the functionality and phenotypes of bacterial organisms. However, comparing large numbers of sequences manually is time consuming, and requires an experienced biologist to analyse each pair of sequences. With the increased rates of genome sequencing now being seen, it is becoming increasingly infeasible for biologists to manually analyse these sequences. Therefore, it is becoming increasingly necessary for computational methods to aid biologists in the systematic derivation of knowledge from this data. AGA would provide the base from which such comparison software could be built by providing up-to-date homology data. Additional responders would be needed for further sequence analysis, such as IslandPath [150]. A combined logical and probabilistic approach has already been developed, and it would be interesting to combine this with in a high-throughput fashion with Microbase [95].

Further development of the AGA browser Web application is an area of particular interest. One possible area of future work would be the integration of the notification system with the AGA viewer application. Biologists might be able to set up triggers that run small, pre-defined queries in response to a notification event. These ‘responder-lets’ might be configured to send an email or otherwise notify the biologist when a new set of data arrives that is of interest to them. For example, when analysis results regarding new organism closely related to one they are studying become available, or when a new `Blast` report contains hits to a gene or set of genes they are investigating.

The nature of academic software means that many analyses have not been tested under a wide range of configurations. For example, applications may make assumptions about the availability of administrator access is available; others might be heavily-reliant on third-party libraries which require administrator access to install. One area of exploration for future research would be the use of `VM` technology to increase configuration flexibility and expand the number of worker nodes available to Microbase. Several Grid projects have already exploited the advantages that virtualisation provides [94, 118, 221]. Microbase already uses `VM` images with the Amazon EC2 system. At Newcastle University, the use of virtualisation technology for Microbase is feasible. Virtualisation technology is already in use to provide students using Linux clusters access to a standard campus Windows installation without the need to reboot their machine. While the use and installation of a Microbase `VM` on every campus desktop would still require a potentially lengthy administrative process, there is at least a precedent for the technology’s use. Although virtualisation is a source of further system overhead, including both an additional hypervisor layer, and the extended start-up times associated with launching a virtual machine, it offers significant advantages over using a ‘raw’ physical machine and there is evidence to suggest that the additional overheads are not massive [221]. A `VM` could be customised to better suit Microbase and its users. Whereas system administrators are reluc-

tant to provide privileged access to physical hardware, it is probable that administrator access would be provided to certain users who oversee a Microbase installation composed of VMs. The installation of a Linux VM on campus Windows machines would allow these currently unused machines to participate in large-scale bioinformatics analyses. Also, more people may be willing to donate their machines to the processing pool if all Microbase processes were contained within a VM; there is a hugely reduced risk of Microbase processes adversely affecting the host machine, since VMs typically work within enforced limits of RAM and disk space.

As Cloud computing technology continues to mature, the potential computing power available to researchers has increased to a point unimaginable a few years ago. At the same time, available data in bioinformatics is increasing at an equivalent or even greater rate. There is a need for parallel and distributed computing frameworks which have a straightforward programming model that hide the underlying complexities. However, abstraction usually comes with an increased risk of trade-off against flexibility, speed, or efficiency. In order for Grid abstractions to deliver on performance as well as simplicity, the middle-wares they are built upon must make the most efficient use of the available hardware. The work presented in this thesis has addressed several of these challenges by providing a design pattern that fits the usage patterns of a large number of bioinformatics analysis tools, backed by an appropriate framework that has been shown to fit current Grid and Cloud distributed computing models.

Appendix A

How to write a responder

A.1 Introduction

Microbase implements a generic Grid infrastructure, providing your applications with a distributed processing environment to operate in. A common use-case for Microbase is to wrap existing non-distributed, command line applications in a such a way that many instances can be run in parallel within a distributed environment.

At the end of this tutorial, you should be able to:

- package and deploy applications to a distributed environment
- use event notifications to schedule computationally-intensive tasks
- monitor jobs running in real time

To run programs within the Microbase environment, a `responder` must be implemented. Responders are modular components that can be registered with a Microbase installation to allow domain-specific functionality to be integrated. Responders act as the interface between the Grid-based Microbase core components, and stand-alone domain-specific applications. Compute-intensive applications wrapped by a responder might either be pure Java, or may be an existing command line utility written in any language. A common use-case for Microbase is to wrap existing command line applications in order to run them in parallel over a set of distributed worker nodes. This guide explains how to implement a responder as a thin wrapper around an existing command line application, and how to overcome operating system and CPU-architecture differences when deploying native applications to a group of heterogeneous worker nodes.

This guide explains how to package a widely-used bioinformatics application, [BLAST](#).

A.2 Microbase

To successfully build, deploy, and run a distributed application using Microbase requires three separate environments: a development environment; a server environment and a work environment. The *development environment* contains the necessary tools and source files to compile your application(s). It is also the base from which these applications will be deployed to the server environment.

The *server environment* hosts much of the core Microbase infrastructure, and the server-based parts of user-developed responders. This includes a set of Web services(hosted in a container such as Tomcat), and their supporting databases (such as PostgreSQL). This environment provides permanent storage of Microbase housekeeping data, log files as well as user-data generated from responders. As such, the server environment should be located on ‘reliable’, dedicated machines. The *server environment* should perform ‘lightweight’ operations, such as responding to new data events, and scheduling appropriate CPU-intensive jobs to run in the *work environment*. In other words, entities hosted in the server environment should **not** perform computationally intensive work that could otherwise be farmed out to the *work environment*. The server environment may consist of one or more physical or logical servers. Different core Microbase components or responder implementations may be deployed at will to any number of available servers, depending on load-balancing or other concerns such as disk storage availability. Web servicesmay also be located on different servers than their supporting databases.

The *work environment* is responsible for performing computationally-intensive work. The *work environment* consists of one or more worker nodes whose hardware may range from a standard office PC, up to high-specification dedicated compute cluster node. The only requirement is an installation of Java, and some means of starting the Microbase compute client, for instance via ssh, Condor, Sun Grid Engine, or equivalent.

A.2.1 Requirements

A development environment containing:

- Java 6 JDK
- Subversion: used for managing the Microbase source tree

- Microbase source code: your responder will be built against the provided public APIs
- Maven 2.0.x: used for building projects. Also used to generate the skeleton structure of new responder projects.
- Google Web Toolkit: used for building the Microbase web-based GUI
- Development environment: An IDE such as Netbeans 6.x is recommended (Netbeans supports Maven projects with the appropriate plugin)

Deployment server providing:

- Java 6 JRE
- Tomcat 6.x: used as a application container for Microbase core services, and server-side components of responders. Tomcat may be obtained from: <http://tomcat.apache.org/>.
- PostgreSQL: structured storage system used by Microbase components. Versions 8.2 and 8.3 have been tested.
- SSH server: used to copy compiled web applications from the development environment to the deployment server.

Work environment with:

- Java 6 JRE
- Some means of starting the compute client (manually, via SSH, via Condor, etc ...)

A.3 Quick-start virtual machine image

In a ‘real world’ deployment, each of the environments described in the previous section would be located on physically distinct hardware. However, setting up a large-scale system suitable for distributed computation involves a large amount of system administration (installing servers, configuring database server connectivity, etc). This is a daunting task, especially for new users to learning how to write applications for the Microbase framework. Therefore, for the purposes of learning and small-scale development and testing of responders, we have constructed a virtual machine quick-start image containing everything required to write and deploy a simple responder application. You

can immediately get to work on your responder without the hassle of installing Microbase, PostgreSQL, Tomcat, etc from scratch. The VMware image provided will obviously not be able to provide production-scale performance. You can use the virtual machine as a convenient environment as both your development and deployment system. However, for more serious development it would make more sense to use the VMware appliance as a staging deployment server, while performing actual development on a faster physical machine.

In terms of hardware requirements for the virtual machine, RAM is the most limiting factor. To run the VMware machine, we recommend a host machine with a minimum of 1GB. The virtual machine image is configured to provide 600MB RAM to the guest operating system. If your host machine has more than 1GB RAM, we recommend ‘upgrading’ the virtual machine to use a higher amount of RAM for increased performance.

The virtual machine image may be downloaded from this location:

<http://madras.ncl.ac.uk/microbase-vmware/>

It should be possible to run the image on a Windows or Linux PC by downloading the free player available here: <http://www.vmware.com/products/player/>

PCs running Mac OSX need to use: <http://www.vmware.com/products/fusion/>

Virtual machine notes:

- Once opened in VMware, you can use the username ‘microbase’ and password ‘microbase’ to log into the virtual machine. If you need administrator access (for restarting services, etc), you can use the command `sudo -s` to obtain a root shell.
- If the screen resolution appears low, it can be increased by resizing the VMware window *after* proceeding past the initial login screen.
- Increasing the memory available to the virtual machine can dramatically increase performance.
- If you encounter any problems implementing the responder developed during this tutorial, the virtual machine image contains a complete ready-to-compile responder project that can be used for reference purposes. It is available in the directory `$HOME/microbase-trunk/microbase-tutorial`.
- The virtual machine supports ‘snapshots’. This feature can be used to save progress through the tutorial, or to roll back changes. As a last resort, it is possible to roll back to the snapshot named ‘original’ which restores the VM to its original state.

A.4 Responder architecture

A responder wraps an entire unit of domain-specific functionality. This typically comprises:

1. a server-side component for responding to external event notification messages from the Microbase system
2. compute component(s) that perform the CPU-intensive operations. Multiple instances of these components will be run in a distributed fashion.

Responders can either be written purely in Java, or can wrap an existing application written in any language. The guide focuses on wrapping an existing command line application within a responder, in order to run it within a distributed environment. This is the most challenging case, since it requires the platform-native command line application to be packaged in an appropriate way for automated deployment.

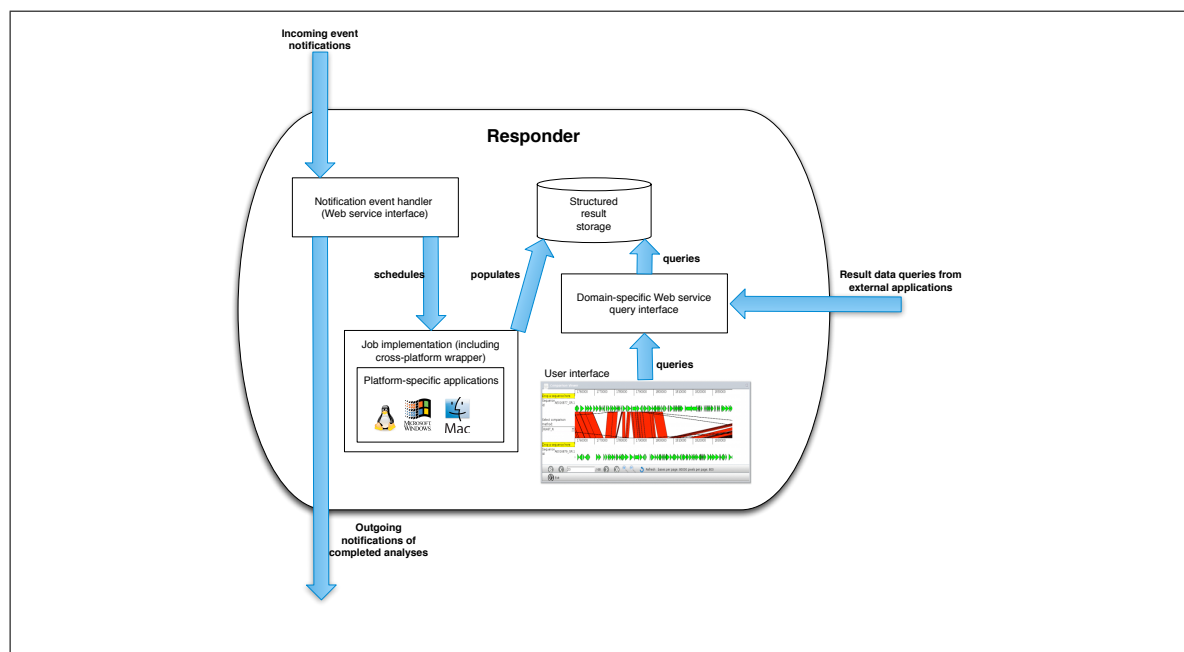


Figure A.1: Responder architecture

The event handler part of a responder is responsible for receiving and acting upon events from external sources. In the 'real world', these events will deliver structured, machine-parsable message from other responders or other sources (such as a process that periodically scans FTP sites for new data). For the purposes of this guide, the events will be simple plain-text messages. Event handlers must interpret incoming event messages to determine the amount of computationally-intensive work that needs to be performed. The event handler should **not** perform this work itself; it should merely

inform Microbase that there is new work to be performed. Microbase will handle the actual queuing, scheduling, worker node configuration, job enactment, and job failure retries. Finally, Microbase will inform the responder's event handler when the computational work is complete, via another notification. Event can be used to chain responders together into an automated processing pipeline, with the output of one responder triggering the input of the next, and so on.

The 'compute job' part of a responder is responsible for performing necessary 'heavy lifting' computational work. This may include custom Java code, or command line applications. Jobs need to be distributable and migratable to worker nodes available to the system.

The distinction between these two types of component arises from the need to deploy event handlers to a server environment, and compute jobs to worker nodes. The distinction is also useful for responder development purposes, cleanly separating what work to run (event handler) from how to run that work (compute job).

A.5 Writing a responder

This guide explains how to construct a simple responder that will perform all against all pairwise alignments of several bacterial genome sequences. This is an $O(n^2)$ problem on the number of genomes to be compared, so is ideally suited to being processed in a distributed environment. The remainder of this guide assumes that you are at least familiar with the basic Microbase architecture. It also assumes that you are using the virtual machine quick-start appliance, or already have a working Microbase installation deployed.

One of the most challenging aspects of constructing a Microbase responder is creating the initial project structure that will house your application. To be able to interact with Microbase, your application will depend on several libraries provided by the Microbase system. Additionally, some components of a responder will reside within a server environment (i.e., Tomcat), while others need to migrate between, and execute on worker nodes available to the system. This requires that the different types of component are handled appropriately, and registered correctly with Microbase.

Microbase provides tool support for automating many of these deployment and registration tasks. However, to do so it must be able to introspect your project's structure to determine the function, and therefore the appropriate destination for each component. This means that your responder project should follow a structured, modular pattern. The advantages of following this approach are:

- it is immediately obvious to the developer which parts of the project execute on a server, and

which parts execute on worker node.

- all responder projects will follow the same basic layout, enabling faster responder development once the design pattern is mastered. A familiar project layout also aids understanding of responders developed by other people.

In order to simplify matters for both the developer, and the automated install process, we have developed a set of Maven archetypes¹ that help to create the initial project structure template. These archetypes handle the construction of necessary project directory structures, sample Web service-configuration files and template Java code. The Microbase libraries required by responders are also added as project dependencies. In short, the Maven archetypes will perform all the necessary administrative work required to allow you to start writing Java immediately. Project(s) created by the archetypes can be opened immediately in Netbeans.

A.5.1 Root project directory

It is recommended to organise all of a responder's components under a root project directory named after the responder. This is useful if your application requires the development of several responders. Each responder then has its own components neatly contained in its own directory. This approach is useful if responder code needs to be shared between multiple applications; a copy can be made of the root directory, which includes all responder-specific web-services, database support and user interface code.

To start, a root project directory needs to be created. This project does not actually contain any code, but it will act as a home for your responder's components (for 'real' responders, you should probably change `uk.ac.ncl.mygroup` and `my-new-responder` to something meaningful):

If you are using the virtual machine, double click the terminal icon labelled home on the desktop to obtain a terminal window. The following command should create a new responder project root directory within your user's home directory. Since this is quite a long command line, it is probably easiest to copy and paste into the terminal window.

```
mvn archetype:create \  
-DarchetypeGroupId=uk.org.microbase \  
-DarchetypeArtifactId=responder-base \  
-DarchetypeVersion=1.0 \  
-DgroupId=uk.ac.ncl.mygroup \  
-DartifactId=my-new-responder \  
-Dversion=1.0
```

¹See <http://maven.apache.org/guides/introduction/introduction-to-archetypes.html> for an introduction to Maven archetypes, although a full understanding is not required.

After a few moments, you should see `BUILD SUCCESSFUL`, and a new directory will have been created, containing a single `pom.xml`:

```
my-new-responder/  
'-- pom.xml
```

The `my-new-responder` will now be the root directory for this responder project. This directory is just a standard maven project directory. This tutorial covers Microbase-specific modules only, but any Maven project may be added as a child project.

A.5.2 Compute job sub-project

A compute job implementation executes within the work environment (A.2). This section describes how to package a command line application for deployment to a Microbase system, and how to write the necessary Java class wrapper.

A.5.2.1 BLAST

First, let's take a look at the application to be wrapped and its requirements. The program can be downloaded from here: <http://www.ncbi.nlm.nih.gov/BLAST/download.shtml>, or is available in the VMware session under `$HOME/blast`.

BLAST is capable of aligning either DNA or protein sequences. Several bacterial sequences have been placed into `$HOME/genomes`. To see it in action:

```
cd ~/blast/blast-2.2.18/bin  
./bl2seq -p blastn -e 0.00058 -i ~/genomes/NC_000964.fna \  
-j ~/genomes/NC_002570.fna -o blast_output.txt
```

This particular example should complete within a few seconds. It should create a file `blast_output.txt` of around 3MB. The output file contains the alignment. The actual content is not important, but you might wish to compare it to the output generated by Microbase, when the job implementation is executed, later on. However, what is of interest is the way the program was run and the meaning of the various parts of the command string:

- Executable program name: **bl2seq**
- Input parameters (passed by value): **-p blastn** and **-e 0.00058**
- Input parameters representing files (passed by 'reference'): **-i /genomes/NC_000964.fna** and **-j /genomes/NC_002570.fna**

- Input parameters representing output files (passed by ‘reference’): **-o blast_output.txt**

The program name corresponds to the file name and location of the executable on the computer’s disk. Input parameters are values that are passed to the program in-line. The content embedded within the command line string are a data items that will either be used as-is by the application, or parsed into an appropriate data type, such as a floating point number or a file name. Input parameters are necessarily small data items in order to fit within an operating system’s command line buffer space. While small quantities of data can be passed ‘in-line’ (such as the `-e` value above), when large quantities of data need to be accessed by a program, the data must be placed into a file. An input parameter is used with an appropriate ‘pointer’ value to the file containing the required data content. Similarly, input parameters can be used to specify where a program places its output files.

The distinction between the different parts of a command line is important when executing an application within a Microbase environment. In a distributed environment executable files, data files, and command line strings need to be transported to remote worker nodes. Microbase implements a bulk data transport mechanism that is efficient at transferring large blocks of data, such as entire files. However, this transport mechanism is much less efficient at transferring tiny data items such as the `-e` cutoff value. Small data items are transferred via a more appropriate method. The responder developer needs to be inform Microbase which transport mechanism should be used for each command line parameter.

