Managing Active Object Scalability on Distributed Memory

With a Case Study in Parallel VRML

Thomas Rischbeck

Advisor Prof. Paul Watson

Department of Computing Science
University of Newcastle upon Tyne

UNIVERSITY OF NEWCASTLE

Thesis submitted to the University of Newcastle upon Tyne in partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computing Science, January 2003.

Abstract

The de-facto standard for programming distributed memory parallel architectures are the PVM and MPI message passing libraries. While they are geared towards maximum efficiency, their low level of abstraction makes the programmer's task error-prone and reduces application portability. SODA is a novel programming model that presents a much higher level of abstraction and manages most low-level distribution and parallelism details implicitly. SODA is based on an extension of the active objects paradigm.

This work is structured into two main parts. In the first part we present a novel data-flow synchronisation mechanism for active objects that increases ease-of-use, efficiency, liveliness and correctness compared to previous approaches. SODA active objects are the units of concurrency and distribution and they make the underlying parallelism largely implicit. Details, such as mapping, communication and decomposition are transparent. This reduces programming overheads and increases portability. SODA is supported by a source-to-source translator and a Java runtime library. A set of micro-benchmarks is used to evaluate efficiency trade-offs.

The second part is a demonstration of SODA's benefits in the light of a complex, real-world application. It shows how SODA's active object concept can support object-oriented programming paradigms and therefore becomes a viable solution to large-scale real-world programs. Our example application is a parallel VRML execution engine implemented on top of SODA. We can observe a gain in productivity and programmability that outweighs the performance trade-off introduced by SODA's high level of abstraction. Beyond a proof-of-concept for SODA, the examination of potential parallelism in the VRML execution model is valuable in its own right. Since this is novel work, it is explored in more detail than would have been required for a mere case study.

Acknowledgements

This work would not have been possible without the encouragement and help of many other people. First and foremost I'm extremely grateful to my supervisor Prof. Paul Watson. He spent unbelievable amounts of time and patience to help me find the right way forward. There was always some freshly brewed tea in his office and room for discussion of the newest ideas. He took my constant pushing of deadlines with a very stoic mood, often spending his weekends with corrections and revisions. Thanks also to Prof. Pete Lee and Dr John Lloyd, who were providing regular feedback on the work and pointing out new directions to me. John was also instrumental in ensuring I could stay on in Newcastle after my initial ERASMUS exchange year. Einar Vollset provided great comparative work by developing a parallel VRML server on the basis of MPI. Bin Qu developed VRML models to test with the system.

My friends Dr. Budi Arief, Dr. Arnaud Simon, Richard Achmatowicz, Sam Aaron, Ram Chakrabarti, Dr Savas Parastatidis, Dr. Jim Webber and Dr. Denis Besnard provided continuous support, encouragement and many fruitful discussions. Estelle Chatard was another important contributor to my success, always there when needed and she never failed to motivate me and push me forward when I was desperate. Thanks for the fun time and the nice environment to everybody at Leazes Terrace, especially the co-wardens Chew, Celine Fitzmaurice, Mel Whewell and Ehsan Mesbahi. I'm indebted to everybody at HP Arjuna Labs where I worked for the past year. Dave Ingham was a great team leader, always supportive of my work. Not only could I access the infrastructure for my research work, but I also received support and flexible spare time to finish my PhD work.

In addition I would like to express gratitude to my parents and family in Germany. My uncle Hans Steigenberger was frequently donating financial assistance for the "poor researcher" and great to discuss every aspect of life-except for computers. My sister and her family were unbeatable for delivering bad jokes.

This work was funded for three years by the Engineering and Physical Sciences Research Council.

Table of Contents

List of Figures

List of Tables

List of Codes

Glossary

AOI	Area-Of-Interest (Management)
BSP	Binary Space Partitioning
COOP	Concurrent Object-Oriented Programming
CORBA	Common Object Request Broker Architecture
COTS	Commodity-Off-The-Shelf
COW	Cluster of Workstations
DSM	Distributed Shared Memory
FIFO	First-In First-Out
HPC	High Performance Computing
HTML	Hypertext Mark-up Language
JDK	Java Development Kit
JNI	Java Native Interface
JVM	Java Virtual Machine
LAN	Local Area Network
LRU	Least Recently Used
MIMD	Multiple-Instruction, Multiple-Data
MPP	Massive Parallel Programming
NET-VE	Networked Virtual Environment
NOW	Network of Workstations
NUMA	Non-Uniform Memory Access
OOP	Object-Oriented Programming
RAID	Redundant Arrays of Independent Disks
RMI	Remote Method Invocation
RTS	Runtime System
SAM	SODA Abstract Machine
SIMD	Single-Instruction, Multiple-Data
SISD	Single-Instruction, Single-Data
SMP	Symmetric Multi-Processor
SODA	Search of Decaffeinated Acronym
SSI	Single System Image
VRML	Virtual Reality Modelling Language
X3D	Extensible 3D Language
XML	Extensible Mark-up Language

Chapter 1

Introduction

The use of computers is becoming increasingly ubiquitous in our society. Early use of computers was restricted to scientific computation and data processing, but application fields are now wide open. We expect computer systems to solve more complex problems more efficiently. This leads to an increased demand on computational resources.

In theory, parallel processing across a cluster of workstations is one way to address this demand. In practice however, the development of efficient, maintainable, correct and affordable parallel applications has proven to be a non-trivial task. Part of the challenge lies in the distributed nature of the execution platform and inadequate programming methodologies. Frequently, programmers have to concentrate on the management of parallelism rather than on algorithm issues.

This dictates the need for novel programming methodologies that help to decompose large and complex programs and run them efficiently on a parallel machine. Concurrent object-oriented programming (COOP) is a strong contender in this regard. The object-oriented abstraction and information hiding principles have the potential to mask underlying complexities. The stimulus behind this work has been to make distributed-memory programming a less daunting task and expose a cluster as a unified resource to the developer. For this purpose we address the shortcomings of existing programming systems and provide solutions to some of the problems encountered.

1.1 Parallel Computation

The motivation to solve problems faster has been a central issue throughout the history of computing. Pfister [136] identifies three possible approaches, each of which gave rise to a major field of research. He explains these by drawing an analogy of "working harder", "working smarter" or "getting help" (see Table 1-1).

Table 1-1 Human-Computer Analogy to Achieve Higher Performance [136].

Human Approach	Computer Analogy	Requirements
Work Harder	Use *faster hardware*, e.g. processors with shorter cycle times that can execute more instructions per time.	**Faster Processors**
Work Smarter	Express the original problem with *optimised algorithms* to utilise available hardware resources more efficiently.	**Expert Knowledge, Suboptimal Algorithms**
Get Help	Use several resources at the same time and in parallel, e.g., use *parallel processing* to solve the problem.	**Multiple Processors**

In the 1970s and 80s it was believed that computer performance was best improved through "working harder" [31]. Indeed, over the last decades we have witnessed an exponential increase in processor speed, well known as *Moore's law*. This trend, driven by shrinking component size and increasing transistor count, is predicted to continue for some years to come [115]. Nevertheless, physical limits, such as the speed of light, quantum effects and heat dissipation, will ultimately set a barrier to further gains. Economic considerations could set feasibility limits even earlier [115][1].

Often, the lack of fast hardware was seen as an encouragement to develop optimised algorithms or to "work smarter". As many standard algorithms that are important in performance-sensitive applications tend to be very well coded nowadays, only marginal performance gains can be expected from this approach. This indicates that neither "working harder" nor "working smarter" is likely to yield a long-term development path.

"Getting help" augments the other two approaches and revolves around combining the power of multiple processors. Parallel processing is generally seen as the basis for high performance computing (HPC) and as key enabler to yield future *Petaflops* performance [161]. Impressive results have been demonstrated on numerical applications in science and engineering [65]. Parallelism has also proven successful in the database domain; many vendors nowadays provide parallel versions of their products.

Despite these successes and the conviction that "the future is parallel" [61], general-purpose parallel processing has not yet reached the mainstream. One reason is that massively parallel computers tend to be very expensive as they are largely built from proprietary components. The other reason is that program development is more difficult.[2]

[1] The *Second Law of Moore* states that development costs for each new chip generation grow exponentially with processor performance.

[2] On a more philosophical note, Bräunl's observation [28] is interesting: Parallel programming is difficult because humans tend to think sequentially in terms of cause and effect. However, while our

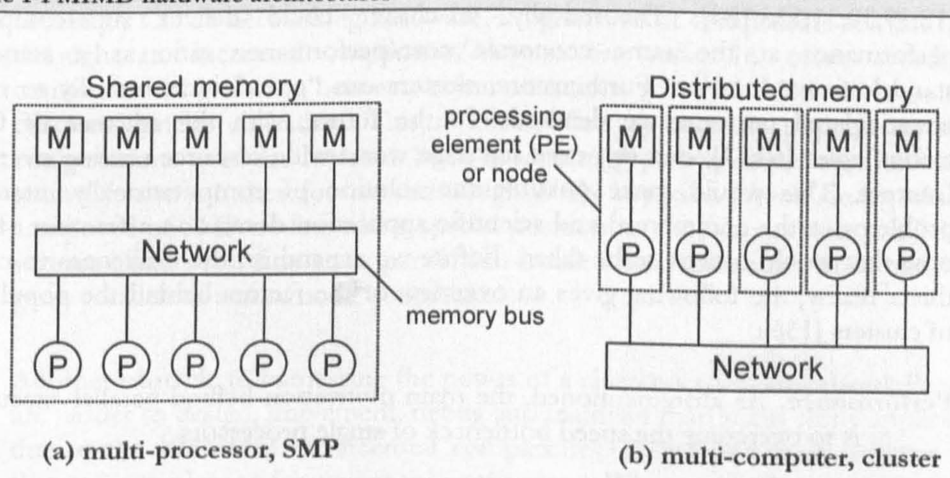
The following section will give a brief overview of basic concepts of hardware parallelism, before we focus on clusters of workstations as a low-cost alternative to conventional massively parallel supercomputers.

1.2 Parallel Architectures

Parallel architectures are distinguished in their multiplicity of *instruction streams*–i.e., simultaneous *activities* or *threads of control*–they support. Either the processors can all work in *synchronous parallelism* or the processors can work in *asynchronous parallelism*. Synchronous parallelism is supported by *single-instruction multiple-data* (SIMD) systems, such as vector or array computers. SIMD is restricted to regular, mainly numeric problems, where the same algorithm is applied to a large number of different data elements. All processors work in lock-step fashion steered by a central control unit. In *multiple-instruction multiple-data* (MIMD) systems, the processors function independently in *asynchronous parallelism.*[3] MIMD is more flexible, in that every processor can perform a different activity.

MIMD can be further subdivided into systems with shared memory (Figure 1-1(a)) and distributed memory (Figure 1-1(b)). In a *shared memory MIMD* (also called SMP or *multi-processor*), all processors have access to the same memory with a *single address space.* In a *distributed memory MIMD* (also called *multi-computer*) every processor has its own local memory, i.e., the machine consists of autonomous processor-memory pairs (so-called *processing elements*) connected through a network. Access to remote memories is significantly more expensive than to the processor's local memory.

Figure 1-1 MIMD hardware architectures



(a) multi-processor, SMP **(b) multi-computer, cluster**

stream of consciousness appears to be sequential, it is the massive parallel processing power of billions of neurons in the human brain, which generates its astounding potential, despite the low firing rate of the switching elements. This can be seen as a biological "proof of concept" for parallel processing.

[3] In Flynn's Classification, SISD is equated with conventional sequential computers, while MISD does not have any practical implementations.

MIMD programs need to provide separate *activities*, in order to keep all processors busy. In operating system terms, activities can be implemented as either *processes* or *threads*. Processes have a private memory address space, protected by the operating system. Threads can only exist within a process and have no private memory. As a result, threads have a much lower *context switching overhead* than processes.

The number of activities in a parallel program is referred to as its *potential parallelism*. The number of available processors, in contrast, is the hardware's *physical parallelism*. *Parallel slackness* is defined as the ratio of potential parallelism to physical parallelism. Of course, this value is dependent on the target machine. A parallel program with sufficient parallel slackness, can achieve an *overlapping of computation and communication*.[4] through activity multiplexing. This is useful, since in parallel programs, blocking is more frequent, due to increased communication and synchronisation needs [157]. On the other hand, if parallel slackness of a program is too high, individual processors will waste too much time on context switching between activities. Ideally, the parallel slackness should be constant across architectures, i.e., the program should expose *adaptive parallelism* at runtime.

We will now turn our attention to clusters of workstations as a particularly interesting exemplar of a distributed memory MIMD architecture.

1.3 Towards Scalable and Affordable Supercomputing

Clusters of workstations (COW) with high-bandwidth low-latency interconnects are becoming increasingly ubiquitous in today's networked computing environments. This trend has generated much interest in using a cluster as a unified resource for tackling large-scale and complex compute problems [12;19;30;31;136;183]. Theoretically, a cluster could deliver supercomputer performance at the same economic cost/performance ratio as a standard standalone workstation. Furthermore, clusters can "grow" incrementally to meet unanticipated performance demands. In the future, with the advance of Grid technologies [62;71], one can even envisage world-wide resource sharing over the Internet. This would make possible the solution of computationally intensive problems in the commercial and scientific application domain in a fraction of the time that would otherwise be taken. Before we expand on the challenges to make this a reality, the following gives an overview of the factors behind the popularity of clusters [136]:

Performance. As aforementioned, the main motivation behind parallel execution is to overcome the speed bottleneck of single processors.

Availability. Clusters are built from autonomous, independently failing components. This natural redundancy can be exploited to provide high availability systems. On the other side of the coin, the probability of a single node failure increases exponentially with the cluster size.

[4] i.e., remote communication of one activity can be overlapped by useful computation of another activity.

Cost/Performance Ratio. Clusters are built around commodity hardware (COTS). They can therefore extend the cost/performance ratio of current workstations into the area of high performance computing.

Incremental Growth. The entry price for a COW, depending on the number of nodes, is small, while at the same time leaving the potential for growth.

Scalability. A size of a cluster seems virtually unrestricted; there are no theoretical limits of how many workstations can be combined through a network. This has led to the vision of a "global supercomputer", pooling together millions of participating hosts.

Scavenging. In many organisations base hardware is often readily available and not fully utilised. Systems are conceivable that spread work over a local network and use the spare time of otherwise idle workstations. This is extensively studied in the NOW project [12].

Over the next years, we can expect cluster-based HPC to take a pervasive position throughout industry and academia [30;31]. We expect that the growing recognition of clusters will lead to two developments: More novice and inexperienced users will want to develop HPC programs. In addition, we expect a huge diversification in the range of applications. By contrast, conventional supercomputer programming has been geared towards highly skilled programmers and the support of numerically intensive, regular and *transformational*[5] applications. The challenge now is to provide an easy-to-use programming environment, which simplifies cluster programming and enables a larger user base to effectively use parallel architectures.

Despite recent advances, clusters still have significant communication latencies caused by COTS networking hardware. Programs must compensate for this latency through an increased *computation/communication ratio.* In other words, the computational *grain size* between communications must be relatively large for an efficient system utilisation. This is exemplified by recently popular distributed computations over the Internet. Such *embarrassingly parallel computations* have a large grain size and can therefore amortise high WAN latencies. *Hybrid clusters,* which are composed of SMP nodes, take a special position, as they can exploit parallelism at various grain sizes: fine-grain parallelism inside a multiprocessor node and larger-grain parallelism across node boundaries.