The heterogeneity of the worker nodes requires that the responder job implementation and the Microbase framework work together to ensure that executable files are installed on worker nodes with a matching platform since programs compiled for Windows will not execute on Linux, and vice-versa. Essentially, Microbase will ensure that the correct version of platform-native software is installed on worker nodes, assuming that the responder:

- provides platform-native executable files in a package suitable for distribution via the Microbase resource system,
- and these packages are tagged with appropriate meta-data that indicates the operating system and processor architecture they are intended to execute on.

Satisfying the responders’ obligations, therefore requires:

1. Writing a Java wrapper. This specifies the I/O requirements of a computational job, and the transport mechanism to be used for data items.

2. Writing a mapping file used by Microbase to determine executable command paths in a multi-platform environment
3. ‘Zipping’ the native application directories, together with appropriate mapping file. One package is required for each platform to be supported.
4. Tagging the resulting zip file(s) appropriately, so that worker nodes may query for it at run-time

This implementation process will now be explained.

A.5.2.2 Java component

Begin by creating a job sub-project for the responder. To create a new compute job project, execute the following from within the *responder root directory*:

```

mvn archetype:create \
  -DarchetypeGroupId=uk.org.microbase\
  -DarchetypeArtifactId=job-quickstart \
  -DarchetypeVersion=1.0 \
  -DgroupId=uk.ac.ncl.mygroup \
  -DartifactId=my-compute-job \
  -Dversion=1.0

```

As a result, your responder project should now look like this:

```

my-new-responder/
|-- my-compute-job
|   |-- pom.xml
|   |-- src
|       |-- main
|           |-- java
|               |-- uk
|                   |-- ac
|                       |-- ncl
|                           |-- mygroup
|                               |-- HelloJob.java
|           |-- mb-resources
|           |-- test.foo
|           |-- resources
|-- pom.xml

```

A.5.2.3 Implementing the Java component of a job

At this point, it should be possible to load the project created in the last step into the NetBeans IDE. Launch the IDE (the NetBeans icon on the desktop), and open the “my-new-responder” project (remember to enable the “Open Required Projects” option in the open dialogue box).

You should now see two projects listed in the “projects” pane on the left hand side. Expand the project named “uk.ac.ncl.mygroup-my-compute-job”. Open the “Source Packages” tree until you reach “HelloJob.java”. Open this file in the editor.

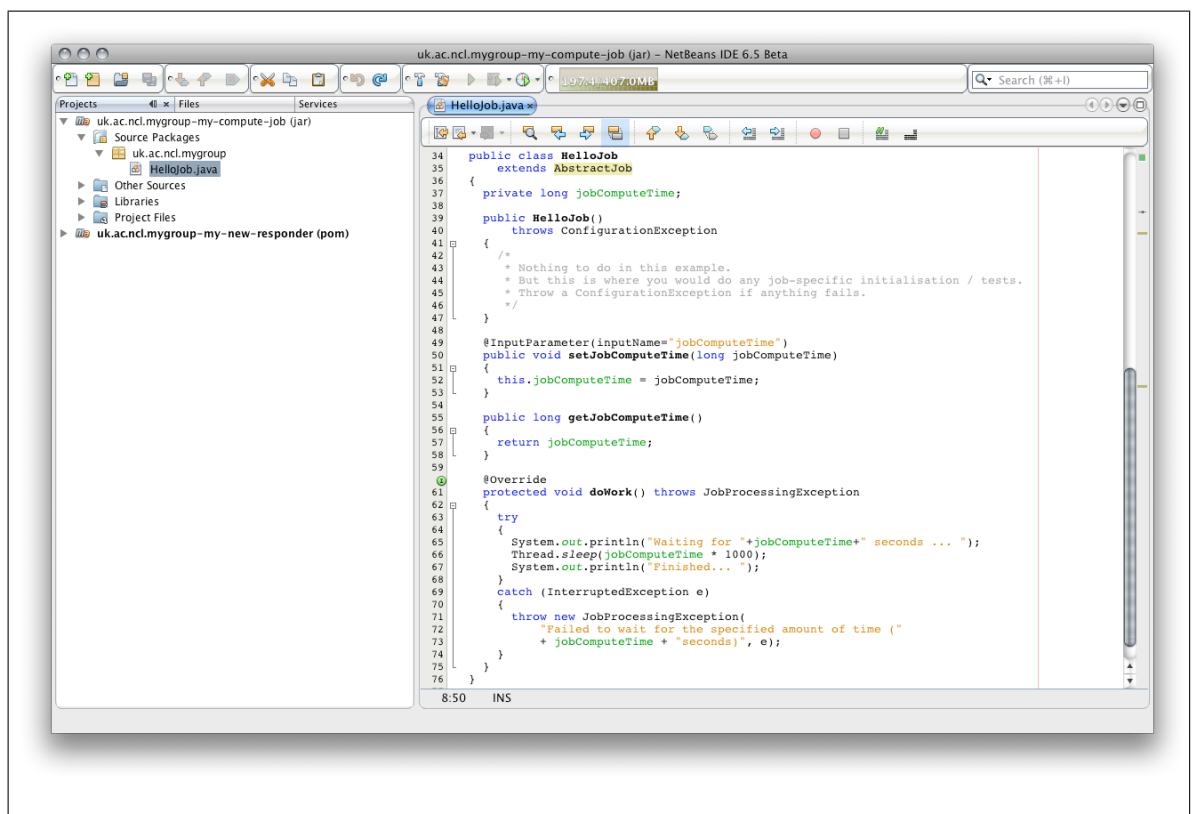


Figure A.2: The ‘hello world’ job as created by the Maven archetype. This job takes one input parameter that specifies how long the job should ‘compute’ for. The doWork method uses this parameter to wait the specified number of seconds.

There are several items of interest:

- a no-arguments constructor
- a property accessed by bean-style getter/setter methods, with annotations
- a “doWork” method

At this point, some information about how Microbase enacts a job may be useful. The steps involved

from the point of view of a worker node are as follows. Bold text indicates where control of the execution thread is passed to the user job implementation:

- The Microbase compute client running on a worker node requests a unit of work from the server. A job description is returned, if there is work available.
- The job description is examined to determine which resources (executable files, input files) need to be downloaded
- After acquiring necessary resources, the Java component of the job implementation is class-loaded. **The constructor of the job implementation is called at this point.**
- The job instance is informed of input parameters and input resources. **The ‘set’ methods of annotated input bean properties are called at this point.**
- The job is then ready to run. **doWork() is called at this point.**
- Job result files are uploaded to the Microbase resource system. **The ‘get’ methods of annotated output bean properties are called at this point.**
- The job implementation class is unloaded, and the entire process repeats with a new unit of work.

The compute job implemented in the example simply prints a message, waits a specified number of seconds, prints a second message, and then exists. It takes one input: an integer value that specifies the number of seconds that it should ‘work’. The input parameter is stored in the variable *jobComputeTime*. This looks like a standard bean property, with ‘getter/setter’ accessor methods with one difference: the ‘setter’ method is annotated with `@InputParameter(inputName="jobComputeTime")`. This annotation is important - it notifies Microbase that ‘jobComputeTime’ is a property that needs to be set before the job is executed.

There are several types of annotation that can be used to specify the inputs to job implementations:

@InputParameter allows the developer to specify ‘small’ data items to be passed to the Java implementation. Valid Java data types are: primitive types (int, long, boolean, etc) and String. Although it may be possible to send ‘large’ (>MBs) Strings as parameters, this is not recommended.

@InputResource allows ‘large’ resources to be sent efficiently to a job implementation via the Microbase resource system. `@InputResource` allows the use of many structured data types

to be used within a Java compute job without the need for the implementation to have knowledge of where the resource originated from, or how to marshal / un-marshal objects across a network. Almost any `serializable` Java type may be specified here (Microbase will handle deserialisation of complex types, including `Maps`, `Sets`, `Lists`, etc). The Java type `File` may also be specified for objects that will not fit into RAM, or for objects that need custom deserialisation. In this case, the raw file is available to the job implementation. Using `File` is especially useful if input data is required for a command line application.

@PlatformSpecificResource used to specify a resource whose content depends on the hardware or operating system platform that the compute job is executing on. This input type is useful for requesting a platform-native binary in a platform-neutral way. Note that any bean property annotated with `@PlatformSpecificResource` *must* be associated with the data-type `NativeExecutable`. This data-type is an interface to a packaged command line application. It allows execution of these applications in a straightforward manner. An example will be shown below.

Annotations can also be used to specify compute job outputs:

@OutputResource allows result items (such as structured Java objects, or files created as a result of executing a command line application) to be ‘collected’ and archived by Microbase. The same data-types supported by `@InputResource` are also supported by `@OutputResource`.

A.5.2.3.1 Job I/O To run BLAST, we need to pass a mixture of input parameters and input resources. First, delete the property `jobComputeTime` and its associated methods. Add the following properties to `HelloJob`:

```
//Native executable
private NativeExecutable blastExePkg;

//Input data
private double eValue;
private File firstSequence;
private File secondSequence;

//Output data
private File alignmentOutput;
```

Figure A.3: The BLAST job implementation will require several inputs and one output. The inputs are: the executable BLAST package; a Java primitive value; two sequence files. The output is a file containing the alignment.

Then add the appropriate getter / setter methods (see Figure A.4 on the following page). Notice that the annotations on these methods specify whether the property is an input or an output, and

```

public NativeExecutable getBlastExePkg()
{
    return blastExePkg;
}

@InputResource(inputName="blastExePkg")
@PlatformSpecificResource(resourceName="blast", resourceVersion="2.2.18")
public void setBlastExePkg(NativeExecutable blastExePkg)
{
    this.blastExePkg = blastExePkg;
}

public double getEValue()
{
    return eValue;
}

@InputParameter(inputName="eValue")
public void setEValue(double eValue)
{
    this.eValue = eValue;
}

public File getFirstSequence()
{
    return firstSequence;
}

@InputResource(inputName="sequenceOne")
public void setFirstSequence(File firstSequence)
{
    this.firstSequence = firstSequence;
}

public File getSecondSequence()
{
    return secondSequence;
}

@InputResource(inputName="sequenceTwo")
public void setSecondSequence(File secondSequence)
{
    this.secondSequence = secondSequence;
}

@OutputResource(outputName="alignment")
public File getAlignmentOutput()
{
    return alignmentOutput;
}

public void setAlignmentOutput(File alignmentOutput)
{
    this.alignmentOutput = alignmentOutput;
}

```

Figure A.4: Microbase must be told which of the properties (defined in Figure A.3 on the preceding page) are inputs and which are outputs. The mapping is achieved by annotating the accessor methods, as shown. Notice that the values of annotation properties such as `inputName` and `outputName` do not need to match the bean property names. **Note:** to save some typing, alt-insert can be used within Netbeans to auto-generate the getter/setter methods. Then, only the annotations need to be added by hand.

how the property is to be transported from a Microbase server to a worker node. For instance, the `e value` input is annotated with an `@InputProperty`, while the two genome file inputs are annotated with `@InputResource`. The alignment file produced by running BLAST is annotated as being an `@OutputResource`. The BLAST executable itself is annotated with `@PlatformSpecificResource`, indicating that the worker node should decide at run-time which version of a resource it should download. The string values embedded within the annotations allow Microbase to bridge its systems with domain job implementations. The details of annotations will be revisited in the following sections.

A.5.2.3.2 Performing computational work A worker node executes your job implementation by calling the `doWork()` method. A job's specified input properties are guaranteed to be populated before a the `doWork()` method is called (assuming suitable values were received from a job server). Command line applications such as BLAST will also be downloaded and installed automatically prior to execution.

There are two ways in which the `doWork()` method may terminate: either cleanly, by returning normally; or by throwing a `JobProcessingException`. If the method returns cleanly, without throwing an exception, then the worker node assumes that the job processing was successfully completed, and informs the Microbase system appropriately. However, if the `doWork` terminates by throwing an exception, the worker node assumes that a processing failure occurred. The Microbase system is informed of the failure, and is passed the stack trace for logging and debugging purposes. In the failure case, the job will also be re-queued so that it will run at a later time, potentially on a different worker node.

The implementation of the `doWork` method required to run BLAST needs to generate a command line, using the available parameters. This is shown in [Figure A.5 on the next page](#).

All command line processes have the ability to output to either or both of the two standard UNIX streams: standard out ([STDOUT](#)) and standard error ([STDERR](#)), whether they are executing on Linux, Windows, or other platforms. Whether these streams contain useful content is entirely application-dependent. For the purposes of this guide, the content of these streams will be sent to the screen and will be displayed when the job runs on a worker node. For 'real world' jobs, you may wish to capture this information by writing it to a file (handily, the Java `FileWriter` class implements `Appendable`, so this is trivial), or you may want to ignore the streams completely if no useful information is returned.

The Java implementation for the compute job is now complete. On completion of `doWork`, the content of the `alignmentOutput` file will be copied back to the Microbase resource system.

A.5.2.4 Packaging platform-native applications

For this example, we will package the BLAST application for two platforms: Linux/i386, and Windows/i386, although only the Linux package will actually be used for the purposes of this guide.

You can find BLAST distributions for various operating systems under `$HOME/blast`. This directory contains unmodified BLAST distributions, as downloaded from the NCBI site.

First, change the working directory to that of the Linux BLAST distribution:

```

@Override
protected void doWork() throws JobProcessingException
{
    try
    {
        System.out.println("Starting BLAST ... ");

        //For this example, we'll just send STDOUT and STDERR to the screen
        Appendable stdout = System.out;
        Appendable stderr = System.out;

        /*
         * We need to specify an output filename for BLAST to use. This can be
         * named anything, but should be created within the temporary working area
         * assigned to this job execution, as returned by getWorkingDirectory()
         */
        alignmentOutput = new File(getWorkingDirectory(), "alignment_output.txt");

        //Runs BLAST, using the input files received from Microbase
        blastExePkg.executeApplication(stdout, stderr, "bl2seq",
            "-p", "blastn",
            "-e", String.valueOf(eValue),
            "-i", firstSequence.getAbsolutePath(),
            "-j", secondSequence.getAbsolutePath(),
            "-o", alignmentOutput.getAbsolutePath());

        System.out.println("Finished... ");
    }
    catch (NativeExecutableException e)
    {
        throw new JobProcessingException(
            "Failed to run BLAST...", e);
    }
}

```

The parameters passed to this method are as follows:

- The first two parameters specify where the ‘standard’ STDOUT and STDERR streams are directed to.
- The third parameter corresponds to a key value in a map. This value does not correspond to an executable file on a disk (although it may be named similarly). Instead, the string ‘bl2seq’ is a platform-neutral name assigned to the application that gets mapped at run-time to a platform-specific command path. This allows the Java job implementation to run an application without having to know the exact location, or for that matter the platform-specific filename (e.g., ‘bl2seq’ on Linux vs ‘bl2seq.exe’ on Windows). Command name \mapsto path mappings will be explained in the following Section A.5.2.4 on the preceding page.
- The remaining parameters form command line parameters passed to the application, equivalent to the BLAST command line described earlier (Section A.5.2.1 on page 223).

Figure A.5: How to run a native executable application from within a Java compute job implementation. The executeApplication() method on the NativeExecutable instance delegates processing to a command line application.

```
cd ~/blast/linux-blast
```

You should see a directory structure that resembles something like this:

```
blast-2.2.18
|-- bin
|-- data
'-- doc
```

Under the ‘bin’ directory, you should see:

```
|-- bin
|   |-- bl2seq
|   |-- blastall
|   |-- blastclust
|   |-- blastpgp
|   |-- copymat
|   ... etc ...
```

Step 1 - create the executable path name properties file There are several executable files within the ‘bin’ directory. Currently, however, these files are not accessible from the job implementation because Microbase does not know that, for instance, `bl2seq` is an executable file. To run one of these files from a job implementation, it is necessary to explicitly specify which file(s) are required. This can be done by constructing a standard Java properties file. This file must be called `exe_mappings.properties`. This file will map an abstract name to an operating system-specific file path. The name field is the identifier used by the Java job implementation at run-time to recognise an executable file. This name does not need to mirror the actual executable file name, but it is advisable that it is similar for readability and consistency reasons.

```
bl2seq = blast-2.2.18/bin/bl2seq
blastall = blast-2.2.18/bin/blastall
blastclust = blast-2.2.18/bin/blastclust
... and so on ...
```

Note that not every executable file under the ‘bin’ directory needs to be added to the mappings file. Only the programs called by the Java job implementation need to be entered. So for this example, only the `bl2seq` entry actually needs to be present.

Step 2 - create the resource file Once the name mappings file is complete, a package can be created that can be used by Microbase. The following command will create such a package. It will create a file “`mb-blast-2.2.18-linux-ia32.zip`”, that contains the original blast distribution, as well as the mapping file created in the previous step.

```
zip -r mb-blast-2.2.18-linux-ia32.zip exe_mappings.properties blast-2.2.18
```

Step 3 - create the resource tag file Files stored within the Microbase resource system may have meta-data associated with them in the form of tags (key/value pairs). Worker nodes can use these tags at run-time to find platform-native resources that match the particular platform they are running on. Therefore, the developer is required to provide these tags in an appropriately named file. The

Microbase installer will use this file at install-time to deploy the resource file created in **step 2** with the appropriate tags.

The resource tag file should be named: `<resource_file_name>.resource.tag.properties`, so in this case, the tag file name will be: `mb-blast-2.2.18-linux-ia32.zip.resource.tag.properties`

The tag file content should be:

```
res.name=blast
res.version=2.2.18
res.file_type=ZIP
res.file_content=PLATFORM_SPECIFIC_RESOURCE
platform_specific.os_name=Linux
platform_specific.os_arch=i386
```

Notes:

1. The values you use for the `res.name` and `res.version` tags are arbitrary. You just need to ensure that the string values specified in the properties file are the same as the string values specified in the `InputPlatformSpecificResource` annotation, within the Java job implementation.
2. The tag `platform_specific.os.name` needs to be set to the name of the operating system that the native executable(s) run on as returned by `System.getProperty('os.name')`.
3. The tag `platform_specific.os.arch` needs to be set to the architecture that the native executable(s) run on as returned by `System.getProperty('os.arch')` Note that this varies according to the operating system. For instance, Linux reports intel ia32 hardware as “i386”, whereas Windows reports the same hardware as “x86”.
4. The values for the tags `res.file_type` and `res.file_content` should not be changed.

Step 4 - copy files into project directory structure Finally, the files “mb-blast-2.2.18-linux-ia32.zip” and “mb-blast-2.2.18-linux-ia32.zip.resource.tag.properties” should be copied into the `mb-resources` directory of the job implementation Maven project. This will allow the resource to be found and published by the Microbase installer.

This can be accomplished by executing:

```
cp mb-blast-2.2.18-* $HOME/my-new-responder/my-compute-job/src/main/mb-resources
```

Troubleshooting If you encountered any problems with application packaging in the steps above, there is a pre-packaged BLAST file available in `$HOME/blast/prepackaged`. The content and layout of this archive file may be helpful.

Packaging the Windows-native executable

Note: for the purposes of this tutorial, packaging a Windows executable is not necessary. It is here for reference only, to highlight the differences to packaging for Linux/i386.

The process of creating a BLAST package for the Windows/32 platform is identical. Only the platform specific values change. This part is not strictly necessary for this tutorial (unless you wish to test the job implementation on Windows). For a ‘real world’ deployment, the process of creating platform-specific packages would need to be repeated for each platform.

Step 1 - create the executable path name properties file Under Windows, executable file names have the extension “.exe”:

```
|-- bin
|  |-- bl2seq.exe
|  |-- blastall.exe
|  |-- blastclust.exe
|  |-- blastpgp.exe
|  |-- copymat.exe
|  ... etc ...
```

Therefore, the Windows `exe_mappings.properties` file will look like this:

```
bl2seq = blast-2.2.18/bin/bl2seq.exe
blastall = blast-2.2.18/bin/blastall.exe
blastclust = blast-2.2.18/bin/blastclust.exe
... and so on ...
```

Step 2 - create the resource file Again, run the zip command to create the resource file:

```
zip -r mb-blast-2.2.18-windows-ia32.zip exe_mappings.properties blast-2.2.18
```

Step 3 - create the resource tag file Next, the resource tag file needs to be created. The differences are the operating system-specific tag values:

```
res.name=blast
res.version=2.2.18
```

```

res.file_type=ZIP
res.file_content=PLATFORM_SPECIFIC_RESOURCE
platform_specific.os_name=Windows\ XP
platform_specific.os_arch=x86

```

A.5.2.5 Final job implementation directory

The job implementation directory structure should look like the structure shown as follows. If it does, proceed to the next section.

```

my-new-responder/
|-- my-compute-job
|   |-- pom.xml
|   '-- src
|       '-- main
|           |-- java
|               |-- uk
|                   '-- ac
|                       '-- ncl
|                           '-- mygroup
|                               '-- HelloJob.java
|                                   |-- mb-resources
|                                       |-- mb-blast-2.2.18-linux-ia32.zip
|                                           |-- mb-blast-2.2.18-linux-ia32.zip.resource.tag.properties
|                                               |-- test.foo
|                                                   '-- resources
'-- pom.xml

```

A.5.3 Event handler sub-project

The previous section introduced job implementations that run within a worker node. This section will describe the server-resident, event handler part of a responder that is responsible for scheduling instances of the job implementations. The event handler is a web-service implementation of the Microbase notification system's push subscriber. This project therefore builds as a web archive (.war) and will need to be hosted within a container such as Tomcat.

You can create a new skeleton event handler project by executing the following Maven command from *within* the "my-new-responder" directory. After executing the command, the responder directory should look similar to the directory structure shown in [Figure A.6 on the following page](#):

```

mvn archetype:create \
    -DarchetypeGroupId=uk.org.microbase \
    -DarchetypeArtifactId=event-quickstart \

```



```

-DarchetypeVersion=1.0 \
-DgroupId=uk.ac.ncl.mygroup \
-DartifactId=my-event-handler \
-Dversion=1.0

```

```

my-new-responder/
|-- my-compute-job
|   |-- pom.xml
|   '-- src
|       '-- main
|           |-- java
|           |   '-- uk
|           |       '-- ac
|           |           '-- ncl
|           |               '-- mygroup
|           |                   '-- HelloJob.java
|           |-- mb-resources
|           |   |-- mb-blast-2.2.18-linux-ia32.zip
|           |   |-- mb-blast-2.2.18-linux-ia32.zip.resource.tag.properties
|           |   '-- test.foo
|           '-- resources
|-- my-event-handler
|   |-- pom.xml
|   '-- src
|       '-- main
|           |-- java
|           |   '-- uk
|           |       '-- ac
|           |           '-- ncl
|           |               '-- mygroup
|           |                   '-- HelloEventResponder.java
|           |-- resources
|           |   '-- META-INF
|           |       '-- xfire
|           |           '-- services.xml
|           '-- webapp
|               '-- WEB-INF
|                   '-- web.xml
'-- pom.xml

```

Figure A.6: A responder project containing two sub-projects: a job implementation and an event handler

As you can see, this has created a web application project that is an XFire Web service implementation. When built, this project will generate a web archive (.war file) that can be deployed to a Tomcat (or equivalent) application container. Load this new project into NetBeans.

A.5.3.1 Implementing the event handler

A.5.3.1.1 Registering to receive event notifications Event handlers need to be able to receive events from other components within the Microbase system in the form of messages. These messages may be notifications of new data arriving in the system, or notification that a computational task has completed. Messages can be used to chain multiple responders together into a pipeline.

Although Microbase handles automatic registration of your event handler implementation (such as its Web service end-point and message topic registrations), the developer is responsible for specifying which types of message their responder should receive. This is done by requesting that the responder is registered with a set of topic names. The responder described in this section needs to receive two types of event:

1. Notification of 'new' data files to process
2. Notification of the completion of a BLAST job.

The responder needs to send two types of event:

1. A request to Microbase to schedule a computationally intensive task
2. A message sent in response to a task completion report (in a 'real world' system, this would be used to inform the next responder in a pipeline that a BLAST task has just completed).

In the NetBeans IDE, open the Java file `HelloEventResponder`. You should see some example topic registrations in the constructor of this class. Change the constructor to read:

```
public HelloEventResponder()
    throws ConfigurationException
{
    setIncomingTopicIds("new sequence data",
        MicrobaseWellKnownTopics.TASKSYS__TASK_REPORT);
    setOutgoingTopicIds(MicrobaseWellKnownTopics.TASKSYS__TASK_DESCRIPTOR,
        "blast complete");
}
```

The first line informs Microbase that this event handler should receive events of type 'new sequence data', and a Microbase event that indicates task completion. The second line informs Microbase that the responder will send messages of type 'blast complete'. Microbase will handle the registration of these topics within the notification system, and will also register the responder as a subscriber to these messages.

A.5.3.1.2 Handling event notifications At this point, the class is a valid event handler and would receive events if deployed to Microbase. However, it currently does nothing with these events. Events are handled with the `dealMessage()` method. All events received by the handler are delivered via this method. Before events can be handled, they need to be filtered by type because different types of messages must be handled in different ways. We will use the `dealMessage()` method to split messages based on their topic, and delegate onto appropriate handler methods:

```
@Override
protected void dealMessage(MessageLogItem messageItem)
    throws UnrecoverableException, TransientException
{
    try
    {
        if (messageItem.getTopicId().equals("new sequence data"))
        {
            handleNewDataAvailable(messageItem);
        }
        else if (messageItem.getTopicId().equals(
            MicrobaseWellKnownTopics.TASKSYS__TASK_REPORT))
        {
            handleTaskCompletionReport(messageItem);
        }
    }
    catch(Exception e)
    {
        throw new TransientException("Message processing failed", e);
    }
}
```

A handler method for ‘new sequence data’ notifications will now be written. What does a ‘new sequence data’ message look like? It’s entirely up to the responder developer. In a production-quality system, this would usually involve an XML-based message, containing appropriate domain-specific meta-data. In this example, the message content will be a human-friendly space-separated list of sequence names. The message content will need to be parsed, and the names extracted. It will then be necessary to schedule an appropriate number of jobs so that an all-vs-all comparison is performed. Each job will perform a single pairwise comparison between two sequences.

```

protected void handleNewDataAvailable(MessageLogItem messageItem)
    throws FailedToPublishException
{
    //Parse the message content as a series of space-sparated sequence names
    String content = new String(messageItem.getMessageBody());
    String[] names = content.split(" ");

    //Job implementation details
    String mvnGroupId = "uk.ac.ncl.mygroup";
    String mvnArtifactId = "my-compute-job";
    String mvnVersion = "1.0";
    String exeClass = "uk.ac.ncl.mygroup.HelloJob";

    //Construct a task (this is the container of a set of related jobs)
    TaskDescriptionMessage task = new TaskDescriptionMessage();
    task.setTaskId(UidGenerator.generateUid()); //Assign a unique task id
    task.setTaskType("Blast"); //Assign a unique, but human-readable type name

    // Generate a set of jobs based on input data
    for (String firstSequence : names)
    {
        for (String secondSequence : names)
        {
            //Create a job description that will compare two sequences
            JobDescriptorMessage job = new JobDescriptorMessage();
            job.setMavenGroupId(mvnGroupId);
            job.setMavenArtifactId(mvnArtifactId);
            job.setMavenVersion(mvnVersion);
            job.setRunnableClass(exeClass);
            job.setUid(UidGenerator.generateUid()); //Assign a unique job id

            job.addInputParameter("eValue", String.valueOf(0.00058));

            job.addInputResourceId("sequenceOne", firstSequence);
            job.addInputResourceId("sequenceTwo", secondSequence);

            job.addOutputResourceName("alignment");

            //Add the new job to the task container
            task.addJob(job);
        }
    }

    //Send the task request to the notification system
    publishMessage(
        MicrobaseWellKnownTopics.TASKSYS_TASK_DESCRIPTOR,
        new String[0], BeanSerialisationTools.serialise(task));
}

```

New data notifications are dealt with as follows:

1. The content of the message is parsed to obtain a list of sequence names that need to be compared.
2. The sequences are to be compared in an all against all fashion. It is necessary to schedule a series of compute jobs implemented in Section A.5.2 on page 223 to execute each pairwise comparison. Since the event handler and compute job implementation are separate decoupled projects, the event handler needs a way to reference the job implementation. This is achieved by using the Maven project information specified when the compute job project was created (Section A.5.2.2 on page 225), specifically group id, archetype id, version. In addition, the fully-qualified class name containing the job implementation is required. This information enables a remote worker node to find, download and install the required compute job at run-time.

3. For efficiency reasons, sets of related job descriptions are bundled into a ‘task’ (the task is simply a container).
4. The next code section loops over the sequence names and creates job ‘descriptions’ for each pair of sequences. It is important that the input parameters and input resource names specified here match the annotation strings specified in the job implementation (see Figure A.4 on page 229). The newly created jobs are added to the ‘task’ container.
5. Finally, the task description is published as a notification. The Microbase notification system propagates the task message to the job management system, where the jobs will be queued until suitable worker node(s) are available to process them.

On completion of the task (i.e., all jobs complete), a notification will be sent back to the BLAST responder. The responder must handle this event as well. In this case, a simple ‘blast complete’ notification will be sent. Currently this message is not used by component. However, if another responder were added to the system at a future time, the ‘blast complete’ message history could be used to inform the new responder of previously completed work. The code fragment in below shows how this is achieved. This demonstration application emits human-readable ‘blast complete’ messages in response to a ‘task complete’ message being received from the job management system. These messages are generated in the following way:

1. A set of job execution reports are obtained from the ‘task complete’ message.
2. A human-readable summary is produced by iterating through these job execution reports including: inputs used, output resource produced, and whether the job run was successful or not. If the enactment was unsuccessful, an execution log is appended.
3. Finally, the ‘blast complete’ message is published to the notification system.

```

protected void handleTaskCompletionReport(MessageLogItem messageItem)
    throws FailedToPublishException
{
    TaskJobsCompleteMessage taskReport = (TaskJobsCompleteMessage)
    BeanSerialisationTools.deserialize(messageItem.getMessageBody());
    if (!taskReport.getTaskType().equals("Blast"))
    {
        return;
    }

    StringBuilder completionMessage = new StringBuilder();
    for (ExecutionLogItem jobExe : taskReport.getFinalJobExecutionReports())
    {
        String firstSeqId = jobExe.getInputResourceIds().get("sequenceOne");
        String secondSeqId = jobExe.getInputResourceIds().get("sequenceTwo");

        String alignmentId = jobExe.getOutputResourceIds().get("alignment");

        completionMessage.append("Alignment of ");
        completionMessage.append(firstSeqId);
        completionMessage.append(" against ");
        completionMessage.append(secondSeqId);
        if (jobExe.getState() == JobExecutionStates.COMPLETED_SUCCESS)
        {
            completionMessage.append(" succeeded, output resource: ");
            completionMessage.append(alignmentId);
        }
        else
        {
            completionMessage.append(" failed...\n");
            completionMessage.append(jobExe.executionLogToString());
        }
        completionMessage.append("\n");
    }
    //Finally, publish the message
    publishMessage("blast complete", completionMessage.toString().getBytes(),
        messageItem.getMessageId());
}

```

A.5.3.2 Modifying services.xml

At this point, the event handler should be capable of handling events from the Microbase notification system. However, one small configuration detail needs to be completed before the project can be compiled.

In the filesystem tree (above), you should be able to find a file named `services.xml`. Currently this file looks something like this:

```

<beans xmlns="http://xfire.codehaus.org/config/1.0">
    <service>
        <name>PushSubscriber</name>
        <serviceClass>uk.org.microbase.notification.ws.subscriber.push.spec.PushSubscriber</serviceClass>
        <implementationClass>HelloEventResponder</implementationClass>
    </service>
</beans>

```

The `<implementationClass>` line needs to be changed from this:

```

<implementationClass>HelloEventResponder</implementationClass>

```

to this:

```

<implementationClass>uk.ac.ncl.mygroup.HelloEventResponder</implementationClass>

```

The package name needs to be added because the event handler Maven archetype currently does not support inserting the group id / package name into XML files. The `<implementationClass>` line should *always* mirror the fully-qualified Java classname of the event handler implementation class. Bear this in mind if you re-factor the event handler project.

At this point, the entire responder has been implemented and is ready to be deployed.

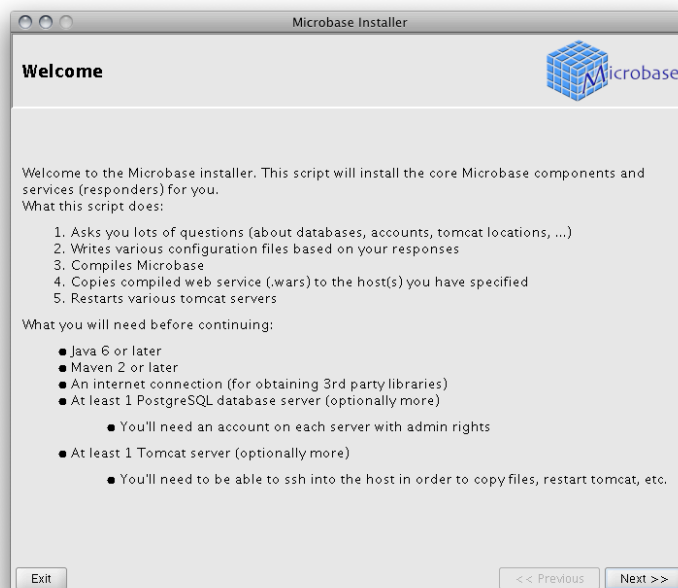
A.6 Installation / Deployment

This section assumes that you have a completed responder, ready for compilation and deployment to a Microbase installation. The Microbase installer is capable of installing responders as well as the core Microbase components. If you are using the virtual machine image, the Microbase core components will already have been installed, and you can skip many of the installation steps.

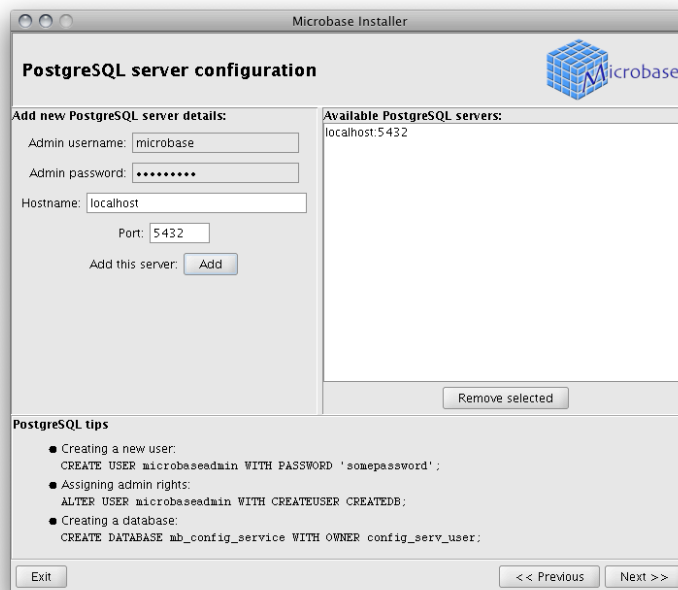
To run the installer, open the terminal session named `Microbase build`. In the tab named `installer`, run the following script:

```
./installer.sh
```

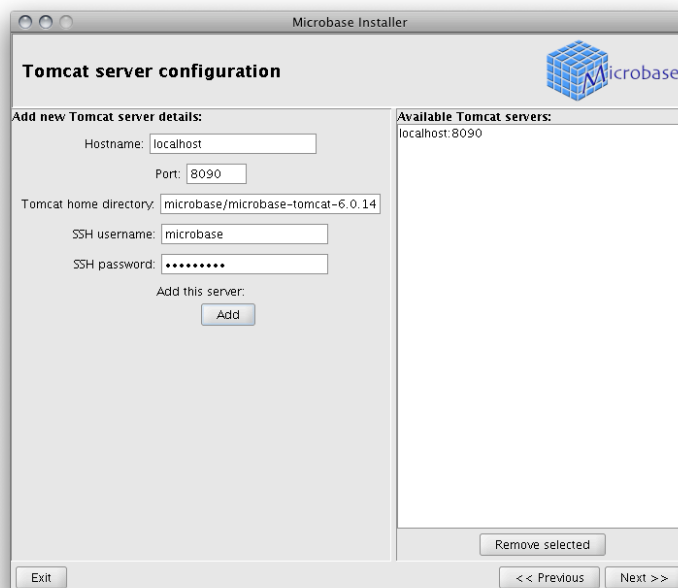
You should now be at the “welcome” screen of the GUI installer. The VMware image used for this guide already has many of the settings configured for you. For instance, you can simply click through the PostgreSQL and Tomcat configuration screens. They are shown below for clarity.



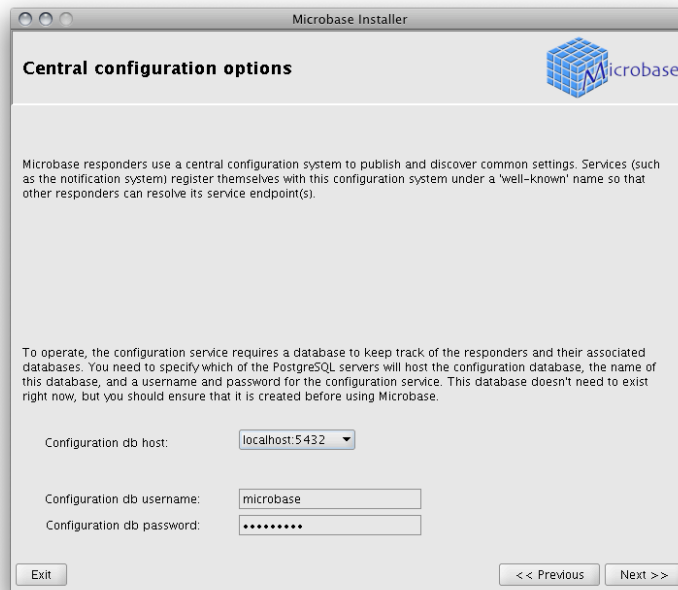
Installer step 1: if you see this screen, then the installer has compiled and started successfully.