Another obstacle to harnessing the power of a cluster is software-related: Programs are harder to design, implement, debug and maintain than stand-alone applications, due to the additional architectural complexities. Developers must address issues that are not relevant for sequential applications: Where and when activities should be started, how data is communicated and how asynchronous processing is coordinated. As a result, it is difficult to build programs that are correct, portable, efficient and inexpensive [153]. The full potential of clusters as an inexpensive and powerful computing platform is not realised.

[5] i.e., converting a set of input values into a set of output values, without further runtime-interaction. *Reactive* programs, in contrast, deal with dynamic values only available at runtime.

As Pfister notes, "hardware only provides the potential for high performance, the fulfilment lies in software." [136]. This raises the need for a programming methodology that simplifies development by shielding a programmer from implementation details. Ideally, a high-level programming methodology would make architectural artefacts **transparent** while still providing satisfactory runtime **efficiency**. Many traditional parallel programming techniques have been inadequate in this respect: they are process-centric, non-modular and expose the underlying parallelism, requiring the developer to take detailed decisions over where and when to perform parallel activities.

To this end, parallel software development is increasingly being addressed with implicit and object-oriented solutions. These approaches are introduced in the following three sections. As will be shown, this combined approach strikes a successful balance between extraction of massive parallelism, high performance, and ease of use.

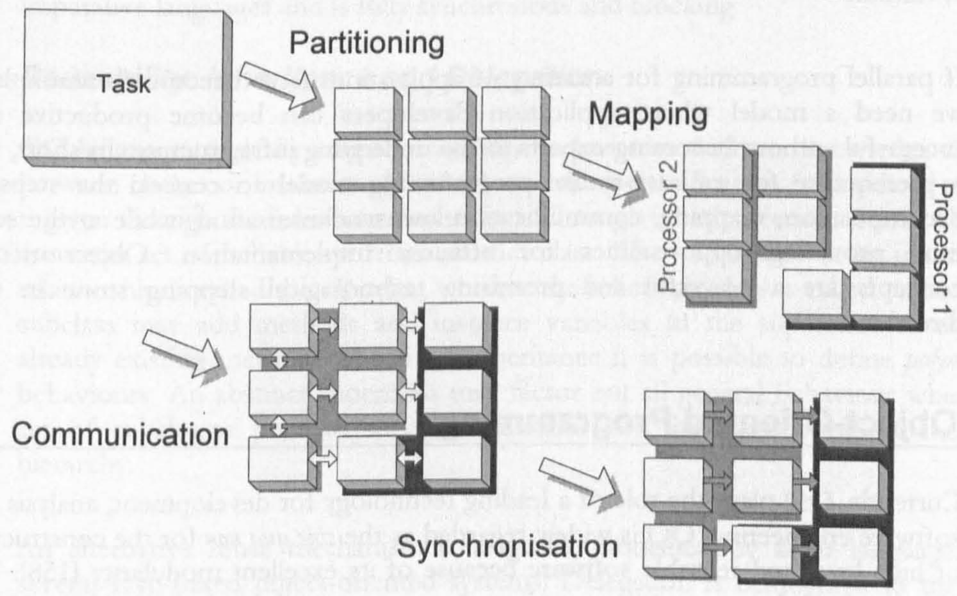
1.4 Explicit vs. Implicit Parallelism

A *programming model* is an interface separating high-level properties from low-level machine details. A programming model provides the image of an *abstract machine* to developers. The aim of such abstraction is to simplify program development while hiding underlying architectural complexities. Developers can so focus on program development for the abstract machine, whereas it is the implementers' task to provide an efficient abstract machine layer over the physical hardware. This reduces programming overhead and can provide better portability, since the abstract machine can conceal heterogeneity.

Parallel programming models can be classified into the two categories of *explicit parallelism* and *implicit parallelism*. In explicit parallelism the programmer is responsible for controlling all aspects of parallelism at a low level of abstraction. In the implicit model, parallelism is completely transparent. i.e., a sequential program is automatically converted into a parallel version and fed to the hardware.

Bräunl [28] describes the common programming "recipe" for explicit parallelism: find subtasks, allocate them to processors and define their communication structure and synchronisation points (see Figure 1-2). Practitioners following this recipe are forced to constantly "reinvent the wheel". They spend more time focussing on parallelism management than on high-level algorithms. As a result, development becomes cumbersome, error-prone and inefficient [135], "effectively [ruling] out scalable parallel programming." [157]. These problems are exemplified in the *message passing* model (see §2.1.4).

Figure 1-2 A "Recipe" for Explicit Parallel Programming



Higher-level, implicit programming techniques, try to remove the burden of explicit parallel programming from the programmer. Here, the user does not specify and thus cannot control, underlying parallelism in the physical architecture. While ideal for the programmer, this approach has its limitations, especially in the context of non-declarative languages and clusters as target architectures. Parallelising compilers commonly only manage to extract very fine-grain parallelism more suited to shared memory systems [28]. The challenge therefore is to find a model that is sufficiently abstract to simplify programming but at the same time not too abstract that it becomes difficult to provide efficient implementations on real architectures [157].

While explicit parallelism gives programmers a high degree of control over the hardware, it is inadequate for tackling medium to large-size or irregular problems, or any realistic software projects for that sense. In sequential computing, the same could be said of assembler languages in the days before efficient compilers. Assemblers offered detailed control over every processor register and direct exposure to the hardware. In spite of this, high-level languages are now the preferred choice for sequential software development as they hide complexity and provide modularity. Similarly, if parallel programming is to become mainstream, it needs to be made easier for the average programmer. Otherwise it will remain relegated to few high-value applications for which investment of great development efforts can be amortised.

A further advantage of implicit parallelism is that an implementation has considerably more flexibility in terms of compile-time or runtime optimisations. This is important for *reactive* applications, where optimal decisions to mapping and decomposition often depend on the execution environment, such as runtime parameters and target architecture. Unpredictable processing times of subtasks, for instance, make every static allocation suboptimal. In this situation, optimum processor utilisation may only be obtained through dynamic load balancing of tasks at runtime.

If parallel programming for a variety of applications is to become commonplace, we need a model where application developers can become productive and successful without becoming experts in the underlying infrastructure. In short, it is a prerequisite for an easy-to-use programming model to conceal the steps of decomposition, mapping, communication and synchronisation, while at the same time providing opportunities for efficient implementation. Object-oriented concepts are a powerful and promising technological stepping stone in that direction.

1.5 Object-Oriented Programming

Currently, OO plays the role of a leading technology for development, analysis and software engineering. OO is widely regarded as the *sine qua non* for the construction of high-level and reusable software because of its excellent modularity [158]. The object-oriented programming paradigm encourages modular design and knowledge sharing (in particular code reuse). The concept of object-oriented programming has its roots in SIMULA,[6] developed in the late 60's in Oslo [48] (see also [142] for an introduction). Since then it was further developed as an important software engineering methodology [180] [117]. In the following sections we highlight the mainstays of the OO model as these are important for the further discussion.

Classes and Encapsulation

An OO program consists of a dynamic collection of objects that communicate with each other to drive the computation forward. Each object is an instance of a *class* (or abstract data type, ADT) which serves as an object template. The class describes instance variables and methods that operate on them. These internals are *encapsulated* through the method interface and not directly visible from the outside. Such encapsulation or *information hiding* minimises interdependencies among separately written classes and allows class-internal changes to be made safely without affecting existing clients. The user of an object need not be concerned with the class itself or its instance objects but only with the abstract interface.

Message Passing

Computation proceeds as objects invoke methods on other objects. In the following we will use the terms *client* and *server* for the two objects engaging in a method call. A message is sent from client to server object when a method should be invoked on the server. This process is known as *passing messages* in the OO terminology. A message contains a method identifier and a set of parameters that are consumed by the server's method. However, the nomenclature of

[6] SIMULA was designed for simulations, and the needs of that domain provided the framework for many of the features of object oriented languages today.

communication is misleading in a sense that it does not imply any asynchronous communication. In fact, the invocation model is identical to procedure activation in imperative languages and is fully synchronous and blocking.

Reusability: Inheritance and Delegation

OO languages typically adopt a *reusability mechanism* to facilitate knowledge sharing between classes or instances. A natural modelling classification for kinds of objects is given through the *inheritance* mechanism. Through inheritance a class may derive a common set of behaviours from another class. The initial class in an inheritance relationship is called the *superclass*, whereas the inheriting class is called *subclass*. The subclass may add methods and instance variables to the superclass or *overwrite* already existing methods. Based on inheritance it is possible to define *polymorphic* behaviours: An abstract superclass may factor out all general behaviour whereas a set of subclasses provide specific behaviour for the lower level of the type hierarchy.

An alternative reuse mechanism is *delegation*, pioneered by actor languages and several Lisp-based object-oriented systems. Delegation is orthogonal to the class concept. When an object does not know how to respond to a message, it can delegate this message to one of a set of designated objects. The original server adopts the role of a client for the selected delegate object.

Dynamic Binding

In general, which method is to be invoked is not known until the message's dispatch time because a method in an object may share the same name with one in another object. Depending on the runtime type of the destination object, the appropriate method definition is chosen dynamically. The exact semantics of such *dynamic binding* may vary from language to language, depending on how the dynamic method lookup is implemented.

1.6 Concurrent Object-Oriented Programming

Object-orientation is a useful methodology to attack program complexity. However, most object models do not address concurrency and distribution. Despite adaptation of a message passing metaphor and the implied autonomous activity of each object, most OO languages remain sequential. One object is active at a time and activity is transferred from one to another, piggy-backed onto message-passing. This is for historical reasons, which mapped OO programs onto sequential target architectures. In this sense, COOP appears as a generalisation of OO programming by giving further autonomy to objects.

The *active object* or *actor* paradigm identifies objects as the unit of concurrency and distribution and associates synchronisation between objects at the message passing level. This integration is fruitful because it preserves the modularity and simplicity of OO while enforcing self-containedness and autonomy of objects. Autonomous activity of each object exhibits inherent concurrency between them

[6;10;29;37;130;188]. This paradigm also makes it very natural for humans to reason about programs [80;101]. The active object paradigm also serves as a foundation for higher-level *agents* systems [76].

At the design level, the operational view of an application as a collection of cooperating and communicating objects has clear advantages; it provides a clean conceptual model that can express the components of an application and their interaction at a high level of abstraction. However, the marriage between object-orientation and concurrency has been difficult [113] [91;133] [116]. A notable integration obstacle is the seeming incompatibility of inheritance and concurrency, an observation for which Matsuoka and Yonezawa [113] coined the term *inheritance anomaly*.

In addition, active object systems rely on *cluster middleware* for their implementation. Such a software layer should reflect the natural hardware scalability. This means that software overheads should not grow with the number of the nodes; central components must be avoided, as they could become potential bottlenecks. Finally, the potential parallelism of a program should dynamically adapt to physical parallelism available in the cluster. No changes to a program should be required whether it is run on just a single node, or a cluster of 10 or 10,000 workstations; all available processors and network interconnects should be used to optimum efficiency and the program should yield speedups if the program is sufficiently complex and long-running

In the past, implementations have often failed to make a distinction between activities in the programming model and in the execution model. Existing active object implementations are therefore often heavy-weight and do not expose adaptive parallelism. Insofar, implementations were not well suited to scalable distributed memory architectures. We address these shortcomings with the development of a new programming model, called SODA. The operational semantics of SODA are provided by a runtime system that aims at maximum efficiency in adaptation to available hardware parallelism. A detailed description of the contributions of the presented work is given in the following section.

1.7 Thesis Contributions

This thesis introduces a set of contributions that address the problems identified above. These aim at the provision of an implicit and object-oriented programming model for the efficient utilisation of cluster installations.

- **The novel SODA Programming Model and Runtime System** as a basis for further research and development built on an extended active object model. SODA combines ease-of-use through a combination of COOP and implicit parallelism concepts (see Table 1-2). Features of the programming model and RTS are closely interrelated and include:
  - **Futures and Funnels.** Funnels are a novel mechanism to support data-driven implicit synchronisation of asynchronous Future-based calls.

  Funnels avoid cyclic deadlocks in the context of atomic active objects and can express data dependencies in a straightforward manner. This overcomes the limitations of atomic active objects while maintaining their ease-of-use advantages.
- **Detached Methods** as a means to increase liveness of atomic active objects while keeping the benefits of their mutually exclusive invocation semantics.
- **Ferenczi Guards.** To our knowledge, SODA is the only active object system to implement Ferenczi Guards [59], a mechanism to circumvent the inheritance anomaly.
- **Implicit Decomposition.** Active objects are *potential* units of concurrency. The implementation therefore has considerable freedom for runtime optimisations, such as *inlining of invocations* and *thread-multiplexing of active objects.* This approach allows a dynamic adaptation to the physical parallelism available at runtime.
- **Implicit Mapping.** *Location-Transparent Active Objects* allow the portability of SODA programs over a range of different cluster configurations. This is also a prerequisite for dynamic load balancing through the *Migration of Active Objects.*

- **Case Study: Parallel VRML Execution Engine.** In this case study, a large irregular server application is built on top of SODA to examine its ease-of-use and performance for real-world applications. Attendant contributions include:
  - **Analysis of Parallelism inherent to the VRML97 Execution Model.**
  - **Implementation of a Scalable VRML Server** on the basis of parallel event cascade evaluation.
  - **A novel, Client-server based approach to VRML.** Allows multiple users with varying hardware configurations access to the scalable simulation server. This is based on view frustrum culling and dynamic level of detail selection.

Table 1-2 Explicit Parallelism Issues (see Figure 1-2) addressed in SODA.

Issue	Soda Approach
Decomposition	Dynamic matching of potential to physical parallelism through adaptive inlining of active method calls and multiplexing of active objects onto threads (lightweight active objects)
Mapping	Active Objects in SODA are location-transparent. Dynamic load balancing through migration of active objects is designed, but not implemented.
Communication	Object-oriented model, communication only through messages to objects (i.e., method calls). All communication is based on the well-established call-reply model (i.e., every method call returns a result or an exception).
Synchronisation	Futures and Funnels, based on a dataflow abstraction.
Scalability	Dynamic decomposition (see above) allows varying degrees of scalable parallelism in adaptation to the availability of resources in the underlying runtime environment. The cluster can be grown without change to the program.
Portability	Java based runtime system to accommodate platform heterogeneity

1.8 Dissertation Outline

The scope for the remaining chapters is as follows:

Chapter 2 gives a brief overview of some popular parallel programming models and their level of abstraction. We identify the active object approach as reflecting our goals of scalable, easy-to-use parallelism. We then discuss various models in this category and their implementations on distributed memory architectures, focussing on a combination of features, which we believe to be critical to widespread user acceptance.

Chapter 3 introduces the SODA programming model and its novel features, which combine active objects with a dataflow-like, non-blocking synchronisation mechanism. We illustrate, by example, how SODA is used for highly implicit and object-oriented parallel programming. Further, we explain design choices taken and give the reasons for doing so.

Chapter 4 covers the implementation of the runtime system, which is responsible for mapping the SODA programming model onto a real distributed memory architecture. This contains a detailed analysis of the protocols and algorithms used within SODA and shows how adaptive parallelism is exposed.

Chapter 5 then presents a performance evaluation of SODA, which serves as justification of the design choices in SODA. We present a set of micro-benchmarks that cover typical application patterns.

Chapter 6 presents a case study into building a large-scale, irregular and interactive server application on top of SODA. The case study focuses on the design and implementation of a parallel execution engine for the VRML event model. The algorithms exploit parallelism inherent to the VRML execution model. This allows the execution of large-scale VRML scenes with many participating users. We examine SODA's ease-of-use for developing this system and give some performance figures.

Finally, **Chapter 7** concludes the thesis with a summary and a discussion of some open-ended issues.

Chapter 2

Review of Related Work

In this chapter we analyse related work, which aimed at combining of object-orientation and concurrency. To that end, we focus on the ease of use of existing programming models and systems as well as on the possibility of their efficient implementation on distributed memory. To put object-oriented concurrency into a wider context we briefly examine alternative programming methodologies. The object-oriented models can be seen as an approach "from the middle", neither explicitly exposing parallelism nor making it completely implicit. One main focus of our review lies on integrative approaches that merge the concepts of concurrency and objects into *active objects*. We explore the associated design space and populate it with evaluations of existing work. This review is by no means exhaustive. For a more complete survey of more than 100 languages in the concurrent object-oriented category, see [137;138].

2.1 Parallel Programming Models

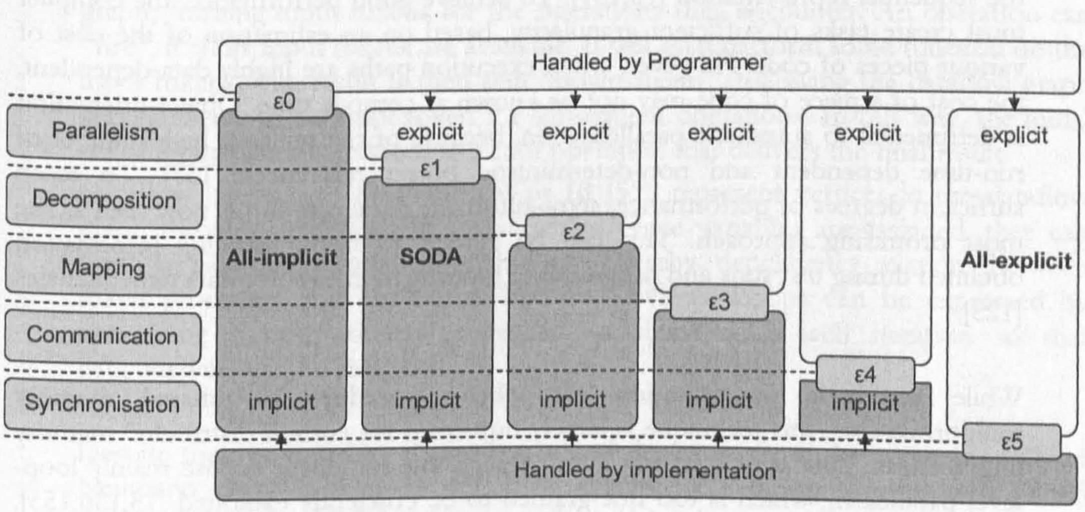
We want to provide a more general backdrop for the discussion of concurrent object-oriented programming. This section therefore gives a general overview of the different fundamental approaches to achieve parallel processing.

One fundamental distinction relates to the degree to which physical parallelism is transparent. Explicit parallelism requires a programmer to specify in detail every aspect of parallel execution. Implicit parallelism simplifies programming but relies on complex compiler technology. Another classification can be taken according to the memory access concept. Each of the two possible memory arrangements for parallel machines (see §1.2) have led to a different programming paradigm. One is based on the view of distributed memory access, the other one uses shared memory. We return to this second case later. First we focus on the tension between explicit and implicit models.

2.1.1 Skillicorn's Classification

Skillicorn and Talia [157] present a classification that places explicit and implicit parallelism at the extremes of a spectrum (see Figure 2-1). This spectrum is structured by increasing responsibilities of a parallel programmer. From explicit to implicit direction, these encompass synchronisation (ε5), communication (ε4), mapping (ε3), decomposition (ε2) and "parallelism awareness" (ε1) (analogous to Figure 1-2, A "Recipe" for Explicit Parallel Programming). A model's ease-of-use is increasing the further it is placed towards the implicit end of this scale. In the most attractive world, a programmer leaves all low-level details to a parallelising compiler (ε0), while being able to fully concentrate on algorithms. This is beneficial to a programmer, but has practical limitations, especially in the context of distributed memory machines. Explicit parallelism (ε5), on the other hand, exposes underlying interactions at great detail, which can limit its portability and usability for large or irregular applications. We will examine the two extremes of implicit and explicit parallelism and then focus on active objects, which we consider an approach "from the middle".

Figure 2-1 Skillicorn's Classification of Parallel Programming Models



2.1.2 Converting Sequential into Parallel Programs

The parallel solution for a given problem is much harder to develop than the equivalent sequential application. Moreover, much existing code is in sequential form, not targeted at parallel architectures. These observations prompted much research into the development of automatically parallelising compilers. Such compilers implement an algorithmic way of transforming a sequential, imperative program (ε0) into a semantically equivalent parallel version.[7]

Most of this work is based on the idea of *extracting* implicit parallelism from loops or multi-way recursive methods. For example, techniques have been devised to extract SIMD parallelism from "dusty deck" FORTRAN programs into vector-parallel form [7;102;112].

Similar ideas have been exploited in the *High Performance Java* project [23;24]. Sequential programs are (semi-) automatically converted into parallel code using the standard Java multithreading mechanism. **JAVAR** [24] is a source-to-source restructuring compiler that relies on explicit annotations. **JAVAB** [23] goes a step

[7] Pfister [136] also talks humorously about "*AMO compilers*" in this context: "You stuff in a sequential program on one end, **A** **M**iracle **O**ccurs and a parallel program comes out at the other end."

further and extracts implicit loop parallelism without programmer intervention directly from Java bytecode. No access to the program's original source code is required.

Parallelising compilers construct a data dependency graph of the program based on *dependency analysis*. Parallel execution is safe when two pieces of code are not interdependent. For languages that allow mutation of shared variables dependency analysis can be quite complex and opportunities for parallelism may be missed. A compiler does not have sufficient domain level knowledge to determine whether insolvable data dependencies are inherent to the application or just an artefact of the sequential representation [75;157]. To achieve good performance, the compiler must create tasks of sufficient granularity, based on an estimation of the cost of various pieces of code. However, when execution paths are highly data-dependent, the cost of a piece of code may not be known at compile time. Object-orientation is detrimental to automatic parallelisation, because of the typically high number of run-time dependent and non-deterministic object references [137]. To reach sufficient degrees of performance, semi-automatic parallelisation is now seen as the most promising approach. This can be guided by both, profiling information obtained during test runs and programmer interrogation to solve data dependencies [125].

While automatic parallelisation can obtain speedups on shared memory multiprocessors, this technique is largely unproven for current distributed memory architectures. The reason is that auto-parallelising compilers expose mainly loop-level parallelism, which is too fine-grained to be efficiently exploited [75;136;155]. In addition, programs for which these techniques are effective are often restricted to the domain of very regular, numerical applications [16;137;157] (p. 89). For general sequential algorithms, they lack the developer's domain-level knowledge to generate efficient parallel algorithms. Philippsen gives the example of a sequential sorting algorithm: It would be impossible for a tool to automatically generate one of the well-known parallel algorithms without "understanding" the program specification. By the same token, Skillicorn argues that responsibility for parallelism awareness rests best with the developer, whereas the implementation takes over lower-level tasks (i.e., category ε1). In fact, some algorithms might be expressed more naturally in a parallel fashion; Bräunl, for instance, gives the example of a vector dot product [28].

A different approach is taken by **High Performance Fortran (HPF)**. In this language, the programmer first writes a sequential algorithm. As a second step, *compiler directives* are added that specify opportunities for parallelism. Based on such explicit parallelism annotations, the compiler can generate highly performing parallel code.

2.1.3 Declarative Languages

Besides automatically parallelising compilers, another approach to all-implicit parallelism is given through declarative languages, with the subcategories of dataflow, logic and functional languages. Most of these are, at least to some degree, *side effect free*, which means that the result of a function call depends solely on the values of its input parameters. This precludes the concept of state. The absence of side effects makes analysis and reasoning about a program much easier. In particular, data dependencies are immediately obvious from the program specification. Impressive performance results have been demonstrated for dataflow languages on shared memory systems [34]. However, declarative languages are less practical for developing programs that are either best expressed or more efficiently expressed by using mutable data, or can benefit from the advanced concepts of inheritance and encapsulation used in OOP [14].

Dataflow languages [55;78] describe a program as a set of data-driven operations, linked together by a directed *dataflow graph*. Data "flows" along the edges of this graph, forming input tokens for the operations they encounter. An operation can "fire", if all its input tokens are available. It will then perform some function on the input tokens. The result is then sent "downstream", following the dataflow graph and can serve as an input token for subsequent operations. In this way, the initial set of input values is reduced to a root operation that delivers the final result.

Dataflow languages [78], like **Sisal** or **Id** [55], represent vertices in the dataflow graph through single-assignment variables. Once variables are assigned, they can serve as parameters to other operations. Thereby, dependency relations between token producers and consumers become obvious. Loops can be expressed by associating a new context (refreshed variables) with each iteration so that dependencies can be resolved..

Despite their potential as alternative to and departure from the conventional von Neumann programming model [17], declarative languages have not gained significant impetus [157]. Skillicorn explains this reluctance of adaptation with two factors: first, human cognition appears as a sequentially ordered process; one aspect of this is the concept of state and sequential modifications to this state. This concept cannot be matched adequately by declarative constructs. A second point is the "inertia" of programmers and the related difficulty of achieving a "critical mass" of available software, programmers and language implementers.

2.1.4 Distributed Memory – Message Passing

The de facto standard programming environments for clusters and other distributed memory machines are doubtlessly **PVM** [68;165] and **MPI** [53;143]. Both use an execution model based on processes that communicate by way of message passing. The message-passing model exposes all-explicit parallelism at a low level of abstraction from the physical hardware (ε5). An application is expressed as a set of communicating tasks, which are mapped onto the available hosts (usually in a round-robin fashion). Tasks can engage in peer-to-peer or collective communication by sending discrete messages to each other. Fundamental send and receive operations are provided in both blocking and non-blocking variants. Characteristically for explicit parallelism, decomposition, communication and synchronisation are major parts of the algorithm. While the message passing model is conceptually extremely simple in practice its explicit parallelism proves error-prone and places a huge burden on the programmer.

Blocking communication allows the definition of explicit synchronisation points amongst tasks. This is problematic, because whenever a task is blocked waiting on a communication event, valuable processor resources are lying idle. To avoid this situation, efficient message passing schemes aim at hiding communication time with other useful computation. This strategy of overlapping computation and communication can be implemented by multi-threading on every host. However, current implementations of PVM [68] and MPI [53] are not thread-safe and map tasks onto heavy-weight operating system processes, which incurs high context-switching overheads. An alternative to such multi-tasking or multi-threading is given through *active-waiting schemes*. For example, a task could engage in independent computation until it is notified through an asynchronous signal about the termination of a communication event. This approach requires considerable sophistication in the control program.

Object-Oriented MPI (OOMPI) [159] is an approach to encapsulate the functionality of MPI in a C++ class library. This is done via member functions of data, communicator and message objects while preserving MPI semantics at a lower level. Through class wrappers, simpler access to sends and receives is possible and convenient C++ stream interfaces can be used. **MPIJava** [18] and **JPVM** [60] are Java libraries, which wrap message passing layers using JNI [163]. **JMPI** [52] is an implementation of MPI in Java.

2.1.5 Shared Memory – Threads

Distributed memory programming is hard, because remote memory locations must be explicitly controlled and updated. It is widely believed that shared memory provides an easier programming model, since communication is much simplified through the use of shared variables. A set of threads operate on these variables, each defining a separate flow of control.

In multithreaded programs it is necessary to protect critical regions from being entered by more than one thread at a time. Such a violation of critical regions would lead to data corruption and non-deterministic program behaviour. Multi-threaded programming systems provide mechanisms, such as *semaphores*, *conditional regions*, *mutexes* and *monitors* [82] for this purpose. Since the programmer is in total control of synchronisation, this approach is quite efficient. However, a drawback stems from the low level of abstraction. While easier to use than message passing, the explicit synchronisation still makes programming difficult if one wants to develop an efficient, predictable, scalable and robust program [101;153]. Synchronisation mistakes are common and a program's synchronisation constraints are often difficult to understand. Bugs due to race conditions are extremely difficult to trace, since threads can interleave in a large number of event combinations [75;134]. Other hazards in multi-threaded programming are deadlock, starvation and lost updates.

The shared memory model is merely conceptual and does not necessarily reflect the physical memory arrangement in the actual target hardware. **Distributed shared**

memory (DSM) systems [131] provide the illusion of shared memory on top of distributed memory systems.[8] Internally, the DSM implementation maps remote memory segments on demand onto the local machine. Necessary network data transfers are transparently initiated by the DSM system. DSM systems have been implemented both in software and in hardware. *Object-based DSM* systems (e.g., **NIPDSM** [135]) use objects as unit of allocation, instead of fixed-size memory pages or segments. This can reduce *false sharing*, where distributed processes compete for the same memory segment.

Distributed JVM implementations can be built on top of DSM. Distributing the JVM itself is one approach to run unmodified Java programs in a distributed fashion. This provides a single system image of a traditional JVM on a cluster. Examples are **Hyperion** [95] **cJVM** [15] and **Java/DSM** [189]. The latter is based on the Treadmarks DSM system and therefore relies on a special JVM that allocates Java objects onto the heap. **Titanium** [184] extends Java with features for high-performance parallel scientific computing, such as fast multidimensional array classes and an explicitly parallel SPMD model of communication. The Titanium compiler translates Titanium into C. Titanium emulates a global address space on both shared-memory and distributed-memory architectures, and is built on the Split-C/Active Messages back-end. The problem for all these systems is that they require special JVMs and therefore give up Java's portability advantage.

The **Linda** model [41;69] is another DSM variation, where shared memory appears as *tuple space;* activities can insert and remove data items from the tuple space. Internally, activities are sequential and have their own private memory. Parallelism occurs only ***between*** activities with tuple space being the only means of communication. Since activities can be programmed in different languages, Linda is often considered a *coordination language*. **BaLinda** [190] and **JavaSpaces** [154] use objects to populate the tuple space. In addition, JavaSpaces provides persistence for the tuple space. However, as the granularity of activities shrinks, the communication medium carries more and more of the computing burden [42].

The scalability of software DSM implementations to large numbers of nodes has not been proven [157]. Programs should therefore be created with data locality in mind [146]. The main performance deficit comes from guaranteeing cache consistency after updates to shared data. Some distributed memory computers therefore employ special hardware to facilitate more efficient and transparent NUMA access to remote memories. Most software systems release strict cache coherence guarantees and the cost overhead of the associated network protocols. We will return to the shared memory problem, from an OOP point of view during the discussion of passive objects in §2.2.1.

[8] A shared memory layer on top of physically distributed memory will provide non-uniform memory access (NUMA) characteristics. Programs will experience large differences in memory access times. If data on remote nodes is accessed, the necessary network transfer will cause massive latencies compared to local data access. Such an asymmetry does not exist for multi-processor systems with physical shared memory (SMP), except for the effect of processor-local caches. Some DSM implementations give up strict cache coherence for performance reasons (ncc-NUMA).