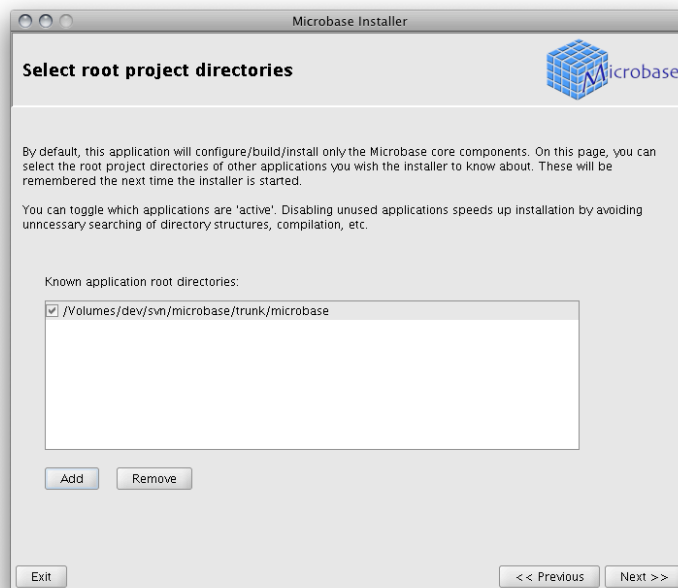
Installer step 2: Configuration of the PostgreSQL server(s) that will store data for the core Microbase components, and potentially user responder components. *For this tutorial, leave these options set to the defaults*



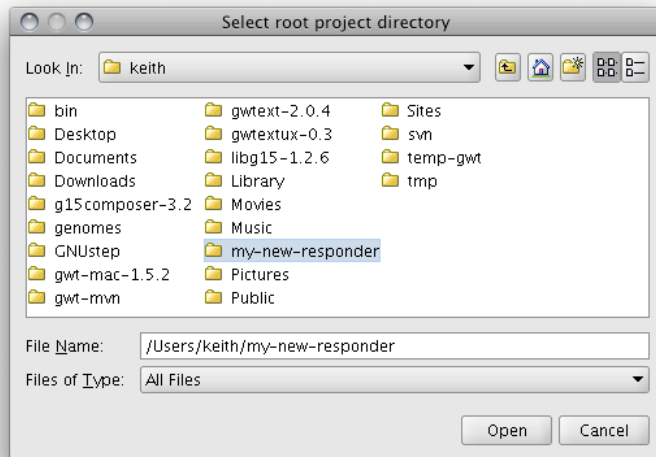
Installer step 3: Configuration of the Tomcat server(s) that will host the server-resident portions of responders. *For this tutorial, leave these options set to the defaults*



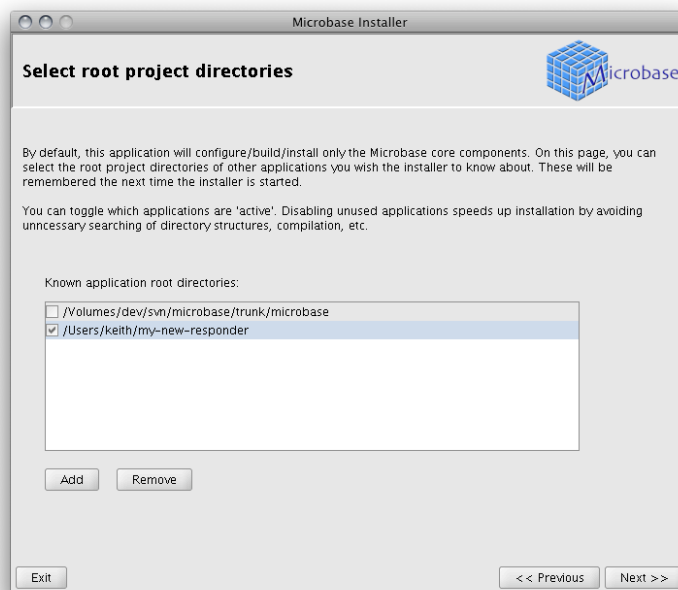
Installer step 4: Database server selection for the configuration service. *For this tutorial, leave these options set to the defaults*



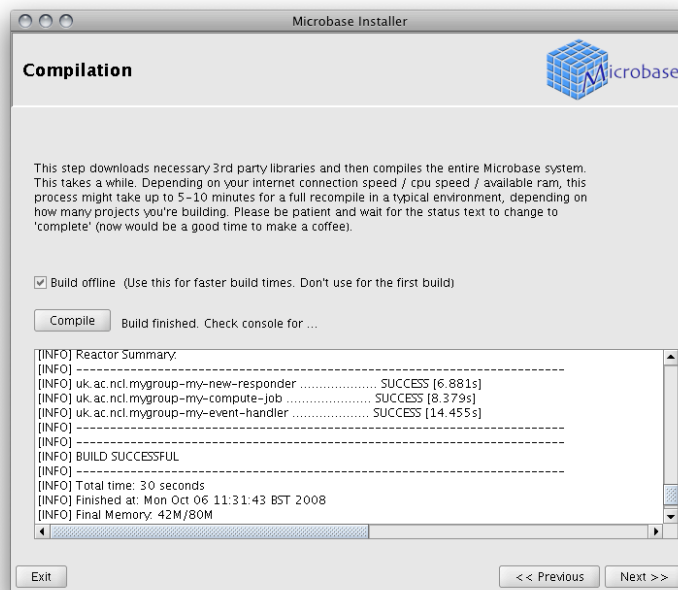
Installer step 5a: Project selection. Here, you can specify the project directory for your responder. This is required because Microbase takes over the compilation and deployment of your responder. Microbase can only do this if it knows the location of your responder project.



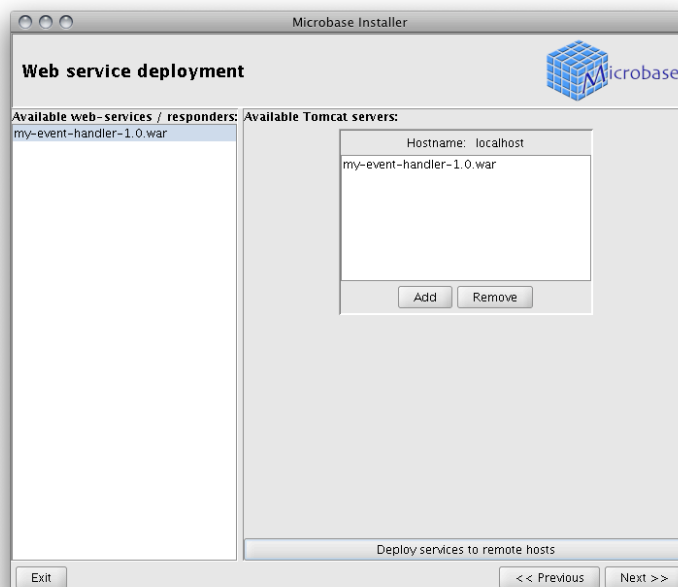
Installer step 5b: Project selection. Select the responder's root directory



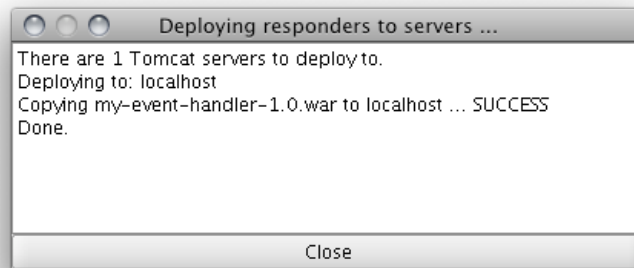
Installer step 5c: Project selection. Make sure that your responder is checked. If not checked, the installer will ignore the project. Make sure that the Microbase project is unchecked. Recompiling Microbase would do no harm, but it is unnecessary in this case, and may take several minutes to compile.



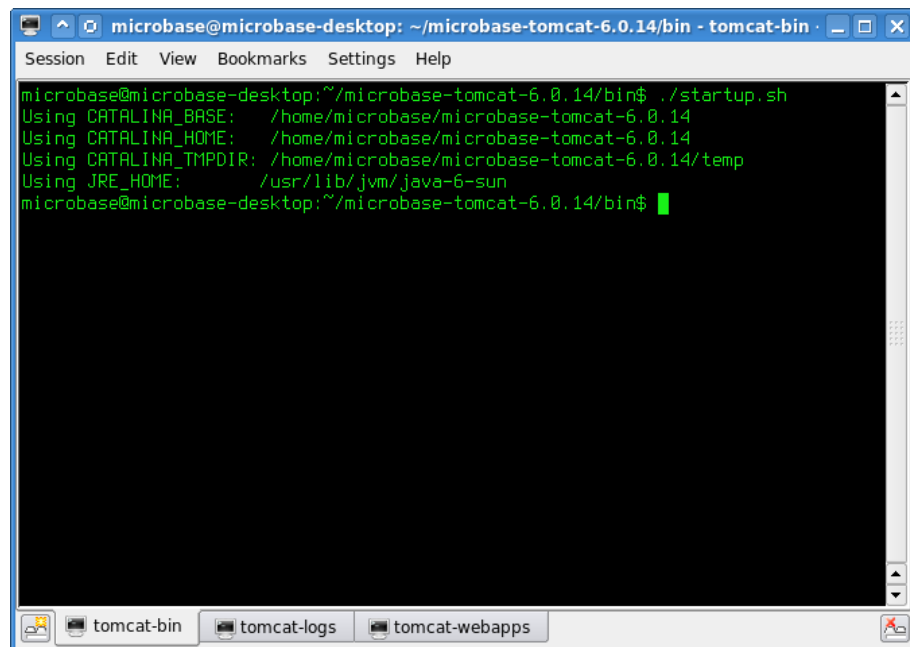
Installer step 6: Compilation. Press the 'compile' button. You should hopefully see a 'BUILD SUCCESSFUL' message. If so, proceed to the next step.



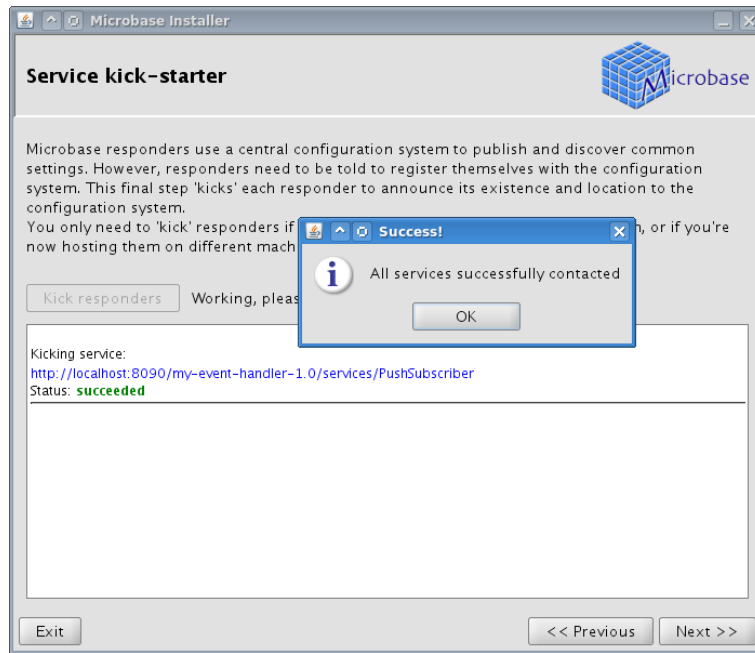
Installer step 7a: Deployment. This step deals with deployment of server-side components to a set of remote Tomcat servers. If more than one Tomcat server is defined in step 3, then multiple Tomcat servers will be available to deploy to. In this case, the Tomcat server is simply 'localhost' because it is running within the virtual machine. Select your responder in the left-hand pane. 'Add' it to the Tomcat server listed on the right hand side. Finally, click the 'deploy' button at the bottom of the screen. You may ignore the Tomcat warning here, since Tomcat has not been started yet.



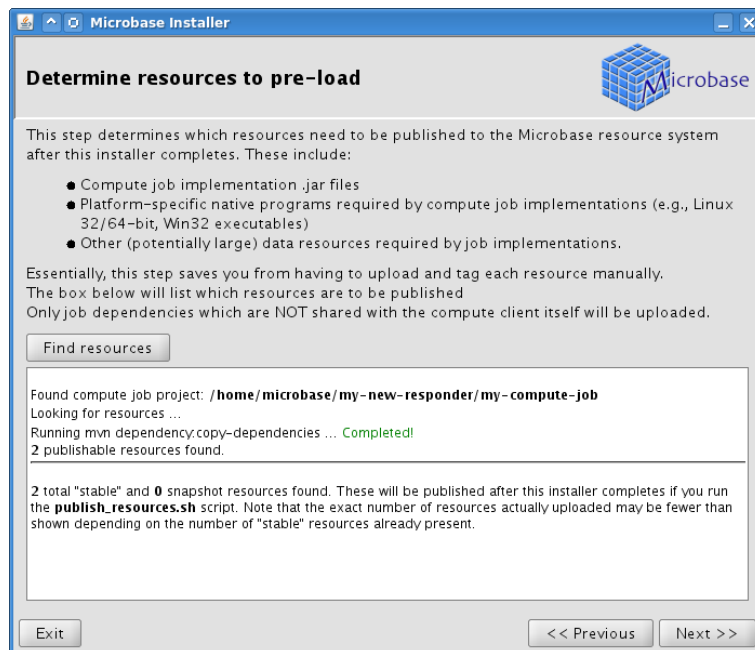
Installer step 7b: Deployment complete. This pop-up window should appear during service deployment. The image shown here indicates that all services were copied correctly.



Installer step 7c: Start Tomcat. At this point, we need to step outside of the installer GUI for a moment in order to start the Tomcat server. On the desktop of the virtual machine, you should see a ‘Tomcat’ terminal icon. Open the terminal, and type `./startup.sh`. This command should have started the server. You can monitor the startup process if you wish by changing to the ‘log’ terminal and typing `tail -f catalina.out`.

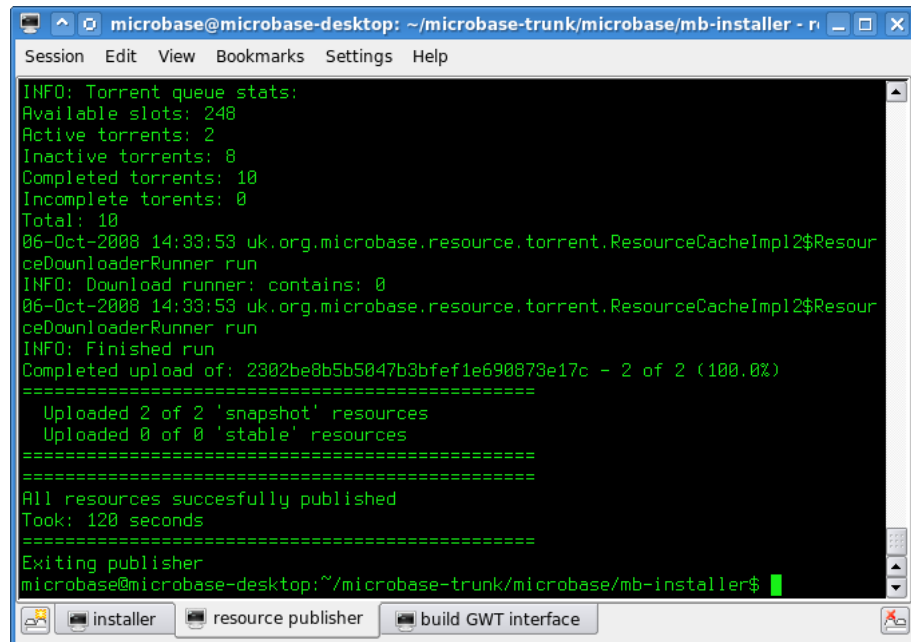


Installer step 8: ‘Kick’ responders. The last step started the Tomcat server. However, it is now necessary to ‘kick’ each of the deployed responders so that they initialise and self-register. The kick-start process allows responders to auto-create the databases they need.



Installer step 9a: Find compute job implementations. It is necessary to deploy the compute-job portion of responders to the Microbase resource system. Compute job ‘jar’ files, as well as their Java dependencies and required command line applications need to be copied to the resource system so that they are accessible to worker nodes. Completing this task allows worker nodes to automatically install applications on-demand. Click the ‘find resources’ button to instruct the

installer to find all compute-job implementation projects.



```
microbase@microbase-desktop: ~/microbase-trunk/microbase/mb-installer - n
Session Edit View Bookmarks Settings Help
INFO: Torrent queue stats:
Available slots: 248
Active torrents: 2
Inactive torrents: 8
Completed torrents: 10
Incomplete torrents: 0
Total: 10
06-Oct-2008 14:33:53 uk.org.microbase.resource.torrent.ResourceCacheImpl2$ResourceDownloaderRunner run
INFO: Download runner: contains: 0
06-Oct-2008 14:33:53 uk.org.microbase.resource.torrent.ResourceCacheImpl2$ResourceDownloaderRunner run
INFO: Finished run
Completed upload of: 2302be8b5b5047b3bfef1e690873e17c - 2 of 2 (100.0%)
=====
  Uploaded 2 of 2 'snapshot' resources
  Uploaded 0 of 0 'stable' resources
=====
All resources succesfully published
Took: 120 seconds
=====
Exiting publisher
microbase@microbase-desktop:~/microbase-trunk/microbase/mb-installer$
```

Installer step 9b: Deploy compute job implementations: Finally, the resources found in the previous step need to be deployed. Unfortunately, it is necessary to go back to the command line again. Switch to the command line window that you started the GUI installer from. Select the 2nd tab, ‘publish resources’. Type the command `./publish_resources.sh`. This script takes the list of resources generated by the GUI installer and copies them to the Microbase resource system via BitTorrent. After this step completes, then your responder is installed and is ready to test.

Note that this final step depends on an active Internet connection to allow the BitTorrent implementation’s distributed database to connect. No torrent data is transferred externally, only lightweight connections to a BitTorrent tracker are made. Please be patient, it may take a minute or two to initiate connections to this external service. For performance reasons in a production-quality deployment, it would be advisable to connect to a torrent tracker running somewhere on your local network. This is relatively straightforward, but is beyond the scope of this tutorial.

A.7 Testing

This section shows how to test the new responder. Before proceeding, it might be worth closing the NetBeans IDE and the Microbase installer if they are still open. If the virtual machine image is running in 600MB (the default), this will free up a substantial amount of memory and will improve performance.

Before jobs can be scheduled, some input data (genome sequences) needs to be uploaded to the resource system. Usually, this would be done automatically by Microbase (by scanning an FTP site, for instance). To upload some sequences, open the ‘Testing’ console icon on the desktop, and run the following commands:

```
./resource-client.sh publish NC_000964 some_description $HOME/genomes/NC_000964.fna
./resource-client.sh publish NC_002570 some_description $HOME/genomes/NC_002570.fna
./resource-client.sh publish NC_003997 some_description $HOME/genomes/NC_003997.fna
```

Each command uploads one genome sequence to the resource system. Once all the data has been uploaded, a notification must be sent to inform the system that new data is available (this is the notification message we defined in Section A.5.3.1 on page 237). In a production system, this notification would be sent automatically.