2.1.6 Concurrent Object-Oriented Programming (COOP)

Another approach to parallel execution is to hide its details behind abstract interfaces of objects. For many years researchers have attempted to integrate the concurrent and object-oriented programming paradigms and make their combined benefits available to the programmer. Work in this area aims at an integration of the flexibility of object-oriented programming (see §1.5 and [117;180]) for harnessing the increased power of parallel machines. For an overview, see [6;29;46;133;171;188]; Philippsen lists more than 100 languages in a survey of concurrent object-oriented programming [137;138]. More than half of the surveyed languages however, do not address distribution and locality at all. The reason is that they have often only been implemented as prototypical proof-of-concept studies or are targeted at shared memory systems.

In the next section we will mainly focus on the COOP subcategory of active object models. Here, a process or thread is associated with every object instance, which makes the model inherently concurrent. It also increases modularity, since it gives further autonomy to objects. Method invocations do not imply the flow of control between objects, as is the case in sequential OOP. Instead, the message passing metaphor is honoured to a much greater extent. A program is expressed as a collection of objects which act as *units of concurrency* and communicate through message passing. This often allows a very natural modelling of many real-world objects as self-contained and active entities [130], while preserving the OOP benefits of rapid prototyping, reusability and modularity. The sequential OOP model can even be seen as a technological restriction of these more general concepts of COOP.

COOP models can span categories ε1 - ε3 in Skillicorn's scale. Communication is always implicit, defined by asynchronous message passing[9] between active object instances. Synchronisation is either completely implicit (e.g., actors) or simple to express in terms of request/reply abstractions built atop of asynchronous message passing (see §2.2.5). This means that active object models belong at least to category ε3. If active objects are location-transparent and implicitly mapped onto processors at runtime, this brings a model into category ε2. A RTS may also implement implicit decomposition (ε1) by treating active objects as *potential* units of parallelism. This can be achieved by multiplexing active object instances onto threads. However, parallelism is always explicit, since it is expressed in terms of active objects. We will explore the active object design space in detail in the next section.

[9] In the context of OOP, message passing is different from the message passing approach to distributed memory programming as mentioned above. It is a communication metaphor between objects, rather than tasks and is independent of assumptions of distributed or shared memory. Message passing in sequential OOP usually implies the flow of control moving between objects. This guarantees balanced request/reply chains. In COOP message passing is independent of control flow and asynchronous: the request message contains a method identification to be executed by the destination plus a list of parameters. The request can also contain a return address for the result of the method call.

2.1.7 Distributed Object Models

Distributed object technologies, such as CORBA [176], RMI [162] see objects as large-grain and heavy-weight entities with an explicit mapping to machines; the aim of these systems is client-server communication and application integration, not HPC. Interaction is synchronous, based on RPC semantics. In contrast, we are concerned with using lightweight and location-transparent concurrent objects that are implicitly mapped (ε2) on a machine.

2.1.8 Formal Modelling of COOP

COOP has been initially defined conceptually, but not formally. There is currently much activity to provide more formal descriptions. This is a non-trivial challenge, because COOP is highly dynamic while computer theory has mainly focused on statically definable and synchronous systems. A computation model for actors has been based on transitions by Gul Agha [3] and a theory of communicating concurrent objects has been proposed by Robin Milner [119]. One of the important issues is how to express object references (or *communication labels*) and behaviours of active objects. This is important for objects to dynamically acquire and reconfigure their acquaintances and to possibly change behaviour. Some recent work in this direction is the π-calculus [120].

2.2 Exploring the Active Objects Design Space

When integrating objects with concurrency and distribution, a number of design choices exist. These are related to how issues of inter-object communication and synchronisation, intra-object concurrency and message acceptance policies, including the inheritance anomaly are addressed. Possible choices are outlined in the following sections.

2.2.1 Object Autonomy

Concurrent object-oriented languages differ in how processes are related to objects. Parallelism can either be added as a separate concept *orthogonal* to objects or it can be *integrated* within objects. These two approaches correspond the *passive* and *active* object models, respectively [46;133].

The Orthogonal Approach: Passive Objects

In sequential OOP, objects are viewed as passive entities with an operational interface. A single thread of control traverses the objects as determined by method calls. The traditional approach to introducing concurrency in this model is the addition of secondary threads. Threads communicate through accessing shared objects. Such access must be synchronised to avoid violation of the *class invariant* [117]. Furthermore, a mechanism is required for spawning new threads.

Smalltalk-80 and the extension **Concurrent Smalltalk** [185] provides processes through a fork command. Mutual exclusion and synchronisation for access to shared objects is provided through (non re-entrant) semaphores. A deadlock potential exists in the situation of (direct or indirect) self-invocations. Further, semaphores are not an implicit part of method calls on objects; i.e., it is the client's responsibility to perform correct synchronisation.

In **Emerald** [83;89] an object can be associated with an optional process that runs in parallel with invocations of object operations. This process starts its execution as soon as the object is instantiated. Objects with a process are active; those without a process are passive. Synchronisation is achieved via monitors (several can be associated with an object, e.g., to allow multiple reader-one writer semantics). Monitors in Emerald are non re-entrant, which can lead to deadlock.

DOWL [2] provides mutexes and wait queues for synchronisation between processes. Together they function very similar to monitors: processes can wait on a queue and notify it.

A more recent example for the orthogonal approach is **Java** [72]. The pre-defined class `Thread` has methods `run` and `start`. When `start` is called, a new thread is created, which executes `run`. Communication and synchronisation is achieved by allowing any method of any object to be specified as `synchronized`. Synchronised methods execute with a mutual exclusion lock that is implicitly associated with every object instance. All classes in Java derive from the Object class that has methods `wait` and `notify`, which implement a simple form of condition synchronisation. This provides functionality similar to the monitor [82] abstraction.

The orthogonal approach gives tremendous flexibility in specifying fine-grained concurrent access to objects; no maximum is imposed on the number of simultaneously active threads. However, explicit intra-object synchronisation can be the source of subtle errors as mentioned in §2.1.5. Moreover, classes that work in a sequential setting can often not be directly deployed into a concurrent environment [49;116]. This limits the reuse potential.

The Integrative Approach: Active Objects

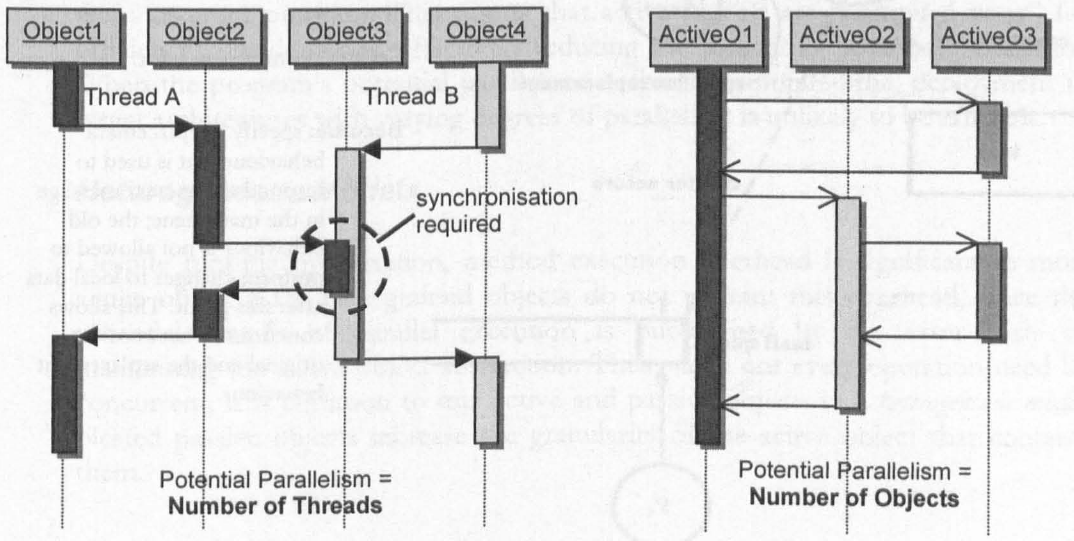
A second approach is the tight integration of objects and activities. *Active objects*, like passive objects, encapsulate state (instance variables) and behaviour (methods). In addition, they encapsulate an activity behind the well-defined interface membrane. In a sense, every active object instance encompasses virtual memory and processing resources within the object boundaries. At the hosting machine, these are mapped onto the locally available physical resources. One or several threads of control are bound to objects and cannot "leave" the object like in the orthogonal approach. Instead, method invocation is more along the lines of the object-oriented message passing paradigm[10] [180]. Objects as autonomous and active entities interact with other objects by sending messages. Messages must conform to the interface of the methods an active object exposes.

Active objects typically maintain a message queue into which incoming messages are spooled. When the active object becomes idle, messages are picked from this

[10] This should not be confused with the message passing style of parallel programming, as mentioned in §2.1.4.

queue and mapped onto method invocations. This approach separates method *execution* from *method invocation*, which increases modularity and object autonomy. An early example of a language supporting active objects is **POOL** [10]. Other examples in this category are **Orient84/K** [86] and **DRAGOON** [16], **Eiffel//**, **C++//** [38] and **ProActive PDC** [39].

Figure 2-2 Orthogonal vs. Integrated Object Model



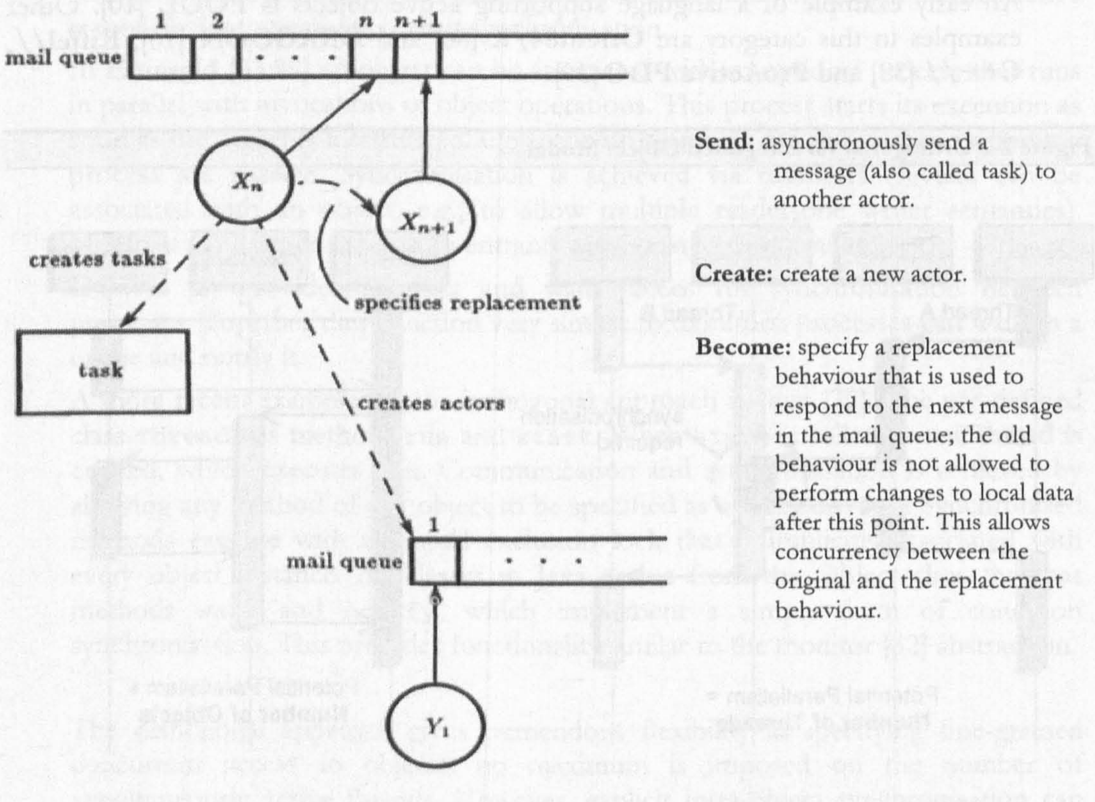
(a) Orthogonal – Passive Objects and Threads **(b) Integrated – Active Objects**

Actors

Actors [3;4;79] are a special type of active object. An actor system consists of a collection of actors, each of which has an incoming message queue or *mail queue.* Messages are delivered asynchronously and processed in FIFO at the destination actor.

An actor repeatedly executes the sequence: read the next incoming message, send messages to other actors whose *mail address* it knows and define a *replacement behaviour* (become) that governs its response to the next message (see Figure 2-3). As soon as the replacement behaviour is specified, processing of the next message can take place. Since this can take place before the processing of the previous message has been completed, actors are internally concurrent. To avoid race conditions, the actor's state can only be modified before the replacement behaviour is specified.

Figure 2-3 The Actor Model
(Gul Agha: A Model of Concurrent Computation in Distributed Systems [3], p. 26).



1
2
n
n+1
mail queue
X_n
X_{n+1}
creates tasks
specifies replacement
task
creates actors
1
mail queue
Y_1
Send: asynchronously send a message (also called task) to another actor.
Create: create a new actor.
Become: specify a replacement behaviour that is used to respond to the next message in the mail queue; the old behaviour is not allowed to perform changes to local data after this point. This allows concurrency between the original and the replacement behaviour.

Concurrent Aggregates (CA) [45] extends the Actor model with *aggregates*. An aggregate is a group of actors of the same kind that share the same name. A message sent to an aggregate is processed by only one constituent actor. This reduces unnecessary sources of serialization on the mail queue, since each aggregate may process several messages simultaneously. Delegation in CA allows the behaviour of an aggregate to be constructed incrementally from that of many other aggregates. CA is geared towards exploiting parallelism on fine-grain massively parallel computers.

2.2.2 Object Heterogeneity

A system that is purely based on active objects is called *homogeneous*. While theoretically elegant, the unification of activities and objects often leads to runtime inefficiencies. This provides the stimulus for *heterogeneous models* in which active and passive objects coexist

Homogeneous Models

Homogeneous models provide a unified object representation for a system developer. Many Actor languages, such as **ABCL/1** [186] and **Act1** [103] fall in this category. In the **POOL** language, active object are comprised from non-active, primitive data types. If such primitive types are not supported, homogeneous models become very fine-grained; even an Integer is then represented by an active object. As a result, much message passing between objects takes place at runtime. This causes much context switching and overheads for queue management and call scheduling. This creates a serious efficiency problem for naïve implementations.

Most purely homogeneous models therefore rely on compilers that reduce concurrency by clustering fine-grained actors to larger runtime entities (see 2.3.1). Heavy compiler optimisation is required to generate code with sufficient granularity from an actor program. This means that active objects are "compiled away" for efficiency considerations, effectively reducing the program's potential parallelism. When the program's potential parallelism is fixed at compile-time, deployment to target architectures with varying degrees of parallelism is unlikely to be efficient.

Heterogeneous Models

Despite runtime optimisation, method execution overhead is significant in most active object RTS. Fine-grained objects do not warrant this overhead, since the potential benefit of parallel execution is outweighed by the extra costs of maintaining the active object abstraction. Thus, since not every operation need be concurrent, it is common to mix active and passive objects in a ***heterogeneous model.*** Nested passive objects increase the granularity of the active object that contains them.

Examples in this category are **Eiffel//** [37], **C++//** [38], **ProActive PDC** [39], **Mentat** [74;75], **Act++** and **Concurrent Smalltalk**. Mentat, uses ***contained*** and ***independent*** objects, which are similar to passive and active objects. To prevent unintended sharing by several objects, **MC++** prohibits passing passive objects as arguments to messages. **Eiffel, C++//, ProActive PDC** and Karaorman's Eiffel extension [92] in contrast, adopt different message passing conventions depending on the type of the parameter: passive objects are passed ***by deep copy*** whereas active objects are passed ***by reference.***

Hybrid [129;133] defines ***domains*** as units of concurrency and distribution. Domains encompass a single process and a collection of related passive objects, so-called ***parts.*** Domains communicate through the exchange of messages following RPC semantics (the calling domain is blocked until the result arrives). Every domain has a message queue. The thread of control given by a sequence of calls is called an ***activity.*** Domains can be either active, blocked (waiting for reply) or idle. Since RPC semantics are strictly followed, every message is associated with exactly one activity. Domains can therefore keep track of the activity that blocked them and allow direct or indirect recursive self-invocations without deadlock.

Instantiation-Based Activation

One argument against heterogeneous models is that it could require an active and a passive version of essentially the same class. To circumvent this problem, some systems (e.g., **ProActive PDC, C++//**) provide the ability to create an active instance from a conventional passive class through ***instantiation-based activation.***

"Activated" objects are supplied with a message queue which serializes method invocations in FIFO order. However, this approach does not honour the fundamentally different semantics between active and passive objects. This leads to a set of drawbacks that affect **ProActive PDC** and **C++//**:

- It is not possible to override the default FIFO message acceptance policy. This reduces flexibility. As an example, consider the bounded buffer object in 2.2.4. In a concurrent setting, it would be desirable to delay requests until they can get served. This is not possible with instantiation-based activation. A method can only throw an exception if it is not serviceable in the object's current state.
- Existing passive objects typically adopt a by-reference parameter passing convention. If such an object becomes "activated", it will subsequently receive passive object parameters by value. This is inherently unsafe: The object is used with parameter passing semantics for which it was not designed. For example, state updates to a parameter could be lost.
- A related problem occurs when an "activated" object is used as drop-in replacement for a previously passive object. The activated instance has different synchronisation guarantees: for example, when a method on the activated instance returns, the client cannot assume that the call already completed. However, this is the case for the previous passive instance that was replaced by the activated instance.
- In **ProActive PDC**, it is possible to create a passive instance of an object and then "turn it active" during runtime. However, some clients might still hold on to the original (passive) reference. Therefore the same object could incorrectly be accessed to through both the passive and active interface.

Summary

Active objects have a high runtime overhead because they require a thread context switch and heap allocation. For efficiency, most active object implementations therefore rely on techniques to "merge" fine-grained active objects into larger-grain runtime entities. The possible approaches are outlined in §2.3.1.

In the heterogeneous model, the programmer must identify objects that warrant the active object overhead through sufficient method granularity. The main drawback of this approach is that active and passive objects form different type hierarchies that could contain objects with similar or identical functionality [133]. On the other hand, the model fosters reuse of existing passive (and possibly non-thread-safe) objects, since these can be tied in with containing active objects.

2.2.3 Intra-Object Concurrency

Autonomous activity of each object exhibits concurrency between them. This is called *inter-object concurrency*. Finer grain concurrency can be obtained by allowing an object to handle several messages concurrently. This is called *intra-object concurrency*. In this case, a mechanism is needed to specify the allowed interleavings of overlapping method invocations.

As mentioned in §2.2.1, passive objects do not have any associated concurrency; they rely on external threads to invoke their methods. If a passive object is accessed by several threads in mutual exclusivity, Wegner calls this a *quasi-concurrent* object [180]. In contrast, active objects have an *encapsulated* activity. If there is exactly one encapsulated thread of control in the active object, it is called *atomic* [126]. Otherwise, the active object is *concurrent*.

Atomic Active Objects

Atomic active objects process messages sequentially and in mutual exclusion. The advantage of this approach is that the encapsulation boundary of the object (its message interface) acts like a monitor (see the monitor design pattern [152]). This effectively eliminates the need for intra-object synchronisation. Nonetheless, it can still be beneficial to have a *message acceptance policy* (see §2.2.4) to alter the execution order of incoming messages.

Atomic active objects can make it easier to reason about formal proofs [116;157]. However, they reduce liveness. For example, it is not possible to implement a CREW protocol (concurrent read, exclusive write). As an aside to the liveness problem, a further disadvantage is that a blocking operation can block the object's internal thread; in this situation, the object is prevented from serving any further messages.

Yonezawa et al. [187] illustrate this problem through the example of a team of problem solver objects, working in parallel on the solution of a very hard problem. If the first finds a solution, it would be desirable to interrupt processing of the other solvers. However, due to the one-at-a-time message acceptance characteristics of atomic objects, this is not possible. In **ABCL/1** they address this issue with *express messages*. *Ordinary* activities in an atomic object can be suspended through the arrival and expedited processing of express messages. To avoid interleaving inconsistencies, an object can explicitly specify which messages it wants to receive in express mode. However, as Meyer [116] points out, it is easy for express messages to violate the "*design by contract*" responsibility [117] of an object: the expedited call might interrupt the active object, while its state is inconsistent. He proposes a mechanism whereby an ordinary method can be interrupted through controlled exceptions. These are handled by the server to "clean up" the object's state before yielding control to the expedited message.

Another advantage of atomic active objects arises in conjunction with a heterogeneous model. They can form an ideal environment for passive objects. For example, in **ProActive PDC** and **C++//** every passive object is restricted to the context of an "embedding" atomic active object; this protects passive objects from receiving concurrent requests and shields them from the concurrent environment. Consequently, there is no need for synchronisation of passive objects. **Hybrid** uses single-threaded domains and embedded parts (passive objects) in a similar fashion.

Concurrent Active Objects

Atomic active objects order incoming messages into some sequence of method invocations. In contrast, concurrent active objects can serve several messages at once: they have *intra-object concurrency*. This increases overall concurrency and liveness. However it brings with it all the challenges of concurrent software design, such as potential race conditions when multiple threads have access to the same object. As a consequence, additional coordination mechanisms are mandatory to coordinate concurrent invocations on the same object and to protect the object's internal consistency. This causes extra work and introduces extra opportunities for error; the problems are similar as they are encountered for passive object synchronisation in an orthogonal environment (see §2.2.1, passive objects). For example, **Jade** [99;146] synchronises concurrent calls on a very fine-grained statement level, whereas other approaches distinguish between readers and writers (*semantic locking*). **CEiffel** [107] `separate` objects are atomic by default, but this strict property can be relaxed through *compatibility annotations*. Each operation can specify a list of other methods that are *compatible* for concurrent execution. Conversely, **Concurrent Smalltalk** objects are concurrent by default but concurrency can be restricted through *method relations*. Method relations allow the definition of sets of methods that can only be executed in mutual exclusion. **Concurrent Smalltalk** uses monitors to control the fine-grained overlapping of methods.

Early reply [105] is another mechanism to increase an active object's internal concurrency. A reply is sent back to the client, before termination of the method call. Statements following the early reply are processed *after* returning a value; therefore, concurrency between client and server is obtained. **ACT++** integrates early reply in an actor model. Any object can become an Actor through subclassing. Otherwise, objects are passive and can only be used within an Actor instance. The drawback of early reply is that it is difficult for a client to find out about server exceptions that happen past the early reply. This defies good software engineering practice.

A further difficulty arises in the context of formal program validation. A proof of correctness must take into account all possible interleavings between concurrent method calls in a concurrent active object. This leads to a combinatorial explosion; atomic active objects are better suited for this endeavour since they reduce the state space [116].

2.2.4 Message Acceptance Policies

Method invocations cannot interleave in atomic active objects. Yet, in some situations it is beneficial to evaluate incoming messages in an order that is different from the order in which they are received. Such a mechanism allows the selective delay of certain messages according to the object's state.

One research challenge is to find specifications for the control of message scheduling that are optimal in the sense of generality and reusability. A particular problem in this regard is the inheritance anomaly [113] – the seeming incompatibility between inheritance and message acceptance policy. We examine various schemes that have been introduced in the past. These schemes differ in their expressive power and flexibility, but mainly in their impact on inheritance as discussed below.

Message Acceptance for Passive Objects

We will motivate the need for a message acceptance policy with an example case on passive objects. Consider the bounded buffer object in Code 2-1 with methods `put()` and `get()` to add and retrieve integer values. In a sequential environment, invocations of `get` and `put` are sequentially ordered. The only correct response to an invocation of `get()` in empty buffer state is to raise an exception (`BufferFullException`). In a concurrent setting the situation is different: here it is possible and more appropriate to temporarily suspend the `get` call until new items become deposited in the buffer by a concurrent activity. In a multi-threaded environment this would be achieved as shown in Code 2-2. However, since this solution relies on the temporary blocking of external accessor threads it does not function with active objects.

Code 2-1 Sequential Bounded Buffer.

```
class SequentialBoundedBuffer {
    int[] buf; int in = 0; int out = 0;
    BoundedBuffer(int size) { buf = new int[size]; }

    void put(int x) throws BufferFullException {
        if (in >= out+size) throw new BufferFullException();
        buf[in++%buf.length] = x;
    }
    int get() throws BufferEmptyException {
        if (in < out+1) throw new BufferEmptyException();
        return buf[out++%buf.length];
    }
}
```

Code 2-2 Passive Bounded Buffer with Monitor.

```
class ConcurrentBoundedBuffer {
    int[] buf; int in = 0; int out = 0;
    BoundedBuffer(int size) { buf = new int[size]; }

    void synchronized put(int x) {
        while (in >= out+size)
            try {wait();} catch (InterruptedException e) {}
        buf[in++%buf.length] = x;
        notifyAll();
    }
    int synchronized get() {
        while (in < out + 1)
            try {wait();} catch (InterruptedException e) {}
        int result = buf[out++%buf.length];
        notifyAll();
        return result;
    }
}
```

Inheritance Anomaly

Many researchers have pointed out that inheritance is in conflict with message acceptance policies[11] [113;133;180]. Depending on the message acceptance policy in use, the subclass author may need to know the implementation of methods in the superclass. The introduction of new methods in the subclass may also necessitate overriding seemingly unrelated superclass methods. Sometimes it may even be necessary to completely redefine the message acceptance policy as a result of extending that class. These effects compromise encapsulation and reusability.

The problem is that the lines of code that implement message acceptance policies (or synchronisation constraints) may be spread across all methods of a class. If a subclass has slightly different synchronisation needs, inheritance anomaly is likely to occur: then instead of inheriting methods from the parent, nearly all methods must be recoded in the subclass. However, in the re-implementations the algorithms themselves remain unchanged; just the synchronisation code lines are modified. Code duplication results in higher maintenance efforts.

The reason lies in the fact that methods introduced in subclasses are not controlled by the synchronisation constraints inherited from the parent class. This severely limits knowledge sharing through inheritance. This phenomenon is known in the literature as ***inheritance anomaly*** and has been extensively studied [9;11;113;133]. Matsuoka and Yonezawa [113] describe three instances of inheritance anomaly:

(IA-1) **Partitioning of States.** The addition of methods in a sub-class requires the acceptance sets of the parent to be changed. This problem is caused through lack of access to local state. As Matsuoka and Yonezawa [113] point out, this instance does not apply for message acceptance policies based on guards.

(IA-2) **History-only Sensitiveness of State.** This applies to methods whose execution depends on the history of events in the object's past. As a result, the parent's methods must be redefined to collect this history information.

(IA-3) **Modification of Acceptable States.** This refers to additional methods in a subclass altering the acceptable states of superclass methods. As an example, a lock mix-in class prevents message processing until an unlock message is received. Consequently, synchronisation constraints for existing methods require modification to account for the execution history of lock.

Due to these circumstances, many COOP languages, such as **POOL-T, Emerald** and **Actors**, do not provide inheritance features. Other languages, such as **ABCL/1, Concurrent Aggregates (CA)** and **Hybrid**, provide ***delegation*** as a replacement for inheritance.

[11] In the related literature, the term "synchronisation code" is often used for what we refer to as "message acceptance policy".

Accept Sets

An accept set [169] is a collection of method names. Multiple accept sets can be defined per object. At any time only a single accept set is applied. Messages belonging to the current accept set may be accepted while the other messages are delayed. Every method specifies a new accept set to use, typically as the last statement in its execution.

Languages based on *accept sets* [91], [169] define for every object a set of abstract, named behaviours that this object may assume. Corresponding to every abstract behaviour is a set of acceptable methods. At any moment during its lifetime, an object can be in exactly one of the abstract states, which determines the set of invocable methods at that moment. After processing an invocation, the abstract behaviour of the object may be updated to reflect its new invocation restrictions. For example, **ACT++** [91] specifies behaviours within C++ classes through the `behaviour` keyword.

Table 2-1 State Partition Anomaly with Accept Sets

State	**Methods**
Empty	put
Partial	put,get
Full	get

(a) original bounded buffer

State	**Methods**
Empty	put
Singular	put,get
XPartial	put,get,**get2**
Full	get,**get2**

(b) buffer with inherited get2 method

Accept sets lead to state partitioning inheritance anomaly (IA-1), as explained in Table 2-1. The left hand side (a) of this table shows a bounded buffer base class and accept sets associated with it. This class is extended to accommodate an additional method `get2` that removes two elements from the buffer (b). This creates a new relevant state, which partitions the previous accept sets. As a result, accept sets must be modified. The subclass must also override all superclass methods in order to incorporate the redefined accept sets.

Delay Queues

Hybrid [129] provides another decentralised control mechanism: Each method of an active object is associated with a *delay queue*. Synchronisation control for accessing an object is achieved by explicitly closing and opening delay queues. A message which requests the execution of a method is blocked if the delay queue associated with the method is *closed*. The message is deferred for later processing when the delay queue is *open* again. This mechanism allows simple coordination (e.g. close the delay queue for `get()` messages when the buffer becomes empty). However, delay queues exhibit inheritance anomaly for active objects: Consider a subclass that adds an additional method. Since the method was not present in the superclass its delay queue is not controlled by superclass methods. Consequently, all superclass methods that need to open or close the delay queue must be revised.

Explicit Message Acceptance

Some active object languages (usually atomic) use a dedicated thread per object, which is in charge of managing messages sent to the object. This thread is bound to a special method, called *body*. The body explicitly accepts messages and answers them through method invocations. This is a message acceptance policy that is centralised in one location rather than being distributed over the object's methods.

One example of this approach is the **POOL** family of languages[8;10;11;13]. POOL has an `answer` statement to accept messages. Karaorman [92] describes an Eiffel library with a **CONCURRENCY** class, from which active objects can inherit by providing a scheduler method, which plays the role of the body. **Eiffel//, C++//** and **ProActive PDC** define a body, called *live routine*, which is started at object instantiation time and used to monitor the message queue and dispatch incoming requests to method invocations on its stack. Several primitives are provided that allow various request serving policies to be implemented. For example, in ProActive PDC, `serveOldest(op)` can be used to accept pending invocations for a method named `op`. Since C++// and ProActive PDC can examine their message queue, declarative synchronisation mechanisms, such as path expressions, guards, etc. can be built on top of the body. Runtime performance is limited, though, because of the heavy use of reflection.