To send the notification, open Firefox (desktop icon). In the bookmark toolbar, click the ‘Microbase GUI’ entry. From the ‘Applications’ menu, choose ‘Notification admin’. You should see a window similar to Figure A.7.

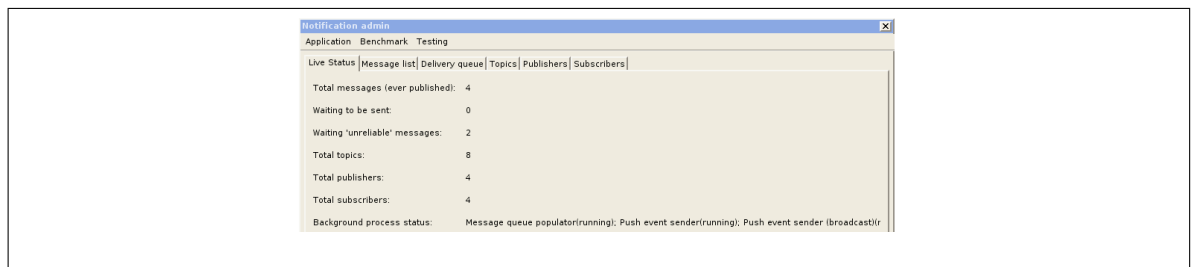


Figure A.7: Notification system administration interface. This can be used for monitoring a running system, as well as injecting messages for testing responders. The ‘message list’ tab allows paging through all archived messages. Double-clicking an item will display the message content in a pop-up window.

Click ‘Testing’ and ‘Send message’. From the drop-down list of topics, select ‘new sequence data’. From the drop-down list of publishers, choose any publisher *except* ‘HelloEventResponder’ (for the purposes of this guide, it does not matter which publisher sends the ‘new sequence data’ message).

In the ‘content’ box, add the following space-separated sequence names ‘NC_000964 NC_002570 NC_003997’ (Figure A.8 on the following page)

Finally, click ‘send’. This sends the notification of ‘new data’, which should arrive at your responder. The responder will then schedule a set of jobs to perform the all-vs-all comparison of the specified genome sequences. These jobs will arrive at the job server, ready for processing.

Currently, the web interface to Microbase is undergoing a re-write. Therefore, to see the jobs added to the system, you will need to browse to ‘Microbase GUI2’ available on the bookmarks toolbar of Firefox. Once there, open the ‘job server’ application (Figure A.9 on the next page).

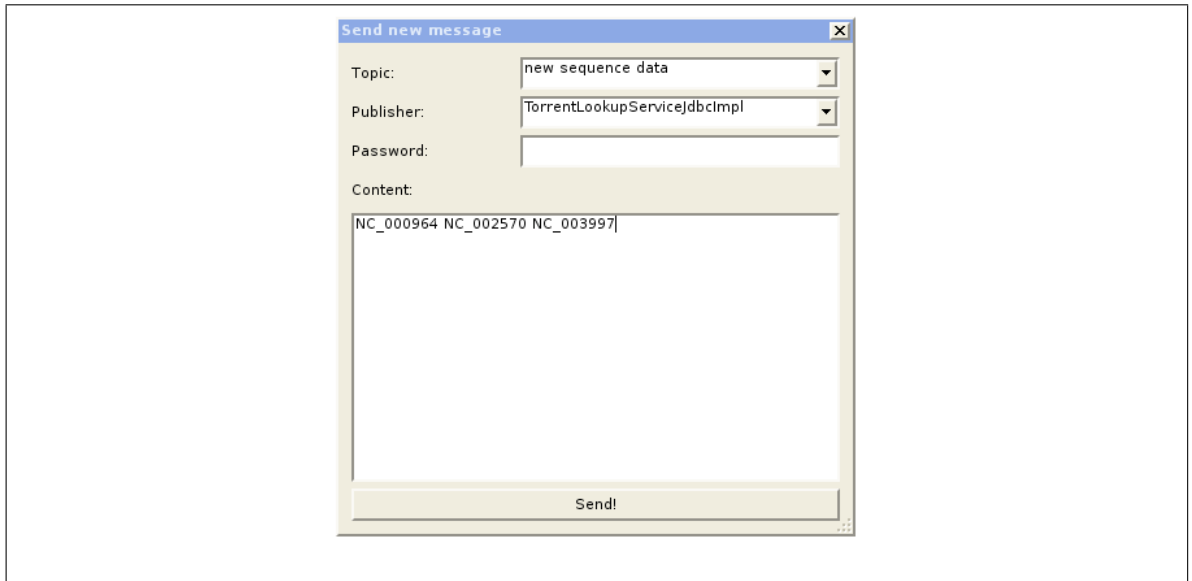


Figure A.8: Sending a test message

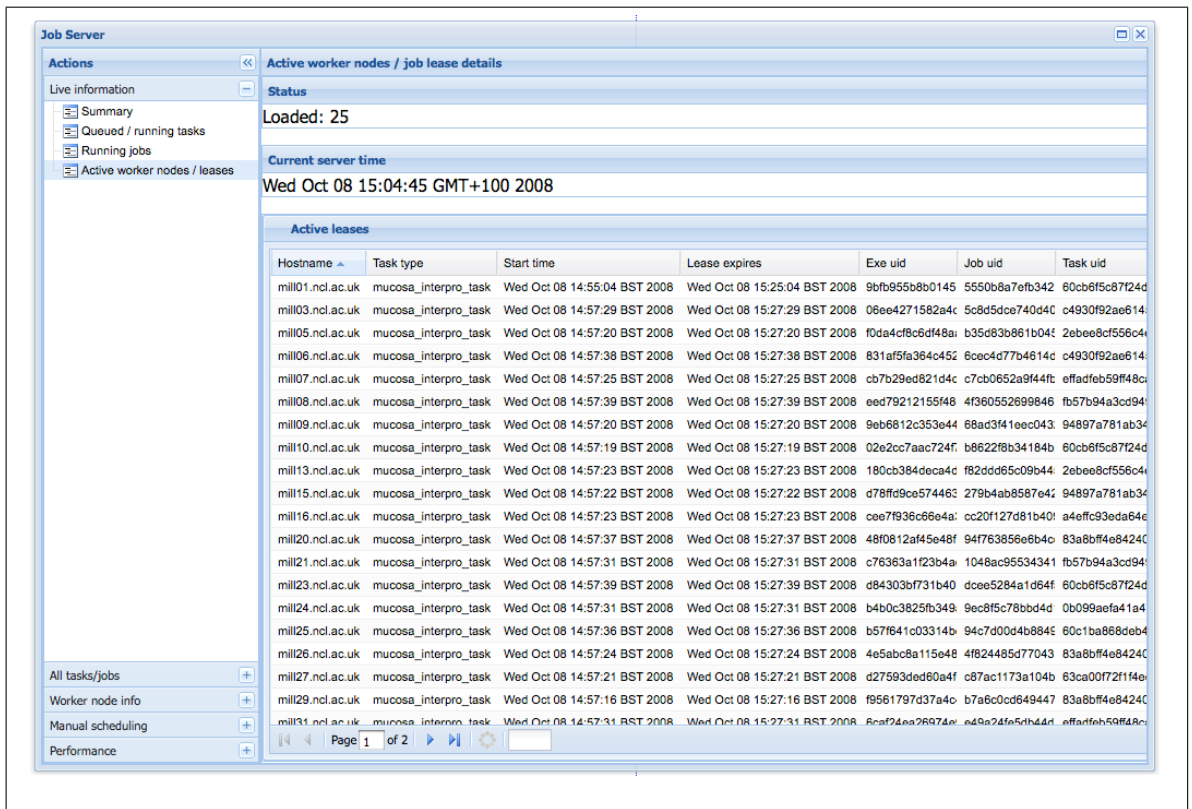


Figure A.9: The job server GUI shows currently queued, processing and completed jobs. In this case, a large number of jobs are running over a cluster of worker nodes.

To run jobs, the Microbase compute client needs to be started. Usually, this would be started via an automated means such as Condor or Sun Grid Engine. For this demonstration it needs to be started manually. Change to the 'job enactment' tab of the 'Testing' console and type the following:

```
./node_start_small_ram.sh --normal
```


After a while the compute node will start. The following sequence of events should occur:

1. The worker will contact the job server (running within Tomcat) for a job description.
2. The required job implementation (jar) will then be downloaded.
3. After class-loading and introspecting the job implementation class, the worker node will download the required resource files (in this case, the Linux/x86 BLAST executable, and two genome files).
4. The BLAST executable will then be installed, and processing should begin — check the GUI within Firefox for ‘processing’ jobs. You might also want to have `top` (see Figure A.10) running within a terminal to see `bl2seq` running.
5. Once the job is complete, Microbase is informed that the resulting alignment file is available for archiving.
6. Simultaneously, the next job description is obtained, and processing begins.
7. Once all jobs have been processed, the compute client will remain running (it can be killed with CTRL-C).
8. The task should be marked ‘complete’, and a notification event will be sent to indicate this.

Finally, if you wish to inspect the data produced, every resource stored by the resource system is located in `$HOME/data/torrents`.

If you had any problems implementing the responder, a complete ready-to-compile version is available in `$HOME/microbase-trunk/microbase-tutorial`.

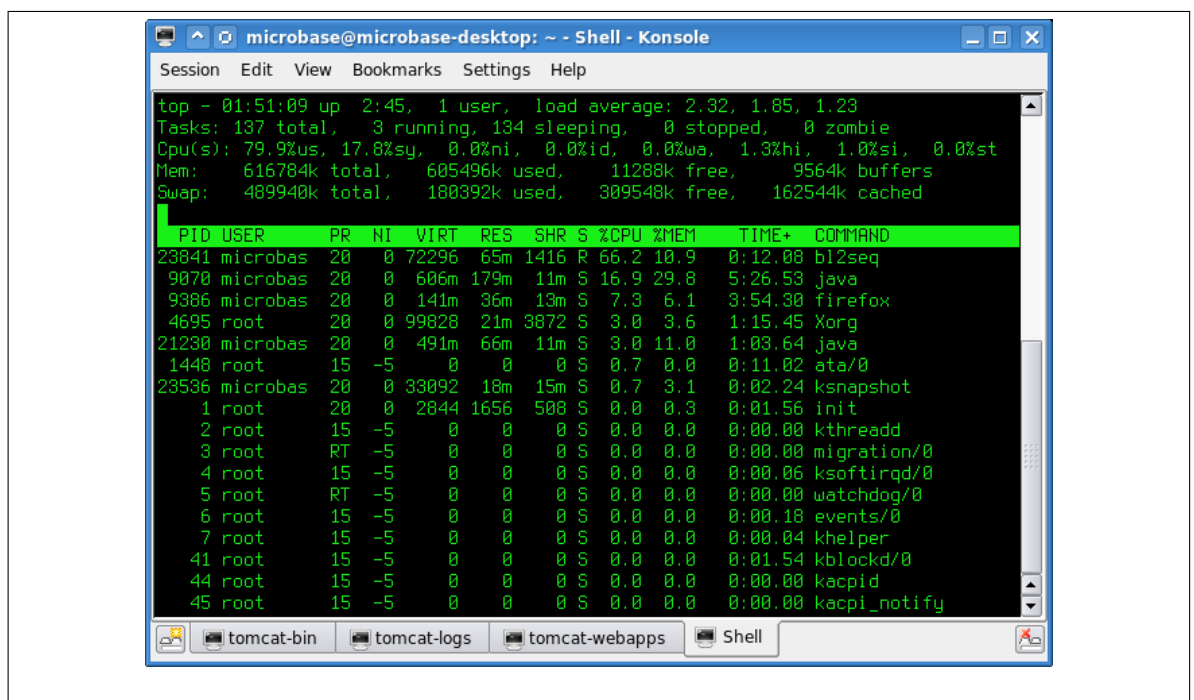


Figure A.10: A 'Bl2seq' processing running within a Microbase job.

Bibliography

- [1] Amazon EC2. <http://aws.amazon.com/ec2/>. [accessed 2009/10/05].
- [2] Google Apps. <http://www.google.com/apps/>. [accessed 2009/10/02].
- [3] JavaCC. <https://javacc.dev.java.net/>, 2009. [accessed October 2008].
- [4] William Allcock, John Bresnahan, Rajkumar Kettimuthu, and Michael Link. The Globus striped GridFTP framework and server. *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Nov 2005.
- [5] I Altintas, C Berkley, E Jaeger, M Jones, B Ludscher, and S Mock. Kepler: Towards a grid-enabled system for scientific workflows. *In the Workflow in Grid Systems Workshop in GGF10 - The Tenth Global Grid Forum, Berlin, Germany*, Jan 2004.
- [6] S F Altschul, W Gish, W Miller, E W Myers, and D J Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, Oct 1990.
- [7] Stephen F Altschul, Thomas L Madden, Alejandro A Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25(17):3389–3402, Aug 1997.
- [8] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the American Federation of Information Processing Societies Conference, AFIPS*, 30:483–485, Jan 1967.
- [9] D Anderson. BOINC: a system for public-resource computing and storage. *Grid Computing, 2004. Proceedings. Fifth IEEE/ACM International Workshop on*, pages 4 – 10, Oct 2004.
- [10] D Anderson, E Korpela, and R Walton. High-performance task distribution for volunteer computing. *e-Science and Grid Computing*, Jan 2005.
- [11] S Anderson. Shotgun DNA sequencing using cloned DNase I-generated fragments. *Nucleic Acids Research*, 9(13):3015–27, Jul 1981.
- [12] T Anderson, D Culler, and D Patterson. A case for NOW (networks of workstations). *Micro, IEEE*, 15(1):54 – 64, Feb 1995.
- [13] T Anderson, M Dahlin, J Neefe, and D Patterson. Serverless network file systems. *ACM Transactions on Computer Systems (TOCS)*, 14(1):41–79, Jan 1996.
- [14] S Androutsellis-Theotokis and D Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys (CSUR)*, Jan 2004.

- [15] Fred Annexstein, Kenneth Berman, and Mihajlo Jovanović. Latency effects on reachability in large-scale peer-to-peer networks. *SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures*, Jul 2001.
- [16] M Ashburner, C A Ball, J A Blake, D Botstein, H Butler, J M Cherry, A P Davis, K Dolinski, S S Dwight, J T Eppig, M A Harris, D P Hill, L Issel-Tarver, A Kasarskis, S Lewis, J C Matese, J E Richardson, M Ringwald, G M Rubin, and G Sherlock. Gene Ontology: tool for the unification of biology. *Nat Genet*, 25(1):25–29, May 2000.
- [17] InfiniBand Trade Association. Infiniband. <http://www.infinibandta.org/home>. [accessed 2009/05/09].
- [18] M Atkinson, D DeRoure, A Dunlop, and G Fox. Web service grids: an evolutionary approach. *Concurrency and Computation: Practice and Experience*, 17:377–389, Jan 2005.
- [19] Jean Bacon. COBEA: A CORBA-based event architecture. in *Proceedings of the 4th Conference on Object-Oriented Technologies and Systems, USENIX*, pages 117–131, May 1998.
- [20] Greg J Badros. A caching NFS client for linux. *4th Annual Linux Expo, Durham, NC*, Nov 1998.
- [21] L Baduel, F Baude, D Caromel, A Contes, and F Huet. *Programming, Composing, Deploying for the Grid*. 2006.
- [22] S Baset and H Schulzrinne. An analysis of the Skype peer-to-peer internet telephony protocol. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1 – 11, Apr 2006.
- [23] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L McGuinness, Peter F Patel-Schneider, and Lynn Andrea Stein. OWL Web ontology language reference. <http://www.w3.org/TR/owl-ref/>, 2004. [accessed 2009/05/07].
- [24] P Beckman. Building the teragrid. *Philosophical Transactions: Mathematical*, Jan 2005.
- [25] Brett Beeson, Steve Melnikoff, Srikumar Venugopal, and David Barnes. A portal for grid-enabled physics. *ACSW Frontiers '05: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*, 44, Jan 2005.
- [26] Jannick Dyrlov Bendtsen, Henrik Nielsen, Gunnar von Heijne, and Søren Brunak. Improved prediction of signal peptides: SignalP 3.0. *Journal of Molecular Biology*, 340(4):783–95, Jul 2004.
- [27] Dennis A Benson, Ilene Karsch-Mizrachi, David J Lipman, James Ostell, and Eric W Sayers. GenBank. *Nucleic Acids Research*, 37(Database issue):D26–31, Jan 2009.
- [28] Alex Berson. Client/server architecture. *McGraw-Hill*, 1992. ISBN: 0-07-005076-7.
- [29] A Bhushan. RFC 141: A file transfer protocol. <http://www.faqs.org/rfcs/rfc114.html>, 1971. [accessed 2009/04/20].
- [30] A Bilas and J Singh. The effects of communication parameters on end performance of shared virtual memory clusters. *Supercomputing, ACM/IEEE 1997 Conference*, Oct 1997.
- [31] A Billion, R Ghai, T Chakraborty, and T Hain. Augur—a computational pipeline for whole genome microbial surface protein prediction and classification. *Bioinformatics*, 22(22):2819–20, Nov 2006.

- [32] Tim T Binnewies, Yair Motro, Peter F Hallin, Ole Lund, David Dunn, Tom La, David J Hampson, Matthew Bellgard, Trudy M Wassenaar, and David W Ussery. Ten years of bacterial genome sequencing: comparative-genomics-based discoveries. *Funct Integr Genomics*, 6(3):165–185, Jun 2006.
- [33] K Birman and T Joseph. Exploiting virtual synchrony in distributed systems. *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, Nov 1987.
- [34] Ewan Birney, Michele Clamp, and Richard Durbin. Genewise and genomewise. *Genome Research*, 14(5):988–95, May 2004.
- [35] J Blythe, S Jain, E Deelman, Y Gil, K Vahi, A Mandal, and K Kennedy. Task scheduling strategies for workflow-based applications in grids. *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, 2:759 – 767 Vol. 2, Apr 2005.
- [36] W Bolosky, R Fitzgerald, and M Scott. Simple but effective techniques for NUMA memory management. *ACM SIGOPS Operating Systems Review*, 23(5), Nov 1989.
- [37] F Bordignon and G Tolosa. Gnutella: Distributed system for information storage and searching model description. *Journal of Internet Technology*, 2002.
- [38] M Brambilla, S Ceri, M Passamani, and A Riccio. Managing asynchronous web services interactions. *Web Services, 2004. Proceedings. IEEE International Conference on*, pages 80 – 87, Jun 2004.
- [39] A Brodtkorb. The graphics processor as a mathematical coprocessor in MATLAB. *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*, pages 822 – 827, Feb 2008.
- [40] I Buck, T Foley, D Horn, J Sugerman, and K Fatahalian. Brook for GPUs: stream computing on graphics hardware. *ACM Transactions on Graphics (TOG)*, 23(3):777–786, Jan 2004.
- [41] K Buetow. Cyberinfrastructure: Empowering a “third way” in biomedical research. *Science*, 308(5723):821–824, Jan 2005.
- [42] C Burge and S Karlin. Prediction of complete gene structures in human genomic dna. *Journal of Molecular Biology*, 268(1):78–94, Jan 1997.
- [43] W Cai, G Coulson, P Grace, G Blair, and L Mathy. The Gridkit distributed resource management framework. *LECTURE NOTES IN COMPUTER SCIENCE*, Jan 2005.
- [44] J Cao, S Jarvis, S Saini, and G Nudd. Gridflow: workflow management for grid computing. *Cluster Computing and the Grid*, Jan 2003.
- [45] J Cao, F Liu, and C Xu. P2PGrid: integrating P2P networks into the Grid environment. *Concurrency and Computation: Practice & Experience*, 19:1023–1046, Jan 2007.
- [46] P Carns, W Ligon III, R Ross, and R Thakur. PVFS: a parallel file system for linux clusters. *In Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Jan 2000.
- [47] J Carter, D Khandekar, and L Kamb. Distributed shared memory: where we are and where we should be headed. *Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on*, pages 119 – 122, Apr 1995.
- [48] Tim J Carver, Kim M Rutherford, Matthew Berriman, Marie-Adele Rajandream, Barclay G Barrell, and Julian Parkhill. Act: The artemis comparison tool. *Bioinformatics*, 21(16):3422–3, Aug 2005.