The explicit message acceptance approach exhibits all three forms of inheritance anomaly; as a result, POOL-T does not support inheritance. For Eiffel//, C++// and ProActive PDC a complete reimplementation of the live-routine in the inherited class is suggested.

Method Guards

Method guards [50] are a decentralised, per-method approach to specify message acceptance policies. A Boolean expression is attached to every method which specifies the preconditions under which corresponding messages can be accepted. If a guard evaluates to false, a message is delayed for later execution. Guard expressions are usually based on the object's internal state, but can sometimes include the value of request parameters for added flexibility. As an example why this might be useful, consider a method `getN(int n)`, which retrieves n elements from a bounded buffer.

Guards can be combined with *synchronisation counters*. Synchronisation counters were first introduced in **Guide** and **DRAGOON** [16]. Associated with every operation in an object, they record the number of started, pending, ongoing and completed operations. Therefore, they can serve as a basis to express synchronisation constraints and guard conditions. Synchronisation counters are also called *Deontic logic*. Method guards suffer from the inheritance anomaly (IA-2) and (IA-3) [59].

Circumventing the Inheritance Anomaly

DRAGOON [16] presents an innovative approach to circumventing the inheritance anomaly, based on the Ada language. Guided by the observation that inheritance is incompatible with synchronisation, Atkinson proposes the separation of synchronisation constraints from the inheritance tree. *Sequential classes* in DRAGOON do not define any synchronisation mechanism and can exploit the full power of inheritance. Abstract, generic synchronisation schemes are defined separately from the sequential classes by means of so-called *behavioural classes.* Multiple inheritance, called *behavioural inheritance,* is exploited to associate behavioural classes with sequential classes. Formal method names of the behavioural class are mapped onto actual method names at behavioural inheritance time. Thereby, synchronisation constraints as expressed in the behavioural class are imposed on the resulting *behavioured class.* The most relevant limitation of behavioural inheritance is that neither behavioural classes, nor behavioured classes can be further sub-classed. This limits reuse and modularity, which are of paramount importance in any object-oriented system.

SODA implements an approach proposed by Ferenczi [59]. This is based on the interpretation of *guards as conditional critical regions.* Ferenczi notes that this avoids the inheritance anomaly for atomic active objects in an elegant way. In Ferenczi's proposal, inherited guards represent nested conditional critical regions. Guards are then acquired successively along the inheritance hierarchy. If at any point a guard condition cannot be fulfilled, the thread of control relinquishes all critical regions in scope so that another pending message can be processed (see 3.6.1).

2.2.5 Communication Protocols

Message acceptance policies as described in the previous chapter, govern the response of a server object to incoming requests. There is also scope for various policies on the client side during interaction with a server object. This is related to the extent of integration that an object model provides. For example, remote procedure calls (RPC) are inappropriate in conjunction with active objects, since they cannot recover concurrency between client and server active objects: A client is blocked over the duration of the server's activity. To efficiently exploit the active object model, it is desirable to find call abstractions that do not block the client during the call, but still provide a way of obtaining the return value and possible exception at a later stage.

Remote Procedure Call

Remote procedure call (RPC) [25] is a standard communication abstraction for distributed systems. It is used in Java RMI [162] and CORBA [175]. In these systems a pair of *stub* and *skeleton* objects transparently marshall the remote communication using lower-level transport mechanisms. For every invocation on the stub a request message is sent via the skeleton to the server. A return value or possible exception is then propagated back to the original client via the stub. RPC calls transfer control from client to server in a strictly synchronous manner, blocking the client until the call has terminated. RPC therefore provides great safety and ease-of-use at the expense of performance and flexibility. Since pure RPC-style communication cannot recover concurrency between client and server, this technique is inappropriate for active-object based systems.

An RPC-style communication mechanism is used in **Hybrid** for communication between domains. If a method in a remote domain (remote active object) is called, the calling (single-threaded) domain is *blocked* until it receives the corresponding return message. Otherwise, a domain may be either *active*, executing a method call, or *idle*. If a domain is blocked, incoming messages are queued till they can be accepted. One exception are messages related to the blocking activity. In order to avoid deadlock through self-invocation, these can proceed in a *quasi-concurrent* fashion. This quasi-concurrency within a domain however, makes programming more difficult, since internal consistency can be easily compromised.

Multi-threaded RPC

Falkner et al [58] propose an extension to RMI which supports asynchronous calls through stub modification, thereby retaining interoperability with standard Java RMI objects. Additional threads are embedded into the stub that monitor RMI calls in progress and support a Future (see below) return mechanism. **ARMI** [144] takes a similar approach, using mailboxes to collect return values as does the light-weight Java ORB, **HORB** [81]. Each remote method invocation performed by any of the local objects is reflected through a separate thread. Insofar, concurrency inherent to the active object model is not exploited; in contrary, many additional threads are created on the client side. This scheduling of fine grained threads introduces inefficiencies, especially if objects engage in frequent communication.

Asynchronous Message Passing

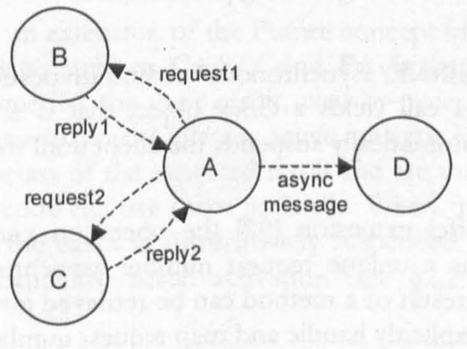
RPC-based communication mechanisms are not well suited for active object interaction, since they cannot recover inter-object concurrency. An appropriate interaction technique must be asynchronous, in order to allow the client to proceed independently with some other useful computation *while* the call is in progress. As Beust [22] notes, "*[...] asynchronism is critical in a distributed world. If you want efficiency and scalability, you need asynchronism.*"

To this end, the Actor model [3] introduced asynchronous message passing as sole communication mechanism between objects. All inter-object communication is based on the one-way sending of request messages, which effectively decouples the client from the method execution on the server side. The request initiator can proceed independently with other useful computation *while* the call is in progress. While this approach provides flexibility and efficiency it comes at the expense of safety and ease-of-use. In order to send back a result of a message, a programmer needs to explicitly specify the reply destination and save the necessary data to handle the reply. For a programmer it becomes difficult to retrieve the result of an invocation or to synchronise on its termination. This does not encourage modularity and structured programming and can be compared to the `goto` statement in sequential programs [166].

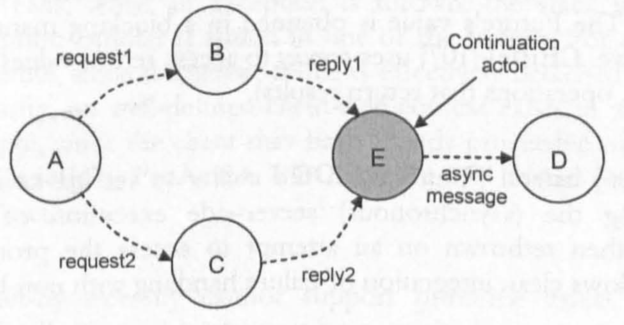
Join continuations [5] associate a *continuation actor* (see Figure 2-4) with a method invocation and pass its mail address as parameter. Operations dependent on the result of these calls are delegated to the continuation actor, which is then executing these upon reception of all required results.

Figure 2-4 Continuation Actor

The Actor A sends a request to Actors B and C and, once the replies are available, sends a message to Actor D. Without a continuation actor this requires explicit matching of the replies by A (a). This is done independently by the continuation actor E, which knows how many replies to expect (b).



(a) before: asynchronous request/reply



(b) after: a continuation actor manages the reply

NexusJava [63] provides asynchronous communication for Java. The communication scheme is based on global pointers which allow addressing objects in remote memories. However, the programming interface is very low-level and verbose and distributed interactions are limited to the invocations of methods explicitly registered as handlers.

Futures

Futures were originally introduced in **MultiLisp** [77] to identify potentially concurrent computations: The expression `(future X)` is a hint to the RTS that expression X can be computed safely in parallel. At runtime, additional threads may be created eagerly or through *lazy task creation* [121].

Since concurrency already exists between active objects, we are interested in a more recent interpretation of Futures. In this interpretation Futures are a mechanism to implement asynchronous but structured method calls. They act as placeholders for partially computed values and decouple method execution from method invocation. This allows a client to proceed past the method call and perform other computations concurrently. At a later stage, and only when required, the Future's value can be picked up. Compared to one-way message-passing, the Future mechanism allows the retention of a balanced request-reply structure, which is important from a software engineering point of view.

In **Concurrent Smalltalk**, asynchronous method invocation is performed through the `&` suffix. Such a call yields a *CBox* object that is a proxy to a Future. The `receive` method automatically suspends the client until the Future has a value.

In Karaorman's Eiffel extension [92] the operation `remote_invoke` invokes a method and returns a unique request number asynchronously. Via the method `claim_result` the result of a method can be retrieved given its request number. A programmer must explicitly handle and map request numbers.

ABCL/1 [186;187] is an extension of the Actor model that associates every asynchronous message with a Future object that is explicitly created to save its reply value[12]. The Future's value is obtained in a blocking manner via the `next-value` primitive. **CEiffel** [107] uses *proxies* to access return values for asynchronous functions (i.e., operations that return results).

Promises [106] extend Futures with the ability to record exceptions which are thrown during the (asynchronous) server-side execution of a request. This exception is then rethrown on an attempt to access the promise's value. This mechanism allows clean integration of failure handling with non-blocking RPC.

Responsibilities [51] are proxies which can be used as first-class objects, i.e., they have global identity and may be passed around by reference. Clients create responsibilities and pass them as by-reference arguments in method calls to the server. The server explicitly provides a value for a responsibility via the `supply(val)` primitive. Therefore, several responsibilities can be attached to a single method call. While this approach increases flexibility, it also makes implementation on a distributed memory architecture more difficult. If not-yet-available responsibilities are exported to remote nodes, how can they be informed about availability of results? In addition, no context for exception handling is defined.

The message-answering semantics of active objects is distinctly different from the message-answering semantics of passive objects with respect to self-invocation.

[12] ABCL/1 actually provides three different call semantics: *now* type supports RPC semantics; *past* type supports one-way message passing; *future* type returns a Future variable.

Namely, to answer a message, an active object must interrupt its own activity. Yet, if an atomic active object sends a message to itself and blocks on the returned Future, we have a situation of *deadlock*. Direct self-invocation, of course, can easily be detected, but indirect self-invocations require an analysis of the complete method invocation graph, which is generally expensive (although this approach is taken in databases). Languages with atomic objects and blocking Futures therefore often require the absence of direct or indirect self-invocations [39].

Wait-by-Necessity

Wait-by-necessity is an extension of the Future concept introduced by the **Eiffel//** language [37] and later used in **C++//** and **ProActive PDC**. Whereas Futures must be explicitly queried for their result, wait-by-necessity makes asynchronous calls and Futures transparent. Calls on active objects return Futures, which are implemented as subclass of the expected result and are therefore transparent to the client. Even legacy code can use them normally. When the Future is used without yet being available, the client is automatically suspended. Wait-by-necessity allows the transparent instantiation-based activation (see §2.2.2) of previously passive objects.

This mechanism is very elegant and promotes reuse, but it does have a set of drawbacks. Firstly, wait-by-necessity is incompatible with exceptions. In synchronous calls, when an exception is thrown, the stack is unwound until a suitable exception handler is found in one of the frames. For asynchronous calls, this is impossible, since the server frame is effectively detached from the client. In wait-by-necessity, no well-defined client-side context exists in which the exception could be caught, since the client may have already proceeded past the method call. Therefore, **CJava** [47], **ProActive PDC** and **C++//** forbid exceptions thrown by active object methods.

Secondly, wait-by-necessity cannot support primitive values which cannot be subclassed. Finally, the result of a wait-by-necessity invocation is in a different level of the inheritance tree than the client might anticipate. This might break legacy code by throwing unexpected casting exceptions.

Summary

In the presence of active objects, asynchronous calls are required in order to obtain concurrency. Completely asynchronous calls defy good software engineering practice, since there is no guarantee that return values or error conditions are ever checked. This opens Pandora's Box and leads to an error-prone programming style. Wait-by-necessity is elegant, but has the disadvantage of not being able to deal with exceptions. With Futures, on the other hand, asynchronous calls are not transparent, but they are safer than wait-by-necessity. One problem in the combination of Futures with atomic active objects is that a blocking Future temporarily stops the client active object from accepting further invocations. As a result, deadlock can occur, for example through cyclic invocations.

The previous paragraphs were concerned with various design choices for an abstract programming model based on active objects. We now go on to consider the various routes that have been taken in the past to implement such a model.

2.3 Runtime Issues for Active Objects

We finish this chapter with an examination of the implementation issues involved in active object systems. This section is mainly focussed on Java-based systems, because that is the language we use for system implementation. Message-passing systems, such as PVM or MPI require the programmer to explicitly specify the placement and scheduling of computations and the communication between them. They do not provide facilities for dynamic creation of tasks, in adaptation to available hardware parallelism, and therefore restrict the extent of automatic optimisations. In contrast, active object programs involve irregular data-driven computations, dynamic creation of tasks and asynchronous communication. A RTS has considerably flexibility in choosing strategies for *granularity control* and *placement* (mapping of objects onto processors). Thus, opportunities and challenges for automatic optimisations are greatly increased.

Active object algorithms are often expressed at a finer level of granularity than distributed memory machines can exploit efficiently. Granularity control is the process by which the granularity of such algorithms is increased. We explore this technique in the following section before turning to issues of object placement.

2.3.1 Granularity Control

A fine-grained active object model holds great potential for high-performance computing, since it inherently exposes large amounts of concurrency. Target architectures will typically offer a variable degree of physical parallelism. One challenge is the reduction of concurrency exposed at runtime to a degree that efficiently matches the target architecture's physical parallelism.

Typically an active object runtime system will expose a set of threads on every participating machine. The total number of threads is called a program's *potential parallelism*. For systems that are based on *explicit decomposition*, active objects are directly represented by threads. This prevents the advantages of fine-grained active object models from being realised, since granularity control is a design-time decision. In heterogeneous models, for example, the granularity of an active object can be increased by embedding more passive objects. Example systems in the category of explicit decomposition are **ProActive PDC**, **C++//** and **ActorFoundry** [132].

More advanced implementations remove decomposition decisions from the programmer's domain. For example, it's common to multiplex several active objects onto a small number of threads in order to achieve *implicit decomposition*. We can classify three different approaches that have been explored in the past:

Compile-time Approach

Homogeneous models often employ a compiler-based approach to "compile away" active objects [5;56;96]. The aim is to convert "superficial" active object instances into sequential structures. Compiler-based techniques lead to efficient code but fix the potential concurrency in a program. It is possible to parameterise the compiler with details of the target architecture. Any such parameters, however, require experimentation to calibrate and cannot accommodate for unpredictable runtime conditions, such as different data sets or varying resource availability. Furthermore, this process may need to be repeated for a different target architecture.

Profiling-Feedback Approach

Charm++ [98] is a runtime system based on C++ and targeted at dynamic and irregular applications in the parallel object-oriented domain. It uses a post-mortem analysis tool that allows runtime optimisations without programmer intervention. Based on traces and analysis of previous execution, the code is interspersed with calls to the Charm++ runtime library to improve parallel execution efficiency. Besides granularity control, the results are also used to control object placement and to optimise communication efficiency.

The **Finesse** tool [125] uses a similar approach, based on profiling information collected through example runs of the program. Finesse also relies on programmer feedback to identify and resolve potential bottlenecks to efficient parallel execution.

The drawbacks are similar as for the compiler-based approach: profiling cannot accommodate for highly dynamic and data-dependent problems and repetition of the profiling process may be required for different target architectures.

Runtime approach

In the most flexible granularity control mechanism, concurrency is dynamically adapted to physical parallelism at runtime. A common method is the *inlining* of calls, when target objects reside locally. Instead of performing a heap-based invocation or using the loopback network interface, calls are converted into stack-based invocations where appropriate. This reduces the overhead for location-transparent objects. For example, **JavaParty [140]** exploits *unexpected locality* by accessing local objects at the cost of a pointer indirection instead of performing expensive network loopback communication. Target objects in JavaParty are passive and multi-threaded. In **ABCL1**/AP1000 by Taura [167] calls are normally heap-based to support the mail queue semantics of actors. However, messages sent to idle actors on the same processor are changed into stack-based, *inlined* function calls for performance.

With actors (or atomic active objects) the inlining approach could lead to deadlock as a result of *parent-child welding* [54]. Client and server (or parent and child task) are effectively welded together through inlining into a single task. If an inlined call becomes blocked waiting for a Future to resolve, the client is blocked as well and is not available for executing other requests. To avoid the resulting deadlock potential, **ABLC1/AP1000** copies the stack frame of a blocking method execution into a heap-allocated frame and saves it in a global scheduling queue for later execution. The stack is unwound and the execution of the previous method continues as if the message had been sent asynchronously.

Lazy task creation [121] is an alternative to inlining. It goes the other way by retroactively exposing more concurrency when processing resources become available. Calls are inlined by default but can be converted into heap-allocated invocations in order to allow *work stealing* through other processors. This mechanism is used in the **NIP** RTS [177;178].

2.3.2 Object Mapping

Distributed objects must somehow be allocated onto the nodes of a distributed memory system. Efficient object mapping revolves around two conflicting issues. On the one hand, frequently interacting objects should be collocated on a node in order to reduce inter-node communication overheads, i.e., *locality* should be exploited. On the other hand, *load balancing* is required so that objects are balanced over all nodes in the cluster to optimise processor utilisation. To strike a balance between these two requirements represents an important challenge to efficient exploitation of active object programs.

The object placement is either explicitly controlled by the programmer (*explicit mapping*–ε3) or handled transparently by the runtime system and/or compiler (*implicit mapping*–ε2). Further, an allocation scheme can be static or dynamic. In a static scheme, an object remains at the node it was instantiated at. In a dynamic scheme, migration of objects can take place at runtime.

Static Explicit Mapping

Distributed object systems built around **Java RMI** or **Corba**, are often based on an explicit mapping. Objects are created at a fixed location in the network and looked up via a central name server. In **ProActive PDC**, a configuration file maps virtual host names onto real host names on the target architecture. This is inflexible, since it requires programmer interaction for every new target architecture and possibly several cycles of experimentation. Another problem is that a static mapping cannot accommodate for objects that significantly change their activity patterns during runtime. Objects should then be dynamically reallocated to avoid load imbalance and/or the physical separation of tightly coupled objects, which engage in heavy communication.

Static Implicit Mapping at Instantiation-Time

Some systems implicitly take allocation decisions at object instantiation time. In DOOM [13] objects are placed according to the system's load situation and their locality at creation. After the initial assignment, no dynamic migration is possible. Allocation decisions may also be taken at compile-time. However, this carries problems similar to compile-time granularity control. A compiler may have difficulties in predicting runtime overheads and communication patterns, especially for reactive programs that are strongly dependent on runtime data. Neither can compilers deal with unpredictable resource availability, for example in non-dedicated clusters. One solution to these problems is the dynamic migration of objects according to runtime conditions.

Dynamic Object Migration

Optimum allocation strategies are often not known *a priori*; rather they are runtime dependent and can therefore only be established during execution. Especially for applications with highly dynamic behaviour any static mapping is likely to perform poorly [173], p. 228. Object migration is essential for the scalable execution of dynamic, irregular applications over sparse data structures [96]. However, few Active Objects systems offer support for dynamic object migration and if so, rely on the programmer to initiate migrations explicitly [89;96].

The positive effects of dynamic object migration have been studied in the **Emerald** project [83;84;89]. In Emerald, objects can migrate between nodes. Objects can be declared *immutable*, which simplifies sharing. There is both inter-object and intra-object concurrency. Objects can be declared as *monitors*, which simplify handling intra-object concurrency. Objects with a process (executing in parallel with the monitor) are active; those without a process are passive. All method calls are synchronous; new threads of control arise by creating a new active object.

Objects are location-transparent in **JavaParty** [140;191]. At instantiation time they are placed onto some machine chosen according to a strategy that can be dynamically modified at runtime. The RTS deals with locality and communication optimisation automatically triggers object migration. Migration decisions are based on communication patterns collected at runtime or on hints given by the programmer. Migration is implemented by locking the object and using Java serialization to move it to a new address space. A *forward-pointer* is left behind that informs any subsequent callers about the new location of the target object so that references can be updated. This is problematic, as it could degenerate to n-1 call attempts for updating a single outdated reference where n is the number of hosts in the cluster. If migration occurs frequently or if many references exist to the object, this would be very inefficient.

An alternative to forward-pointers is used in the **NIP DSM** system. Every object has a *master node*. An object reference contains the current (or last known) location of the target object as well as information about the target object's master node. If the current location is stale, it can be updated from the master node. Since every participating machine in a cluster can act as master node, this is a decentralised name server approach, which does not create a centralised bottleneck.

Function Shipping vs. Data Shipping

In DSM, objects are allocated in a software-emulated global address space that extends over physically disjoint memories. When client and server objects that engage in a method call are not physically collocated, the server object is transparently moved to client-local memory. This usually involves acquiring a lock on the server object and copying its state into the client-local memory before executing the desired method, a technique called *data shipping*.[13]

In contrast, most distributed object models are based on the idea of *function shipping*: when remote objects engage in a method call, target method name and parameter values are encoded in a request message. This request is sent to the server object, processed remotely and a reply is sent back.

Active object implementations are generally based on *function shipping*.[14] If objects are *location transparent*, a runtime system can use a mixture between function shipping and data shipping to operate between the goals of improved locality and load balancing. Due to this hybrid paradigm, dynamic object migration brings flexibility advantages. However, it also relies on runtime analysis of object interaction patterns and processing requirements. This incurs a profiling overhead for recording computational and communicational patterns of objects.

2.3.3 Garbage Collection

Distributed garbage collection is an important topic for active object runtime systems. However, we do not want to examine this in detail or provide a solution for our runtime system, since a large amount of literature and working systems already exist [100;141]. Each of the previous solutions has strengths and weaknesses. Reference counting cannot reclaim garbage cycles [141]. Mark-and-sweep is difficult to implement on distributed memory. RMI uses reference-counting based on leases. It keeps track of all remote JVMs that have references to the locally kept object. Active objects add another dimension of difficulty to this problem [90].

2.4 Summary and Conclusion

While message passing and shared memory are the two predominant parallel programming models in use today, they make parallel programming rather tedious and error-prone, because of their low abstraction level. The active object model is an alternative born from the marriage of object-orientation and concurrency. Active objects can expose a high degree of concurrency and allow the natural modelling of many real-world problems. Parallelism is expressed at a much higher level and is highly implicit. Communication and synchronisation are expressed naturally through method calls. As units of distribution, active objects can provide implicit mapping and implicit decomposition. However, many current implementations suffer from the large runtime overhead of active objects, which forces a programmer to explicitly adjust active object granularity to each particular target platform. Another obstacle is the inheritance anomaly, caused by the seeming

[13] In most cases, DSM system will create or update cached copies of the target object for improved locality.

[14] Unless some DSM system is used that provides a global address space for all active objects.

incompatibility of inheritance and message acceptance control. While Ferenczi's proposal circumvents the inheritance anomaly, the implementation of the underlying conditional critical regions has previously been considered runtime-expensive. A further issue is the problem of providing balanced request/reply chains in the presence of asynchronous message passing between objects.

The subsequent chapter presents the design and implementation of the SODA abstract machine and highlights how runtime mechanisms based on the programming model properties, achieve implicit decomposition and mapping.

Table 2-2 Overview of Active Object Languages

Language	Approach/Model of Concurrency	Object Heterogeneity	Intra-Object Concurrency	Intra-Object Coordination	Client-Server Interaction Protocol
Concurrent Smalltalk	Objects and processes	Only passive objects	yes	semaphores	Synchronous and asynchronous method calls
POOL	Active object with autonomous body	homogeneous	no	Live method, using accept statement	Synchronous (early return, rendezvous-like); one-way in POOL2
Pure Actors	Active object (Actor)	homogeneous	Concurrent, reactive	Become primitive	One-way messages
ABCL/1	FIFO atomic actors, can contain primitive values	Homogeneous (only primitive members allowed)	Atomic, reactive	Live method using **select** statement, delegation	3 types: now (RPC), past (asynchronous), Future (Future)
ACT++ (based on C++)	Active objects derived from Actor class	heterogeneous	concurrent	thread creation using **become**	Asynchronous one-way request; CBox objects to collect reply (Future semantics)
Act 1 [103]	Actors	homogeneous	Futures	serializers	continuations

Table 2-3 Overview of Active Object Languages (continued)

Language	Approach/Model of Concurrency	Object Heterogeneity	Intra-Object Concurrency	Intra-Object Coordination	Client-Server Interaction Protocol
Eiffel//, C++//, ProActive PDC	Active object with autonomous body	heterogeneous	Atomic, autonomous	Explicit acceptance	Wait-by-necessity, exceptions unsupported
Hybrid	Concurrent domains as collections of objects are	Domains and contained objects	Concurrent	Delay queues and delegation	Synchronous RPC, no exception handling
Emerald	Optional processes attached to objects.	Active (opt process) and passive objects	processes	monitor	Synchronous RPC
Dragoon	Passive objects and processes	no	yes	Behavioural classes, Deontic logic	Synchronous RPC
CORRELATE [93]	active objects	Active and passive objects	Atomic, "interface" methods in exclusion with an "autonomous" behaviour (live routine)	Guards on operations	Synchronous and asynchronous (one-way) calls

Chapter 3

The SODA Model and Language

SODA has been designed to overcome limitations encountered in previous active object systems (see Chapter 2). The SODA model introduces three extensions to active objects that address the problems of deadlock, inheritance anomaly and restricted object-internal concurrency: These are 1) a novel mechanism for inter-object synchronisation, called *Futures and Funnels*; 2) implicit intra-object synchronisation based on *Ferenczi guards* and 3) *detached methods to* control intra-object concurrency. We highlight how Futures and Funnels are relevant to achieve implicit communication and synchronisation. We also demonstrate the model's usability by giving example solutions to a set of real programming problems. SODA is implemented as an extension of the Java language. This language is supported by a runtime-library that presents a virtual machine view of a multi-computer system (see Chapter 4). SODA's abstraction level occupies a middle ground between simple actor models [3;5] and high-level agent-based systems [97].

3.1 Motivation and Overview

In the previous chapter we gave an overview over the current state of research in the area of concurrent object-oriented programming. The motivation behind most languages and models in this family is the straightforward and efficient expression of parallel and distributed programs. The category of active object systems is of particular interest for distributed memory architectures: Objects as the unit of concurrency and distribution can be transparently allocated onto physically disjoint address spaces. Each active object strongly encapsulates its variables and confines external access to its method interface. For the runtime-system implementer this has the advantage that no virtual shared memory abstraction is required. SODA uses function shipping instead of data shipping: the state of foreign active objects can only be retrieved through method invocations. Such invocations are transparently mapped onto message sending across the network if client and server active object happen to be located in different physical address spaces at runtime. This circumvents problems of scalability and cache consistency which are often encountered by virtual shared memory runtime-systems [43;131].

Despite the potential of active object systems, Chapter 2 identified a set of trade-offs involved in their design. Tensions exist along various dimensions of the design space: for example, the degree of intra-object concurrency is associated with a trade-off between efficiency and ease-of-use. Another example is synchronous vs. asynchronous call semantics, which represents a trade-off between safety and potential concurrency.

To overcome some of these tradeoffs, we propose three extensions to the active object model. These address the problems of deadlock, inheritance anomaly and restricted object-internal concurrency in the context of atomic active objects:

Futures and Funnels: The SODA model offers a novel dataflow-based return mechanism, called *Futures and Funnels*. This mechanism bridges the gap between synchronous and asynchronous method invocations. Funnels allow asynchronous method invocations while guaranteeing balanced request-reply chains. Funnels also avoid deadlock when used with atomic active objects. Parallelism is the default execution mode in the SODA model; sequencing occurs only for successive method invocations on an active object and in the case where Funnels describe implicit data dependencies between active objects.

Ferenczi Guards: An active object's servicing policy for incoming messages is FIFO by default. SODA implements *Ferenczi guards* [59] to conditionally override this behaviour. Ferenczi's proposal circumvents the inheritance anomaly and, to our knowledge, has not previously been integrated with active objects.

Detached Methods: *Detached methods* in SODA are an integration of the *half-async/half-sync pattern* [152] with active objects. This concept combines the efficiency benefits and expressive power of multi-threaded active objects with the convenience and encapsulation advantages of atomic objects.

We demonstrate that these extensions improve the usability of active object systems and that an efficient implementation is possible. SODA programmers concern themselves with an object-oriented design for the problem at hand. On a per-class basis, the activity semantics have to be decided (see §3.2). Low-level tasks, however, are handled transparently at runtime. For example, the assignment of object instances onto processors is dynamically adjusted by the runtime-system in order to balance load and network utilisation. This allows the execution of SODA programs on a variety of architectures without change. SODA programs are best suited to distributed memory architectures. Multiprocessors or single-processor machines are alternative target architectures. However, SODA programs will not be able to compete with the performance of other programming models that are geared towards these architectures.

SODA adopts an active object concurrency model. As such it inherits the well-known benefits of object-orientation, like rapid prototyping, reusability, modularity and maintainability [158]. Concurrency is implicitly created by asynchronous method invocation. Method invocations are translated into requests. These are queued at the target active object until they can be processed. Unprocessed requests are the driving force behind all computation in the system. The computation is initiated by the runtime system that sends a request to the main method of the primordial active object[15]. Further active objects and requests can be

[15] The main method is in fact a detached method, as outlined in §3.3.2.

created dynamically. In this section, based on the taxonomy described in §2.2, we outline the design choices underlying SODA.