- [49] M Castro, P Druschel, A-M Kermarrec, and A Rowstron. Scribe: A large-scale and decentralised application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, pages 1–11, Sep 2002.
- [50] K Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Jan 1985.
- [51] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson Hsieh, Deborah Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. *Transactions on Computer Systems*, 26(2), Jun 2008.
- [52] Kamalsinh Chavda. Anatomy of a web service. *Journal of Computing Sciences in Colleges*, 19(3):124–134, Jan 2004.
- [53] Chungmin Chen, K Salem, and M Livny. The DBC: processing scientific data over the Internet. *Distributed Computing Systems, 1996., Proceedings of the 16th International Conference on*, pages 673 – 679, Apr 1996.
- [54] M Chetty and R Buyya. Weaving computational Grids: how analogous are they with electrical Grids? *Computing in Science & Engineering*, 4(4):61–71, Jan 2002.
- [55] T Clark, S Martin, and T Liefeld. Globally distributed object identification for biological knowledgebases. *Brief Bioinformatics*, Jan 2004.
- [56] Clotilde Claudel-Renard, Claude Chevalet, Thomas Faraut, and Daniel Kahn. Enzyme-specific profiles for genome annotation: Priam. *Nucleic Acids Research*, 31(22):6633–9, Nov 2003.
- [57] Bram Cohen. Incentives build robustness in BitTorrent. *Workshop on Economics of Peer-to-Peer Systems*, Jan 2003.
- [58] Platform Computing. Load sharing facility. <http://www.platform.com>, 2009. [accessed 2009/05/05].
- [59] World Wide Web Consortium. SOAP Version 1.2 Part 0: Primer (Second Edition). <http://www.w3.org/TR/soap12-part0/>. [accessed 2009/10/02].
- [60] NVIDIA Corporation. Compute unified device architecture (CUDA). http://www.nvidia.com/object/cuda_learn.html. [accessed Nov 2008].
- [61] F Corradini, L Mariani, and E Merelli. An agent-based approach to tool integration. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):231–244, Jan 2004.
- [62] F Costa, L Silva, I Kelley, and G Fedak. Optimizing the data distribution layer of BOINC with BitTorrent. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 – 8, Mar 2008.
- [63] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*, volume Fourth edition. 2005.
- [64] A Cox, S Dwarkadas, P Keleher, Honghui Lu, R Rajamony, and W Zwaenepoel. Software versus hardware shared-memory implementation: a case study. *Computer Architecture, 1994. Proceedings the 21st Annual International Symposium on*, pages 106 – 117, Mar 1994.
- [65] Tracy Craddock, Colin R Harwood, Jennifer Hallinan, and Anil Wipat. e-Science: relieving bottlenecks in large-scale genome analyses. *Nat Rev Microbiol*, 6(12):948–54, Dec 2008.

- [66] S Davidson, S Cohen-Boulakia, A Eyal, and B Ludascher. Provenance in scientific workflow systems. *IEEE Data Bulletin Engineering*, Jan 2007.
- [67] R de Knikker, Y Guo, J Li, A Kwan, K Yip, David W Cheung, and Kei-Hoi Cheung. A web services choreography scenario for interoperating bioinformatics applications. *BMC Bioinformatics*, 5(25), Jan 2004.
- [68] Adrian Perreau de Pinninck, David Dupplaw, Spyros Kotoulas, and Ronny Siebes. The open-knowledge kernel. *International Journal of Applied Mathematics and Computer Sciences*, 4(3):162–167, Jun 2007.
- [69] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan 2008.
- [70] K Decker, X Zheng, and C Schmidt. A multi-agent system for automated genomic annotation. *Proceedings of the fifth international conference on Autonomous agents*, pages 433–440, Jan 2001.
- [71] A L Delcher, D Harmon, S Kasif, O White, and S L Salzberg. Improved microbial gene identification with glimmer. *Nucleic Acids Research*, 27(23):4636–41, Dec 1999.
- [72] Dan Diephouse. Maven quick-start tutorial. <http://docs.codehaus.org/display/XFIRE/Quick+Start>. [accessed 2009/04/19].
- [73] Dan Diephouse. XFire. <http://docs.codehaus.org/display/XFIRE/Home>. [accessed 2009/04/19].
- [74] Distributed.net. Project RC5. <http://www.distributed.net/rc5/>. [accessed 2009/05/07].
- [75] B Dreier, M Zahn, and T Ungerer. The rthreads distributed shared memory system. *Proc. of the 3rd Int'l Conference on Massively Parallel Computing Systems*, Jan 1998.
- [76] P Druschel and A Rowstron. Past: a large-scale, persistent peer-to-peer storage utility. *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 75 – 80, Apr 2001.
- [77] Michel Dubois, Jin Wang, Luiz Barroso, Kangwoo Lee, and Yung-Syau Chen. Delayed consistency and its effects on the miss rate of parallel programs. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 197–206, Aug 1991.
- [78] Dan Duchamp. Optimistic lookup of whole nfs paths in a single operation. *In Proceedings of the 1994 USENIX Summer Conference*, pages 161–169, Aug 1994.
- [79] R Duncan. A survey of parallel computer architectures. *Computer*, Jan 1990.
- [80] S R Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–63, Jan 1998.
- [81] Robert C Edgar. Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Research*, 32(5):1792–7, Jan 2004.
- [82] Nature Methods Editorial. Metagenomics versus Moore’s law. *Nature Methods*, 6(9):623–623, Jan 2009.
- [83] Jeri Edwards and Deborah DeVoe. 3-tier client/server at work. *John Wiley & Sons*, 1997. ISBN: 0 471-18443-8.

- [84] H El-Rewini, H Ali, and T Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27 – 37, Dec 1995.
- [85] E Elmroth, F Hernandez, and J Tordsson. A light-weight grid workflow execution service enabling client and middleware independence. *Parallel Processing and Applied Mathematics, Lecture Notes in Computer Science*, 4967:754–761, 2008.
- [86] Hakan Erdogmus. Cloud computing: Does nirvana hide behind the nebula? *Software, IEEE*, 26(2):4 – 6, Mar 2009.
- [87] A Erlichson, N Nuckolls, G Chesson, and J Hennessy. Softflash: analyzing the performance of clustered distributed virtual shared memory. *ACM SIGOPS Operating Systems Review*, 30(5):210–220, Jan 1996.
- [88] Jayson Falkner. Tranche project. <https://trancheproject.org/>. [accessed 2009/05/07].
- [89] G Fedak, H He, and F Cappello. Distributing and managing data on desktop grids with Bit-Dew. *UPGRADE '08: Proceedings of the third international workshop on Use of P2P, grid and agents for the development of content networks*, pages 63–64, Jan 2008.
- [90] J Ferreira, J Sobral, and A Proenca. Jaskel: a java skeleton-based framework for structured cluster and grid computing. *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, 1:4, May 2006.
- [91] D Field, G Wilson, and Christopher van der Gast. How do we compare hundreds of bacterial genomes? *Current Opinion in Microbiology*, 9(5):499–504, Jan 2006.
- [92] R Fielding, J Gettys, J Mogul, H Frystyk, L Masinter, P Leach, and T Berners-Lee. RFC 2616: Hypertext transfer protocol – HTTP/1.1. <http://www.faqs.org/rfcs/rfc2616.html>, 1999. [accessed 2009/04/20].
- [93] W Fiers, R Contreras, F Duerinck, G Haegeman, D Iserentant, J Merregaert, W Min Jou, F Molemans, A Raeymaekers, A Van den Berghe, G Volckaert, and M Ysebaert. Complete nucleotide sequence of bacteriophage MS2 RNA: primary and secondary structure of the replicase gene. *Nature*, 260(5551):500–7, Apr 1976.
- [94] R Figueiredo, P Dinda, and J Fortes. A case for grid computing on virtual machines. *Distributed Computing Systems*, Jan 2003.
- [95] K Flanagan, R Stevens, M Pocock, P Lee, and A Wipat. Ontology for genome comparison and genomic rearrangements, Jan 2004.
- [96] R D Fleischmann, M D Adams, O White, R A Clayton, E F Kirkness, A R Kerlavage, C J Bult, J F Tomb, B A Dougherty, and J M Merrick. Whole-genome random sequencing and assembly of haemophilus influenzae Rd. *Science*, 269(5223):496–512, Jul 1995.
- [97] P Flicek, B. L Aken, K Beal, B Ballester, M Caccamo, Y Chen, L Clarke, G Coates, F Cunningham, T Cutts, T Down, S. C Dyer, T Eyre, S Fitzgerald, J Fernandez-Banet, S Graf, S Haider, M Hammond, R Holland, K. L Howe, K Howe, N Johnson, A Jenkinson, A Kahari, D Keefe, F Kokocinski, E Kulesha, D Lawson, I Longden, K Megy, P Meidl, B Overduin, A Parker, B Pritchard, A Prlic, S Rice, D Rios, M Schuster, I Sealy, G Slater, D Smedley, G Spudich, S Trevanion, A. J Vilella, J Vogel, S White, M Wood, E Birney, T Cox, V Curwen, R Durbin, X. M Fernandez-Suarez, J Herrero, T. J. P Hubbard, A Kasprzyk, G Proctor, J Smith, A Ureta-Vidal, and S Searle. Ensembl 2008. *Nucleic Acids Research*, 36(Database):D707–D714, Dec 2007.

- [98] L Florea, C Riemer, S Schwartz, Z Zhang, N Stojanovic, W Miller, and M McClelland. Web-based visualization tools for bacterial genome alignments. *Nucleic Acids Research*, 28(18):3486–96, Sep 2000.
- [99] M J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, c-21(9):948–960, 1972.
- [100] European Organization for Nuclear Research. LHC computing grid. <http://lcg.web.cern.ch/LCG/>. [accessed 2009/05/14].
- [101] MPI Forum. MPI: A message-passing interface standard (version 1.1). <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>, Apr 1995.
- [102] Open Grid Forum. The open grid services architecture, version 1.5. 2006.
- [103] The Gnutella Developer Forum. The Gnutella protocol specification v0.4. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf. [accessed 2009/04/20].
- [104] The Gnutella Developer Forum. The annotated Gnutella protocol specification v0.4. <http://rfc-gnutella.sourceforge.net/developer/stable/index.html>, 2001. accessed 2009/04/20.
- [105] I Foster. Globus toolkit version 4: Software for service-oriented systems. *Journal of Computer Science and Technology*, 21(4):513–520, Jan 2006.
- [106] I Foster and A Iamnitchi. On death, taxes, and the convergence of peer-to-peer and grid computing. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, pages 118–128, Jan 2003.
- [107] I Foster, C Kesselman, J Nick, and S Tuecke. The physiology of the grid. *Grid Computing: Making the Global Infrastructure a Reality*, Jan 2003.
- [108] I Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: enabling scalable virtual organizations. *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on AB - ER -*, pages 6–7, 2001.
- [109] I Foster, H Kishimoto, A Savva, D Berry, A Djaoui, A Grimshaw, B Horn, F Maciel, F Siebenlist, R Subramaniam, J Treadwell, and J Von Reich. The open grid services architecture, version 1.0 (gfd-i.030). page 62, Mar 2005.
- [110] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11:115–128, Nov 1997.
- [111] Ian Foster and Carl Kesselman. The Grid: Blueprint for a new computing infrastructure. *Morgan Kaufmann Publishers Inc.*, 1999.
- [112] The Apache Software Foundation. Apache HTTP Server Project. <http://httpd.apache.org/>. [accessed 2009/09/30].
- [113] The Apache Software Foundation. Apache tomcat. <http://tomcat.apache.org/>. [accessed 2009/04/19].
- [114] The Apache Software Foundation. Maven. <http://maven.apache.org/>. accessed 2009/04/19.
- [115] D Frishman and H Mewes. Pedantic genome analysis. *Trends in Genetics*, 10:415–416, Jan 1997.

- [116] K Fritsche, J Power, and J Waldron. A Java distributed computation library. *Proceedings of the 2nd International Conference on Parallel and Distributed Computing, Applications and Technologies PDCA, Taipei, Taiwan*, pages 236–243, 2001.
- [117] M Fukuda, Y Tanaka, N Suzuki, L Bic, and S Kobayashi. A mobile-agent-based PC grid. *Autonomic Computing Workshop*, Jan 2003.
- [118] A Ganguly, A Agrawal, P Boykin, and R Figueiredo. WOW: Self-organizing wide area overlay networks of virtual workstations. *Journal of Grid Computing*, Jan 2007.
- [119] Mark K Gardner, Wu chun Feng, Jeremy Archuleta, Heshan Lin, and Xiaosong Ma. Parallel genomic sequence-searching on an ad-hoc grid: experiences, lessons learned, and implications. *Conference on High Performance Networking and Computing*, Nov 2006.
- [120] M Garland. Sparse matrix computations on manycore GPU’s. *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 2 – 6, May 2008.
- [121] N Geddes. The national grid service of the uk. *e-Science and Grid Computing, 2006. e-Science ’06. Second IEEE International Conference on*, pages 94 – 94, Dec 2006.
- [122] C Goble and D De Roure. myExperiment: social networking for workflow-using e-scientists. *In: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 1–2, Jan 2007.
- [123] Y Goland, E Whitehead, A Faizi, S Carter, and D Jensen. RFC 2518: HTTP extensions for distributed authoring – WEBDAV. <http://www.faqs.org/rfcs/rfc2518.html>, 1999. [accessed 2009/04/20].
- [124] B Goldsmith. Comptorrent: Applying bittorrent techniques to distributed computing. *eprints.utas.edu.au*, Jan 2006.
- [125] Google. Google web toolkit. <http://code.google.com/webtoolkit/>. [accessed 2009/09/30].
- [126] P Grace, G Coulson, G Blair, L Mathy, and W Yeung. Gridkit: Pluggable overlay networks for grid computing. *Lecture Notes in Computer Science*, 3291, Jan 2004.
- [127] J Grant, R Dunbrack, F Manion, and M Ochs. Beoblast: distributed blast and psi-blast on a beowulf cluster. *Bioinformatics*, Jan 2002.
- [128] PBS GridWorks. OpenPBS. 2001. accessed 2009/05/05.
- [129] A Grimshaw and W Wulf. Legion: The next logical step toward the world-wide virtual computer. *Communications of the ACM*, 40, Jan 1996.
- [130] PostgreSQL Global Development Group. PostgreSQL. <http://www.postgresql.org/>, 2009. [accessed 2009/04/19].
- [131] W3C Working Group. Web services glossary. <http://www.w3.org/TR/ws-gloss/>. [accessed 2009/04/20].
- [132] T Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43(5-6):907–928, Jan 1995.

- [133] Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. Grid-flow: A grid-enabled scientific workflow system with a petri net-based interface. *Concurrency and Computation: Practice and Experience*, 18:1115–1140, Jan 2006.
- [134] L Guo, S Chen, Z Xiao, E Tan, X Ding, and X Zhang. A performance study of BitTorrent-like peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 25(1):155 – 169, Jan 2007.
- [135] R Gupta and A Somani. Compup2p: An architecture for sharing of computing resources in peer-to-peer networks with selfish nodes. *Second workshop on the Economics of Peer-to-Peer systems*, Jan 2004.
- [136] John Gustafson. Reevaluating Amdahl’s law. *Communications of the ACM*, 31(5):532–533, May 1988.
- [137] D Guthridge. Scalable, high performance infiniband-attached SAN volume controller. *Cluster Computing, 2008 IEEE International Conference on*, pages 453 – 458, Jan 2008.
- [138] Neil Hall. Advanced sequencing technologies and their wider impact in microbiology. *J Exp Biol*, 210(Pt 9):1518–25, May 2007.
- [139] Mark Halling-Brown, David Moss, and Adrian Shepherd. Towards a lightweight generic computational grid framework for biological research. *BMC Bioinformatics*, 9:407, Oct 2008.
- [140] B Harris, A Jacob, J Lancaster, and J Buhler. A banded Smith-Waterman FPGA accelerator for mercury BLASTP. *International Conference on Field Programmable Logic and Applications*, pages 765–769, Jan 2007.
- [141] Bjarne E Helvik and Otto Wittner. Network resilience by emergent behaviour from simple autonomous agents. *In Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, pages 449–478, 2005.
- [142] A Hey and A Trefethen. The data deluge: an e-Science perspective. *In: Grid Computing - Making the Global Infrastructure a Reality*, ISBN: 0470853190:809–824, Jan 2003.
- [143] T Hey. e-Science and its implications. *Philosophical Transactions: Mathematical*, Jan 2003.
- [144] Mark D Hill and Michael R Marty. Amdahl’s law in the multicore era. *Computer*, 41(7):33 – 38, Jul 2008.
- [145] C Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, Jan 1974.
- [146] Kathryn E Holt, Nicholas R Thomson, John Wain, Minh Duy Phan, Satheesh Nair, Rumina Hasan, Zulfiqar A Bhutta, Michael A Quail, Halina Norbertczak, Danielle Walker, Gordon Dougan, and Julian Parkhill. Multidrug-resistant salmonella enterica serovar paratyphi a harbors inchi1 plasmids similar to those found in serovar typhi. *J Bacteriol*, 189(11):4257–64, Jun 2007.
- [147] S Hoon, K Ratnapu, J Chia, and B Kumarasamy. Biopipe: A flexible framework for protocol-based bioinformatics analysis. *Genome Research*, Jan 2003.
- [148] E Horowitz and A Zorat. Divide-and-conquer for parallel processing. *Transactions on Computers*, Jan 1983.

- [149] Takashige Hoshiai. Approximate analysis of access contention for multi-processor systems with common bus arbiters. *Systems and Computers in Japan*, 27(9):12–22, Dec 1996.
- [150] W Hsiao. Islandpath: aiding detection of genomic islands in prokaryotes. *Bioinformatics*, 19(3):418–420, Feb 2003.
- [151] D Hughes, P Greenwood, Coulson G, and G Blair. Gridstix: supporting flood prediction using embedded hardware and next generation grid middleware. *World of Wireless, Mobile and Multimedia Networks, 2006. WoWMoM 2006. International Symposium on a*, page 6, May 2006.
- [152] D Hull, K Wolstencroft, R Stevens, and C Goble. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, Jan 2006.
- [153] S Ibrahim, Hai Jin, Li Qi, and Chunqiang Zeng;. Grid maintenance: Challenges and existing models. *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1 – 6, Apr 2008.
- [154] W Sullivan III, D Werthimer, S Bowyer, J Cobb, D Gedye, and D Anderson. A new major SETI project based on project SERENDIP data and 100,000 personal computers. *Proceedings of the Fifth International Conference on Bioastronomy*, 161, Jan 1997.
- [155] A Iliasova. *Personal communication*, Newcastle University, UK., 2009.
- [156] L Ismail and D Hagimont. A performance evaluation of the mobile agent paradigm. *ACM SIGPLAN Notices*, Jan 1999.
- [157] ISO. Information technology - open distributed processing - reference model: Overview. *ISO/IEC 10746-1*, 1998.
- [158] M Itoh and H Watanabe. CGAS: Comparative genomic analysis server. *Bioinformatics*, Feb 2009.
- [159] C Johns and D Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5):503–519, Jan 2007.
- [160] G Johnson, D Kerbyson, and M Lang. Optimization of infiniband for scientific applications. *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 – 8, Mar 2008.
- [161] J Joseph, M Ernest, and C Fellenstein. Evolution of grid computing architecture and grid adoption models. *IBM Systems Journal*, 43(4):624–645, Oct 2004.
- [162] W Min Jou, G Haegeman, M Ysebaert, and W Fiers. Nucleotide sequence of the gene coding for the bacteriophage MS2 coat protein. *Nature*, 237(5350):82–88, May 1972.
- [163] Agnieszka S Juncker, Hanni Willenbrock, Gunnar Von Heijne, Søren Brunak, Henrik Nielsen, and Anders Krogh. Prediction of lipoprotein signal peptides in gram-negative bacteria. *Protein Sci*, 12(8):1652–62, Aug 2003.
- [164] M Junginger and Y Lee. A self-organizing publish/subscribe middleware for dynamic peer-to-peer networks. *IEEE network*, (January/February):38–43, Jan 2004.
- [165] Matjaz Juric, Bostjan Kezmah, Marjan Hericko, Ivan Rozman, and Ivan Vezocnik. Java RMI, RMI tunneling and Web services comparison and performance analysis. *SIGPLAN Notices*, 39(5), May 2004.

- [166] L Juszczuk, J Lazowski, and S Dustdar. Web service discovery, replication, and synchronization in ad-hoc networks. *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*, page 8, Mar 2006.
- [167] M Karo, C Dwan, J Freeman, J Weissman, M Livny, and E Retzel. Applying grid technologies to bioinformatics. *High Performance Distributed Computing, 2001. Proceedings. 10th IEEE International Symposium on*, pages 441 – 442, Jul 2001.
- [168] K Keahey, M Tsugawa, A Matsunaga, and J Fortes. Sky computing. *Internet Computing, IEEE*, 13(5):43 – 51, Sep 2009.
- [169] Pete Keleher, Alan Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, 20(2):13–21, May 1992.
- [170] Z Konfrst. Parallel genetic algorithms: advances, computing trends, applications and perspectives. *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 162–169, Mar 2004.
- [171] A Krishna, V Tan, R Lawley, S Miles, and L Moreau. The myGrid notification service. *In Proceedings of The UK OST e-Science second All Hands Meeting 2003 (AHM'03)*, pages 475–482, Jan 2003.
- [172] A Krishnan. GridBLAST: a Globus-based high-throughput implementation of BLAST in a Grid computing framework. *Concurr Comp-Pract E*, 17(13):1607–1623, Jan 2005.
- [173] A Krogh, B Larsson, G von Heijne, and E L Sonnhammer. Predicting transmembrane protein topology with a hidden markov model: application to complete genomes. *Journal of Molecular Biology*, 305(3):567–80, Jan 2001.
- [174] John Kubiawicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, 2000.
- [175] Tamara Kulikova, Ruth Akhtar, Philippe Aldebert, Nicola Althorpe, Mikael Andersson, Alastair Baldwin, Kirsty Bates, Sumit Bhattacharyya, Lawrence Bower, Paul Browne, Matias Castro, Guy Cochrane, Karyn Duggan, Ruth Eberhardt, Nadeem Faruque, Gemma Hoad, Carola Kanz, Charles Lee, Rasko Leinonen, Quan Lin, Vincent Lombard, Rodrigo Lopez, Dariusz Lorenc, Hamish McWilliam, Gaurab Mukherjee, Francesco Nardone, Maria Pilar Garcia Pastor, Sheila Plaister, Siamak Sobhany, Peter Stoehr, Robert Vaughan, Dan Wu, Weimin Zhu, and Rolf Apweiler. Embl nucleotide sequence database in 2006. *Nucleic Acids Research*, 35(Database issue):D16–20, Jan 2007.
- [176] Stefan Kurtz, Adam Phillippy, Arthur L Delcher, Michael Smoot, Martin Shumway, Corina Antonescu, and Steven L Salzberg. Versatile and open software for comparing large genomes. *Genome Biol*, 5(2):R12, Jan 2004.
- [177] Horacio Lagar-Cavilla, Joseph Whitney, Adin Scannell, Philip Patchin, Stephen Rumble, Eyal Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, Apr 2009.

- [178] O Lampe, I Viola, N Reuter, and H Hauser. Two-level approach to efficient visualization of protein dynamics. *Visualization and Computer Graphics, IEEE Transactions on*, 13(6):1616 – 1623, Nov 2007.
- [179] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul 1978.
- [180] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690 – 691, Sep 1979.
- [181] Paul J Leach and Dilip C Naik. A common internet file system (CIFS/1.0) protocol. <http://www.microsoft.com/about/legal/protocols/BSTD/CIFS/draft-leach-cifs-v1-spec-02.txt>, 1997. [accessed 2009/04/20].
- [182] N Leavitt. Is cloud computing really ready for prime time? *Computer*, 42(1):15 – 20, Jan 2009.
- [183] Jonathan Ledlie, Jeff Shneidman, Margo Seltzer, and John Huth. Scooped, again. *Second International Workshop, IPTPS 2003 Berkeley*, Feb 2003.
- [184] Hurng-Chun Lee, Jean Salzemann, Nicolas Jacq, Hsin-Yen Chen, Li-Yung Ho, Ivan Merelli, Luciano Milanese, Vincent Breton, Simon C Lin, and Ying-Ta Wu. Grid-enabled high-throughput in silico screening against influenza a neuraminidase. *Ieee T Nanobiosci*, 5(4):288–295, Jan 2006.
- [185] Sung Lee, Taowei David Wang, Nada Hashmi, and Michael P Cummings. Bio-steer: A semantic web workflow tool for grid computing in the life sciences. *Future Gener Comp Sy*, 23(3):497–509, Jan 2007.
- [186] Isaac T S Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the smith-waterman algorithm using a field programmable gate array (fpga). *BMC Bioinformatics*, 8:185, Jan 2007.
- [187] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Jun 1989.
- [188] L Li, CJ Stoeckert, and DS Roos. OrthoMCL: Identification of ortholog groups for eukaryotic genomes. *Genome Research*, 13(9):2178–2189, Jan 2003.
- [189] Wenlong Li, Xiaofeng Tong, and Yimin Zhang;. Optimization and parallelization on a multimedia application. *Multimedia and Expo, 2007 IEEE International Conference on*, pages 1854 – 1857, Jun 2007.
- [190] M Linderman, N Ahmed, J Metzler, and J Bryant. A hybrid publish subscribe protocol. *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion table of contents Leuven, Belgium*, pages 24–29, Jan 2008.
- [191] Konstantinos Liolios, Konstantinos Mavromatis, Nektarios Tavernarakis, and Nikos C Kyrpides. The genomes on line database (gold) in 2007: status of genomic and metagenomic projects and their associated metadata. *Nucleic Acids Research*, 36(Database issue):D475–9, Jan 2008.
- [192] M Litzkow, M Livny, and M Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems*, Jan 1988.

- [193] M Litzkow, T Tannenbaum, J Basney, and M Livny. Checkpoint and migration of unix processes in the condor distributed processing system. *Technical Report Computer Sciences Technical Report #1346, University of Wisconsin-Madison*, Jan 1997.
- [194] Michael Litzkow, Miron Livny, and Matt Mutka. Condor - a hunter of idle workstations. *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, Apr 1988.
- [195] Chun-Chi Liu, Chin-Chung Lin, Ker-Chau Li, Wen-Shyen E Chen, Jiun-Ching Chen, Ming-Te Yang, Pan-Chyr Yang, Pei-Chun Chang, and Jeremy J W Chen. Genome-wide identification of specific oligonucleotides using artificial neural network and computational genomic analysis. *BMC Bioinformatics*, 8:164, Jan 2007.
- [196] Weiguo Liu, B Schmidt, G Voss, A Schroder, and W Muller-Wittig. Bio-sequence database scanning on a gpu. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8–15, Mar 2006.
- [197] Yang Liu, Jianrong Li, Lee Sam, Chern-Sing Goh, Mark Gerstein, and Yves A Lussier. An integrative genomic approach to uncover molecular mechanisms of prokaryotic traits. *PLoS Comp Biol*, 2(11):e159, Nov 2006.
- [198] Limewire LLC. Limewire client. <http://www.limewire.com/>. [accessed 2009/04/19].
- [199] Boon Thau Loo, Ryan Huebsch, Joseph M Hellerstein, Timothy Roscoe, and Ion Stoica. Analyzing p2p overlays with recursive queries. *Technical Report: IRB-TR-03-045, University of California at Berkeley, Intel Research Berkeley*, pages 1–6, Nov 2003.
- [200] H Lovell, J Mansfield, S Godfrey, R Jackson, J Hancock, and D Arnold. Bacterial evolution by genomic island transfer occurs via dna transformation in planta. *Curr Biol*, Sep 2009.
- [201] R Lucky. Cloud computing. *Spectrum, IEEE*, 46(5):27 – 27, May 2009.
- [202] B Ludäscher, I Altintas, C Berkley, and D Higgins. Scientific workflow management and the kepler system. *Concurrency and Computation: Practice & Experience*, Jan 2005.
- [203] Yves A Lussier and Yang Liu. Computational approaches to phenotyping: high-throughput phenomics. *Proceedings of the American Thoracic Society*, 4(1):18–25, Jan 2007.
- [204] Yuya Machida, Shin’ichiro Takizawa, Hidemoto Nakada, and Satoshi Matsuoka. Intelligent data staging with overlapped execution of grid applications, Jan 2008.
- [205] Natalia Maltsev, Elizabeth Glass, Dinanath Sulakhe, Alexis Rodriguez, Mustafa H Syed, Tanuja Bompada, Yi Zhang, and Mark D’Souza. Puma2–grid-based high-throughput analysis of genomes and metabolic pathways. *Nucleic Acids Res*, 34(Database issue):D369–72, Jan 2006.
- [206] RG Mann. Astrogrid: the uk’s virtual observatory initiative. *Astronomical Data Analysis Software and Systems XI, ASP Conference Series*, 281, Jan 2002.
- [207] Marcel Margulies, Michael Egholm, William E Altman, Said Attiya, Joel S Bader, Lisa A Bemben, Jan Berka, Michael S Braverman, Yi-Ju Chen, Zhoutao Chen, Scott B Dewell, Lei Du, Joseph M Fierro, Xavier V Gomes, Brian C Godwin, Wen He, Scott Helgesen, Chun Heen Ho, Chun He Ho, Gerard P Irzyk, Szilveszter C Jando, Maria L I Alenquer, Thomas P Jarvie, Kshama B Jirage, Jong-Bum Kim, James R Knight, Janna R Lanza, John H Leamon, Steven M

- Lefkowitz, Ming Lei, Jing Li, Kenton L Lohman, Hong Lu, Vinod B Makhijani, Keith E McDade, Michael P McKenna, Eugene W Myers, Elizabeth Nickerson, John R Nobile, Ramona Plant, Bernard P Puc, Michael T Ronan, George T Roth, Gary J Sarkis, Jan Fredrik Simons, John W Simpson, Maithreyan Srinivasan, Karrie R Tartaro, Alexander Tomasz, Kari A Vogt, Greg A Volkmer, Shally H Wang, Yong Wang, Michael P Weiner, Pengguang Yu, Richard F Begley, and Jonathan M Rothberg. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–80, Sep 2005.
- [208] Neelan J Marianayagam, Nicolas L Fawzi, and Teresa Head-Gordon. Protein folding by distributed computing and the denatured state ensemble. *Proc Natl Acad Sci USA*, 102(46):16684–9, Nov 2005.
- [209] A Matsunaga, M Tsugawa, and J Fortes. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 222 – 229, Dec 2008.
- [210] A M Maxam and W Gilbert. A new method for sequencing DNA. *Proc Natl Acad Sci USA*, 74(2):560–4, Feb 1977.
- [211] CJ McNeil, BJ Gallacher, CR Harwood, J Hedley, P Manning, A Wipat, J R Henderson, and N Keegan. AptaMEMS-ID. <http://gow.epsrc.ac.uk/ViewGrant.aspx?GrantRef=EP/G061394/1>. [accessed 2009/09/30].
- [212] R Melamede. Automatable process for sequencing nucleotide. *United States Patent 4863849*, Jan 1989.
- [213] Raquel Menezes, Carlos Baquero, and Francisco Moura. A portable lightweight approach to NFS replication. *In Proceedings of ROSE'94 Conference*, Oct 1994.
- [214] E Merelli, G Armano, N Cannata, and F Corradini. Agents in bioinformatics, computational and systems biology. *Brief Bioinformatics*, 8(1):45–59, Jan 2007.
- [215] E Merelli, R Culmone, and L Mariani. Bioagent: A mobile agent system for bioscientists. *NETTAB—Agents in Bioinformatics*, Jan 2002.
- [216] Sun Microsystems. Java. <http://java.sun.com/>. [accessed 2009/04/19].
- [217] Sun Microsystems. Java remote method invocation - distributed computing for java. <http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp>. [accessed 2009/04/27].
- [218] Sun Microsystems. Sun grid engine. <http://gridengine.sunsource.net/>. [accessed 2009/05/07].
- [219] Sun Microsystems. RFC1094 - NFS: network file system protocol specification. <http://www.faqs.org/rfcs/rfc1094.html>, 1989. <http://www.faqs.org/rfcs/rfc1094.html>.
- [220] Alex Mira, Howard Ochman, and Nancy A Moran. Deletional bias and the evolution of bacterial genomes. *Trends in Genetics*, 17(10):589–596, Sep 2001.
- [221] R Montero, E Huedo, and I Llorente. Dynamic deployment of custom execution environments in grids. *Advanced Engineering Computing and Applications in Sciences, 2008. ADVCOMP '08. The Second International Conference on*, pages 33 – 38, Jan 2008.

- [222] H Monti, A Butt, and S Vazhkudai. Timely offloading of result-data in HPC centers. *Proceedings of the 22nd annual international conference on Supercomputing*, pages 124–133, Jan 2008.
- [223] Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *In Proceedings of SIGCOMM 2001*, Jun 2001.
- [224] T Mowry, C Chan, and A Lo. Comparative evaluation of latency tolerance techniques for software distributed shared memory. *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, pages 300 – 311, Jan 1998.
- [225] Nicola Mulder and Rolf Apweiler. InterPro and InterProScan: tools for protein sequence classification and comparison. *Methods Mol Biol*, 396:59–70, Jan 2007.
- [226] James M Musser and Samuel A Shelburne. A decade of molecular pathogenomic analysis of group a streptococcus. *J Clin Invest*, 119(9):2455–63, Sep 2009.
- [227] Matt W Mutka and Miron Livny. Profiling workstation’s available capacity for remote execution. *Proceedings of the 12th IFIPWG International Symposium on Computer Performance Modelling, Measurement and Evaluation*, pages 529–544, 1987.
- [228] Anthony Nadalin, Chris Kaler, Phillip Hallam-Baker, and Ronald Monzillo. Web services security v1.0 (WS-Security 2004) [OASIS 200401]. <http://www.oasis-open.org/specs/index.php{#}wssv1.0>. [accessed 2009/09/30].
- [229] S Nakjang. Extracellular protein identification pipeline. http://homepages.cs.ncl.ac.uk/sirintra.nakjang/phd/figures/ExtProteinExtractor_pipeline2.png. [accessed 2009/10/02].
- [230] S Nakjang. *Personal communication*, Newcastle University, UK., 2009.
- [231] S B Needleman and C D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–53, Mar 1970.
- [232] P Neerincx and J Leunissen. Evolution of web services in bioinformatics. *Brief Bioinformatics*, 6(2):178–188, Jan 2005.
- [233] Lei Ni, Aaron Harwood, and Peter Stuckey. Realizing the e-science desktop peer using a peer-to-peer distributed virtual machine middleware. *MCG '06: Proceedings of the 4th international workshop on Middleware for grid computing*, Nov 2006.
- [234] M Nieto-Santisteban, A Szalay, A Thakar, WJ O’Mullane, Jim Gray, and James Annis. When database systems meet the grid. *MSR-TR-2004-81*, Microsoft Research(Technical Report), Jan 2004.
- [235] T Oinn, M Greenwood, M Addis, and M Alpdemir. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, Jan 2006.
- [236] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–54, Nov 2004.

- [237] Athanasios I Papadopoulos and Patrick Linke. A decision support grid for integrated molecular solvent design and chemical process selection. *Comput Chem Eng*, 33(1):72–87, Jan 2009.
- [238] J Parkhurst, J Darringer, and B Grundmann. From single core to multi-core: Preparing for a new exponential. *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 67 – 72, Oct 2006.
- [239] Terence Parr. ANTLR (ANother Tool for Language Recognition). <http://www.antlr.org/>. [accessed 2009/10/05].
- [240] L Pearlman, C Kesselman, S Gullapalli, B Spencer, J Futrelle, K Ricker, I Foster, P Hubbard, and C Severance. Distributed hybrid earthquake engineering experiments: experiences with a ground-shaking grid application. *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 14 – 23, May 2004.
- [241] M Carmen Garcia Pelayo, Swapna Uplekar, Andrew Keniry, Pablo Mendoza Lopez, Thierry Garnier, Javier Nunez Garcia, Laura Boschioli, Xiangmei Zhou, Julian Parkhill, Noel Smith, R Glyn Hewinson, Stewart T Cole, and Stephen V Gordon. A comprehensive survey of single nucleotide polymorphisms (snps) across mycobacterium bovis strains and m. bovis bcg vaccine strains refines the genealogy and defines a minimal set of snps that separate virulent m. bovis strains and m. bovis bcg strains. *Infect Immun*, 77(5):2230–8, May 2009.
- [242] R H Perrott. Parallel programming. *Addison-Wesley*, 1987.
- [243] K Petersen and K Li. Cache coherence for shared memory multiprocessors based on virtual memory support. *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, pages 49 – 55, Mar 1993.
- [244] S Pillai, V Silventoinen, K Kallio, and M Senger. Soap-based services provided by the european bioinformatics institute. *Nucleic Acids Research*, Jan 2005.
- [245] J Postel and J Reynolds. RFC 959: File transfer protocol (FTP). <http://www.faqs.org/rfcs/rfc959.html>, Jan 1985. [accessed 2009/04/20].
- [246] S Potter, L Clarke, V Curwen, S Keenan, and E Mongin. The Ensembl analysis pipeline. *Genome Research*, Jan 2004.
- [247] Apache Maven Project. Maven getting started guide. <http://maven.apache.org/guides/getting-started/index.html>. [accessed 2009/10/08].
- [248] Apache Maven Project. Maven philosophy. <http://maven.apache.org/background/philosophy-of-maven.html>. [accessed 2009/09/30].
- [249] Azureus Project. Azureus. <http://azureus.sourceforge.net/>. [accessed 2009/04/19].
- [250] Kim D Pruitt, Tatiana Tatusova, William Klimke, and Donna R Maglott. Ncbi reference sequences: current status, policy and new initiatives. *Nucleic Acids Research*, 37(Database issue):D32–6, Jan 2009.
- [251] J Pullen, R Brunton, D Brutzman, D Drake, and M Hieb. Using web services to integrate heterogeneous simulations in a grid environment. *Future Generation Computer Systems*, Jan 2005.
- [252] Yutao Qi and Feng Lin. Parallelisation of the blast algorithm. *Cell Mol Biol Lett*, 10(2):281–5, Jan 2005.

- [253] A Rajasekar, M Wan, R Moore, G Kremenek, and T Guptil. Data grids, collections, and grid bricks. *Mass Storage Systems and Technologies*, Jan 2003.
- [254] R Rajkumar, M Gagliardi, and Lui Sha;. The real-time publisher/subscriber inter-process communication model for distributed real-time systems: design and implementation. *Real-Time Technology and Applications Symposium, 1995. Proceedings*, pages 66 – 75, Apr 1995.
- [255] K Ranganathan and I Foster. Decoupling computation and data scheduling in distributed data-intensive applications. *High Performance Distributed Computing*, Jan 2002.
- [256] Virginie Lopez Rascol, Anthony Levasseur, Olivier Chabrol, Simona Grusea, Philippe Gouret, Etienne G J Danchin, and Pierre Pontarotti. Cassiope: an expert system for conserved regions searches. *BMC Bioinformatics*, 10:284, Jan 2009.
- [257] R Rettberg and R Thomas. Contention is no obstacle to shared-memory multiprocessing. *Communications of the ACM*, Jan 1986.
- [258] M Riley, T Schmidt, I Artamonova, and C Wagner. Pedant genome database: 10 years online. *Nucleic Acids Research*, Jan 2006.
- [259] Jonathan M Rothberg and John H Leamon. The development and impact of 454 sequencing. *Nat Biotechnol*, 26(10):1117–24, Oct 2008.
- [260] Antony Rowstron. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *In Proceedings of IFIP/ACM Middleware*, Sep 2001.
- [261] M Roy and A Edward. Inside dcom: Microsoft’s distributed object architecture extends the capabilities of com to work across the network. *DBMS*, 10(4):26–34, Jan 1997.
- [262] S Roy and V Chaudhary. Strings: a high-performance distributed shared memory for symmetrical multiprocessor clusters. *High Performance Distributed Computing*, Jan 1998.
- [263] Mark Russell and Tim Hopkins. CFTP – a caching FTP server. *Third International WWW Caching Workshop*, Apr 1998.
- [264] Declan Ryan, Maryam Rahimi, John Lund, Ranjana Mehta, and Babak A Parviz. Toward nanoscale genome sequencing. *Trends Biotechnol*, 25(9):385–9, Sep 2007.
- [265] F Sacerdoti, S Chandra, and K Bhatia. Grid systems deployment & management using rocks. *Cluster Computing, 2004 IEEE International Conference on*, pages 337 – 345, Sep 2004.
- [266] Carlos Sanchez, John Casey, Vincent Massol, and Jason van Zyl. Better builds with maven (online book). <http://www.maestrodev.com/better-build-maven>. [accessed 2009/04/19].
- [267] F Sanger and A Coulson. Rapid method for determining sequences in dna by primed synthesis with dna-polymerase. *Journal of Molecular Biology*, 94(3):441–&, Jan 1975.
- [268] F Sanger, A R Coulson, T Friedmann, G M Air, B G Barrell, N L Brown, J C Fiddes, C A Hutchison, P M Slocombe, and M Smith. The nucleotide sequence of bacteriophage phix174. *Journal of Molecular Biology*, 125(2):225–46, Oct 1978.
- [269] F Sanger, S Nicklen, and A R Coulson. DNA sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci USA*, 74(12):5463–7, Dec 1977.
- [270] F Sanger and E.O.P Thompson. The amino-acid sequence in the glycol chain of insulin. *Biochem J*, 52(1):iii, Sep 1952.

- [271] Magdalena Sawinska, Dawid Kurzyniec, Jarosaw Sawinski, and Vaidy Sunderam. Automated deployment support for parallel distributed computing. *Parallel, Distributed and Network-Based Processing, 2007. PDP '07. 15th EUROMICRO International Conference on*, pages 139 – 146, Jan 2007.
- [272] Jennifer M Schopf and Bill Nitzberg. Grids: The top ten questions. *Scientific Programming*, 10(2):103–11, Jul 2002.
- [273] K Schreiner. Distributed projects tackle protein mystery. *Computing in Science & Engineering*, 3(1):13 – 16, Jan 2001.
- [274] L Schroeder and A Bazzan. A multi-agent system to facilitate knowledge discovery: an application to bioinformatics. *Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, pages 11–14, Jan 2002.
- [275] M Senger, P Rice, and T Oinn. Soaplab-a unified sesame door to analysis tools. *Proceedings of the UK e-Science All Hands Meeting*, Jan 2003.
- [276] Robert F Service. The race for the \$1000 genome. *Science*, 311:1544–1546, 2006.
- [277] A Shah, D Barthel, P Lukasiak, and J Blazewicz. Web & grid technologies in bioinformatics, computational and systems biology: A review. *Current Bioinformatics*, 3(1):10–31, 2008.
- [278] A Shah, V Markowitz, and C Oehmen. High-throughput computation of pairwise sequence similarities for multiple genome comparisons using scalablast. *Life Science Systems and Applications Workshop, 2007. LISA 2007. IEEE/NIH*, pages 89 – 91, Oct 2007.
- [279] Michael Shirts and Vijay S Pande. Screen savers of the world unite! *Science*, 290(5498):1903–1904, Oct 2000.
- [280] T F Smith and M S Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–7, Mar 1981.
- [281] S Soltis, T Ruwart, and M O’Keefe. The global file system. *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, Jan 1996.
- [282] D.J Sorin, M.M.K Martin, M.D Hill, and D.A Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 123–134, 2002.
- [283] Lincoln D Stein. Integrating biological databases. *Nat Rev Genet*, 4(5):337–345, May 2003.
- [284] Robert D Stevens, Alan J Robinson, and Carole A Goble. mygrid: personalised bioinformatics on the information grid. *Bioinformatics*, 19 Suppl 1:i302–4, Jan 2003.
- [285] H Stockinger. Distributed database management systems and the data grid. *Mass Storage Systems and Technologies, 2001. MSS '01. Eighteenth IEEE Symposium on*, pages 1 – 1, Apr 2001.
- [286] H Stockinger, T Attwood, S Chohan, and R Cote. Experience using web services for biological sequence analysis. *Brief Bioinformatics*, Jan 2008.
- [287] H Sugawara, O Ogasawara, K Okubo, T Gojobori, and Y Tateno. Ddbj with new system and face. *Nucleic Acids Research*, 36(Database issue):D22–4, Jan 2008.

- [288] Dinanath Sulakhe, Alex Rodriguez, Michael Wilde, Ian Foster, and Natalia Maltsev. Interoperability of gadu in using heterogeneous grid resources for bioinformatics applications. *IEEE transactions on information technology in biomedicine : a publication of the IEEE Engineering in Medicine and Biology Society*, 12(2):241–6, Mar 2008.
- [289] Guangzhong Sun, Jiulong Shan, and Guoliang Chen. Job scheduling for campus-scale global computing with machine availability constraints. *Computer and Computational Sciences, 2006. IMSCCS '06. First International Multi-Symposiums on*, 1:385– 388, 2006.
- [290] Hao Sun and Ramana V Davuluri. Java-based application framework for visualization of gene regulatory region annotations. *Bioinformatics*, 20(5):727–34, Mar 2004.
- [291] Xian-He Sun and Jianping Zhu. Performance considerations of shared virtual memory machines. *Parallel and Distributed Systems, IEEE Transactions on*, 6(11):1185–1194, 1995.
- [292] VS Sunderam. PVM: A framework for parallel distributed computing. *Concurrency Practice and Experience*, Jan 1990.
- [293] M Swanson, L Stoller, and J Carter. Making distributed shared memory simple, yet efficient. *In Proc. of the 3rd Int'l Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 2–13, Jan 1998.
- [294] F Tandiyari, S Kothari, A Dixit, and E Anderson. Batrun: utilizing idle workstations for large scale computing. *Parallel & Distributed Technology: Systems & Applications, IEEE [see also IEEE Concurrency]*, 4(2):41 – 48, Jan 1996.
- [295] Andrew S Tanenbaum and Maarten Van Steen. Distributed systems principles and paradigms. *Pearson Prentice Hall*, 2007. ISBN: 0-13-613553-6.
- [296] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program gpus for general-purpose uses. *ACM SIGOPS Operating Systems Review*, 40(5):325–335, Oct 2006.
- [297] M Taschuk. *Personal communication*, Newcastle University, UK., 2009.
- [298] Ian J Taylor. From p2p to web services and grids: Peers in a client/server world. *Springer*, 2005. ISBN: 1-85233-869-5 Computing Library.
- [299] JBoss Team. Hibernate. <http://www.hibernate.org>. [accessed 2009/04/19].
- [300] The Axis Development Team. Axis. <http://ws.apache.org/axis/>. [accessed 2009/04/19].
- [301] Top500 Team. Top 500 supercomputer sites. <http://www.top500.org/stats/list/31/archtype>. [accessed 2009/04/20].
- [302] D Thain, T Tannenbaum, and M Livny. Distributed computing in practice: the condor experience. *Concurrency and Computation: Practice & Experience*, Jan 2005.
- [303] Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Gathering at the well: creating communities for grid i/o. *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 58–68, Nov 2001.
- [304] J D Thompson, D G Higgins, and T J Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22(22):4673–80, Nov 1994.

- [305] Nicholas R Thomson, Debra J Clayton, Daniel Windhorst, Georgios Vernikos, Susanne Davidson, Carol Churcher, Michael A Quail, Mark Stevens, Michael A Jones, Michael Watson, Andy Barron, Abigail Layton, Derek Pickard, Robert A Kingsley, Alex Bignell, Louise Clark, Barbara Harris, Doug Ormond, Zahra Abdellah, Karen Brooks, Inna Cherevach, Tracey Chillingworth, John Woodward, Halina Norberczak, Angela Lord, Claire Arrowsmith, Kay Jagels, Sharon Moule, Karen Mungall, Mandy Sanders, Sally Whitehead, Jose A Chabalgoity, Duncan Maskell, Tom Humphrey, Mark Roberts, Paul A Barrow, Gordon Dougan, and Julian Parkhill. Comparative genome analysis of salmonella enteritidis pt4 and salmonella gallinarum 287/91 provides insights into evolutionary and host adaptation pathways. *Genome Res*, 18(10):1624–37, Oct 2008.
- [306] P Trunfio, D Talia, H Papadakis, and P Fragopoulou. Peer-to-peer resource discovery in grids: Models and systems. *Future Generation Computer Systems*, Jan 2007.
- [307] P Uppuluri, N Jabisetti, U Joshi, and Y Lee. P2p grid: service oriented framework for distributed resource management. *Services Computing, 2005 IEEE International Conference on*, 1:347– 350 vol.1, 2005.
- [308] S Vazhkudai, Xiaosong Ma, V Freeh, J Strickland, N Tammineedi, and S Scott. Freeloader: Scavenging desktop storage resources for scientific data. *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 56 – 56, Oct 2005.
- [309] S Vinoski. Distributed object computing with corba. *C++ Report*, Jan 1993.
- [310] W Vogels. Web services are not distributed objects. *IEEE INTERNET COMPUTING*, 7(6):59–66, Jan 2003.
- [311] Andreas von Bubnoff. Next-generation sequencing: the race is on. *Cell*, 132(5):721–3, Mar 2008.
- [312] G von Laszewski, I Foster, J Gawor, and P Lane. A java commodity grid kit. *Concurrency and Computation Practice and Experience*, Jan 2001.
- [313] Mathias C Walter, Thomas Rattei, Roland Arnold, Ulrich Gueldener, Martin Muensterkoetter, Karamfilka Nenova, Gabi Kastenmueller, Patrick Tischler, Andreas Woelling, Andreas Volz, Norbert Pongratz, Ralf Jost, Hans-Werner Mewes, and Dmitrij Frishman. Pedant covers all complete refseq genomes. *Nucleic Acids Research*, 37:D408–D411, Jan 2009.
- [314] Jingwen Wang, Songnian Zhou, Khalid Ahmed, and Weihong Long. LSBATCH: A distributed load sharing batch system. *Technical Report CSRI-286, Computer Systems Research Institute, University of Toronto*, Jan 1993.
- [315] B Wei, G Fedak, and F Cappello. A case for efficient execution of data-intense applications with bittorrent on computational desktop grid. *distributed computing*, Jan 2005.
- [316] Baohua Wei, G Fedak, and F Cappello. Collaborative data distribution with bittorrent for computational desktop grids. *Parallel and Distributed Computing, 2005. ISPDC 2005. The 4th International Symposium on*, pages 250 – 257, Jun 2005.
- [317] Baohua Wei, Gilles Fedak, and Franck Cappello. Towards efficient data distribution on computational desktop grids with bittorrent. *Future Gener Comp Sy*, 23(8):983–989, Jan 2007.
- [318] A Wespi and E Rothausser. Utilizing idle cpu cycles in a distributed computing system. *Aerospace and Electronics Conference, 1997. NAECON 1997., Proceedings of the IEEE 1997 National*, 1:173 – 180 vol.1, Jun 1997.

- [319] Brian White, Michael Walker, Marty Humphrey, and Andrew Grimshaw. LegionFS: a secure and scalable file system supporting cross-domain high-performance applications. *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, page 59, Nov 2001.
- [320] M Wiesmann, F Pedone, A Schiper, B Kemme, and G Alonso. Understanding replication in databases and distributed systems. *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464 – 474, Apr 2000.
- [321] Barry Wilkinson and Michael Allen. Parallel programming: techniques and applications using networked workstations and parallel computers. *Prentice Hall*, 1999. ISBN: 0136717101.
- [322] Mark Wilkinson, Heiko Schoof, Rebecca Ernst, and Dirk Haase. BioMOBY successfully integrates distributed heterogeneous bioinformatics web services. the planet exemplar case. *Plant Physiol*, 138(1):5–17, May 2005.
- [323] Mark D Wilkinson and Matthew Links. BioMOBY: an open source biological web services proposal. *Brief Bioinformatics*, 3(4):331–41, Dec 2002.
- [324] Paul Wilkinson, Nicholas R Waterfield, Lisa Crossman, Craig Corton, Maria Sanchez-Contreras, Isabella Vlisidou, Andrew Barron, Alexandra Bignell, Louise Clark, Douglas Ormond, Matthew Mayho, Nathalie Bason, Frances Smith, Mark Simmonds, Carol Churcher, David Harris, Nicholas R Thompson, Michael Quail, Julian Parkhill, and Richard H Ffrench-Constant. Comparative genomics of the emerging human pathogen *photorhabdus asymbiotica* with the insect pathogen *photorhabdus luminescens*. *BMC Genomics*, 10:302, Jan 2009.
- [325] Anil Wipat, Jennifer Hallinan, Daniel Swan, Morgan Taschuk, Matthew Pocock, Mike Cooling, Craig Turner, Mathew Robinson, Goksel Misirli, Hang Zhao, Arunkumar Krishnakumar, Jessica Tarn, James Murray, and Jane Hong. <http://2009.igem.org/Team:Newcastle>. [accessed 2009/09/27], 2009.
- [326] Adrianto Wirawan, Chee Keong Kwoh, Nim Tri Hieu, and Bertil Schmidt. CBESW: sequence alignment on the Playstation 3. *BMC Bioinformatics*, 9:377, Jan 2008.
- [327] Paul R Woodward, Jagan Jayaraj, Pei-Hung Lin, and Pen-Chung Yew. Moving scientific codes to multicore microprocessor cpus. *Computing in Science & Engineering*, 10(6):16 – 25, Nov 2008.
- [328] Ming-Wei Wu and Ying-Dar Lin;. Open source software development: an overview. *Computer*, 34(6):33 – 38, Jun 2001.
- [329] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. An approach for homology search with reconfigurable hardware. *Genome Informatics*, 12:374–375, Nov 2001.
- [330] Yoshiki Yamaguchi, Tsutomu Maruyama, and Akihiko Konagaya. High speed homology search with fpgas. *Pacific Symposium on Biocomputing Pacific Symposium on Biocomputing*, pages 271–82, Jan 2002.
- [331] J Yang, Y Wang, and Y Chen. Gpu accelerated molecular dynamics simulation of thermal conductivities. *Journal of Computational Physics*, 221(2):799–804, Jan 2007.
- [332] Jian Yang, Jinhua Wang, Zhi-Jian Yao, Qi Jin, Yan Shen, and Runsheng Chen. Genomecomp: a visualization tool for microbial genome comparison. *Journal of Microbiological Methods*, 54(3):423–6, Sep 2003.

- [333] X Ye and Y Shen. A middleware for replicated web services. *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, Jan 2005.
- [334] Yongjin Yeom, Yongkuk Cho, and Moti Yung;. High-speed implementations of block cipher aria using graphics processing units. *Multimedia and Ubiquitous Engineering, 2008. MUE 2008. International Conference on*, pages 271 – 275, Mar 2008.
- [335] T Ylonen. RFC 4252: The secure shell (SSH) authentication protocol. 2006.
- [336] M Zaki, M Ogihara, S Parthasarathy, and W Li. Parallel data mining for association rules on shared-memory multi-processors. *Supercomputing, 1996. Proceedings of the 1996 ACM/IEEE Conference on*, pages 43 – 43, Jan 1996.
- [337] E M Zdobnov and R Apweiler. InterProScan - an integration platform for the signature-recognition methods in InterPro. *Bioinformatics*, 17(9):847–848, Sep 2001.
- [338] Y Zhang, H Franke, J Moreira, and A Sivasubramaniam. Improving parallel job scheduling by combining gang scheduling and backfilling techniques. *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 133 – 142, Apr 2000.
- [339] Ming Zhao, Jian Zhang, and R Figueiredo. Distributed file system support for virtual machines in grid computing. *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 202 – 211, May 2004.