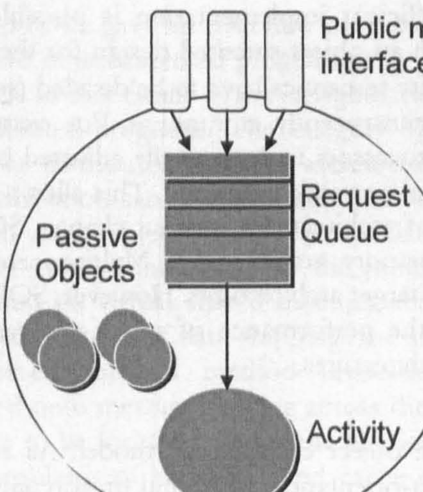
3.2 Object Heterogeneity

The SODA model is heterogeneous: active and passive objects coexist. Active objects reside in a global address space. This address space can transparently extend over several physically distributed address spaces. The identity of passive objects is only valid within the scope of a single, owning active object. The following definitions are similar to those that appear in the definitions of such languages as Java [72] and C++ [26].

3.2.1 SODA Active Objects

Definition 3-1 (SODA Active Object). An active object in SODA is a *state and activity container* with globally valid identity. The state is hidden from the outside world, or encapsulated. An active object has external methods (or operations) which provide the only means of accessing the state. All invocations of one of the external methods are handled asynchronously by the active object's encapsulated activity. The internal state of an object may consist of references to other active objects, passive objects and primitive data types, or some combination of these.

Figure 3-1 SODA Active Object Anatomy



Note that Definition 3-1 excludes the possibility of accessing the fields (or state) of an object directly. All external access is via method calls. This approach provides a strong degree of encapsulation and avoids internal state inconsistencies through concurrent updates. This is not a restriction, since simple getter/setter methods for every field can be provided. In fact, the consequent usage of the JavaBeans pattern is a good software engineering principle.

By the same token, SODA does not support class variables. Class variables are shared amongst all objects of the same class. This language feature makes centralised assumptions about the environment that conflict with the implicit distribution model. Maintaining a consistent state of such variables would be a very costly operation in a distributed system and we therefore decided not to directly support this in the programming model.

Definition 3-2 (SODA Active Object Method). An *active object method* is a function that is externally exposed to access an active object and perform computations based on the object's state. Each method takes zero or more *arguments*, and possibly returns a value. Invocations are asynchronous: a client is not blocked waiting on their completion. Instead, method invocations in SODA implicitly create concurrency by committing the internal activity of the target active object to their asynchronous processing.

Note that constructors are treated as active object methods in SODA (see below). Object instantiation is therefore an asynchronous operation. This has the advantage that a client may instantiate several active objects concurrently. Each of the constructors can potentially run in parallel.

Active objects support access modifiers for active object methods as follows:

- **private**: internal calls only (self-invocation).
- **protected**: access only from within the active object and its package.
- **public**: default visibility, allows global access.

Active Object Declaration and Instantiation

Figure 3-2 shows the interface of an active class in SODA. Active classes are marked with the **`active`** keyword, which by default allows public access. The signature of all active object methods is based on a **`Future`** return value. This enables the asynchronous calling of these methods as described in detail below. Methods with the special name **`init`** declare the constructors for the active object class.

Figure 3-2 Interface of an Active Class.

```
active class Buffer1 {
  public Future init(String name);                    // constructor
  public Future put(int element);
  public Future get();
  public Future getName();
}
```

Figure 3-3 demonstrates the instantiation of an active object instance of class `Buffer`. Note that the constructor invocation acts like a normal active object method, i.e., it returns immediately with a `Future` as a placeholder for the actual active object reference. Via the `get()` operation the actual value can be obtained from the `Future`. This is discussed in detail in §3.4.1.

Figure 3-3 Active Object Instantiation

```
Future f = Buffer1.init("myBuffer"));
try
{
   Buffer1 buf1 = (Buffer1) f.get();
}
catch (Exception e)
{
  // when the instantiation failed
}
```

3.2.2 SODA Passive Objects

Definition 3-3 (SODA Passive Object). A passive object in SODA does not have a global identity, but is always private to the scope of a single active object. Within the context of its owning active object, a passive object can be used with the conventional Java syntax and semantics. i.e. calls are synchronous. Passive objects cannot be referenced from foreign active objects. Instead, access must always be mediated through the owning active object. Together with its private passive objects, an active object forms a single concurrency unit.

Definition 3-3 implies that passive objects cannot be passed by reference between active objects. If passive objects are used as parameters for method calls they are therefore passed using deep-copy semantics. This involves recursive copying of the complete *object graph*, i.e., the state of the passive object and all referenced objects. Deep copying is necessary, regardless of whether the client and server share a physical address space at runtime. SODA offers a set of optimisations that relax the strict deep-copying of passive parameters if these fulfil certain criteria (see §3.7).

3.2.3 Programming Methodology

Heterogeneous models have previously been chosen for performance considerations. In some systems active objects can cause significant overheads due to synchronisation and thread management costs. Only objects with a certain granularity threshold warrant this overhead. Passive objects are a means to reduce the number of active objects and to increase their granularity. However, the programmer carries the burden of assessing object granularity and identifying suitable active objects in a design. This is difficult, because the granularity threshold is influenced by hardware characteristics (such as processor performance and network latency) and may also depend on the value of runtime parameters.

SODA offers a different programming methodology: active objects in SODA have a comparatively small runtime overhead[16] (see Chapter 4). Consequently, programmers can create large numbers of active objects without having to worry about the performance impact.[17] The SODA runtime-system adjusts the grain size dynamically in response to runtime conditions. This has the advantage that on highly parallel platforms more concurrency can be exposed, while still allowing efficient execution when physical parallelism is restricted.

The choice to adopt a heterogeneous model in SODA was therefore motivated not so much by performance considerations, but rather by flexibility and convenience factors. In SODA the following are suitable use cases for passive objects:

Legacy classes. Firstly, passive objects allow the instantiation of legacy classes. Since an active object can shield a contained passive object from concurrent invocations, this allows the reuse of a legacy class, even if such a class was not intended for usage in a concurrent setting.

Operating system classes. Secondly, operating system resources, such as sockets, are supported as passive objects. This has consequences for the mobility of the owning active object (see §3.2.4).

Data Containers. Thirdly, passive objects are effective in modelling structured data containers. They should not have complicated compute-logic. This can be useful to reduce the frequency of (remote) inter-object communication: As a method call argument, passive objects are passed by value and therefore effectively cached at the server object. This is more efficient than querying a foreign active object repeatedly for its state via methods that return primitive data types. Where active objects are not collocated, this can lead to drastic performance improvements. Data containers provide some form of data shipping in the otherwise function-shipping oriented SODA model.

The following are use cases for active objects:

"Computational" Objects. All objects that have one or more methods that are more significant than just simple setter/getter methods. Although the overhead for active objects is relatively small in SODA, methods must have a certain minimum granularity to amortise this (see §5.3).

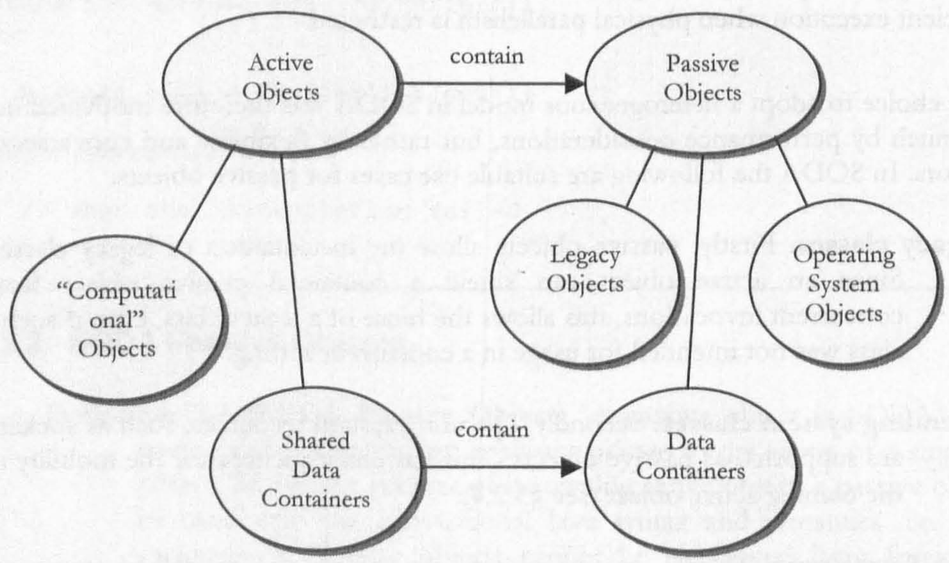
Shared Data Containers. Since SODA lacks a distributed shared memory abstraction, copies of a passive object data container will run out of the synchronisation. Sometimes it is desirable to have a data container which

[16] SODA chooses at runtime between a set of techniques to implement active object method invocations. In the most optimised case, active object overhead in SODA is only marginally higher compared to conventional passive object calls. This is the case, if the server object (1) resides on the same node of the distributed architecture, (2) is idle and (3) the method guard evaluates to true.

[17] However, there is also the issue of memory consumption for the active object state infrastructure management. This cannot be avoided by runtime-based systems.

gives identical views to a set of active objects. If such "shared object" semantics are required, they can only be modelled through an active object wrapper.

Figure 3-4 SODA Programming Methodology



3.2.4 Dual Semantics of Active and Passive Objects

No Active-Passive Polymorphism

Instantiation-based activation is used in some object models in order to avoid a programmer having to provide multiple versions of a class according to use. However, this concept is inherently error-prone as it does not fully honour the dual semantics of active and passive objects as shown in §2.2.2. To avoid these problems, SODA active and passive objects form strictly separate class hierarchies and do not support polymorphism. Since in SODA the use cases for each object type are so different, the need for creating multiple versions of a class should not normally arise.

Synchronised Passive Objects

One problem with the integration of legacy objects deserves mention in this context: while unsynchronised passive objects can be embedded without restrictions into active objects, care has to be taken for passive objects that define object-internal synchronisation constraints. If a *synchronised* object was embedded as private passive objects into an active object, any blocking in this passive object would deadlock the owning active object. For example, the embedded object could be a single-space buffer, which needs to be accessed in alternation by a producer and consumer. The passive object's internal blocking condition can only be removed through another thread, but the single thread of the owning active object is already tied to the waiting condition in the embedded passive object. The solution is to use detached methods (see §3.3.2) and allow them access to the synchronised passive object.

Blocking synchronised objects are infrequent, as an examination of the Java 1.3.1 standard class API revealed. Most synchronized objects only enforce mutual exclusion between their operations according to the monitor pattern (i.e. enforce mutual exclusion of all method invocations, but contain no `wait` statements, in which case this problem does not arise [152]).

Operating System Classes and Mobility

Some passive objects, which encapsulate local operating system resources, cannot be moved to another host without violating their internal consistency. For example, a `Socket` object is meaningless if its reference into the operating system no longer exists. The same is true for objects that interoperate with local files or databases. To account for this situation, active objects can be marked as fixed to be exempt from the automatic migration mechanism. Such objects then act as services, e.g. a file service or a database service (comparable to services in the Grid [62;71]). This mechanism should also be used to mark large-volume shared data containers for which migration would be very expensive.

Alternatively, active objects can implement the interface `MigrateControl`, which defines two methods `prepareMigrate` and `migrateDone`; these can be used to perform low-level handling of passive objects.

Per-Host Active Object Instances

By default, active objects are location-transparent: a programmer has no influence over where an instance is created. Some active objects, however, have special requirements on their execution environment. As an example, consider an active object, which provides database access or interoperates with a non-shared file-system. SODA allows allocating active objects onto hosts, which fulfil certain criteria. It is also possible to instantiate an active class on every host participating in the cluster. This is achieved through obtaining host meta-objects, which can then be used as a parameter for instantiating new active objects on an explicit location. With the exception of per-host active object instance, distribution is completely implicit in the logical model of an application.

3.3 Intra-Object Concurrency

SODA uses an atomic active object model. This means that only a single internal thread handles all method invocations and state changes. We explain the motivation for this design decision. We then turn to the concept of detached methods, which allows active objects to encapsulate additional threads in a manner that does not remove the benefits of atomic active objects.

3.3.1 Atomic Active Objects

SODA active objects are atomic. They only have a single internal thread that is responsible for serving incoming requests. Every method invocation on the active object is translated into a request. Requests are queued until the internal thread becomes idle for their processing. Incoming messages are processed in FIFO order by default. A different servicing policy can be specified through Ferenczi guards (see §3.6.1). Once activated, methods run in mutual exclusion, without interruption or blocking. This absence of object-internal locking mechanisms makes programming easier and avoids infinite delays *during* execution of a method. With individual methods having non-blocking semantics, the active object can never block subsequent method invocations. As long as new requests arrive and they fulfil the corresponding method guards, the active object guarantees to process them. In detail, atomic active objects offer the following benefits:

Better Design. From a design point of view, atomic active objects have the advantage of complete encapsulation of state, behaviour, and single activity on a per-object basis.

Mutual Exclusion of Methods. The undisciplined use of *mutexes* or *semaphores* to implement mutual exclusion on non-atomic active objects is prone to error. It is easy to misplace *wait* and *signal* operations, or even omit them altogether. Data inconsistencies are a likely result when shared data is accessed concurrently. Atomic active objects effectively implement the *monitor pattern*: All methods are guaranteed to run in mutual exclusion. The atomic active object model makes programming easier, since the effects of concurrent access to an active object need not be considered by a programmer. Fine-grained intra-object synchronisation mechanisms are superfluous. For example, a programmer does not need to identify compatibility sets for methods that may proceed concurrently.[18]

Enable Ferenczi Guards. Single-threaded active objects enable the use of method guards with semantics of concurrent critical regions as suggested by Ferenczi. This provides a high-level synchronisation constraint while avoiding the inheritance anomaly. Guards for superclasses are acquired consecutively.

Weak Mobility Support. Atomicity of method invocations makes the provision of *weak mobility* [32] easier on the part of the runtime-system: an object can be migrated to a new physical address space, whenever execution of the current request finishes and before execution of the next queued request commences. The active object can then be transferred with its complete internal state and the queue of unprocessed requests.

Transactional Active Objects. Atomic method execution could be extended to provide *transactional active objects*. A transactional active object would

[18] While a compiler could perform this task for simple methods, it would be difficult in the presence of embedded passive objects being updated.

perform the following actions under transactional control: (1) remove a request from the pending request queue, (2) processing of the request and all state changes caused to the active object and (3) request sending to foreign active objects. If a failure occurs during the processing of a request, the object's state and queue can be rolled back to the last consistent state. Otherwise modifications to queue and state are committed and all created requests sent to foreign active objects. Programs built on this principle can be resilient to non-catastrophic failures by maintaining an object-store in non-volatile memory. For example, a program could recover from the crash of a machine in the cluster. Another approach that could be taken to provide Fault Tolerance and High Availability is the replication of active objects. However, these issues are not explored any further in this dissertation.

In the literature review we exposed a trade-off related to the degree of intra-object concurrency. Although atomic active objects are easier to use, they restrict potential concurrency. For example, an atomic active object cannot implement the classic CREW (concurrent read, exclusive write) pattern. However, if we consider read-operations to be of relatively short duration, then this does not amount to a significant loss of parallelism. This problem can be further mitigated by nesting active objects: because complex systems are almost always constructed of subsystems several levels deep before getting to leaf-level components, it is a natural extension to the active object model to permit active objects to contain other active objects. Although atomic active objects do not support true intra-object concurrency, delegation to contained active objects is a reasonable substitute for many applications. More powerful however, is the concept of SODA detached methods: these allow intra-object concurrency while retaining the advantages of atomic active objects.

3.3.2 Detached Methods

Definition 3-4 (Detached Method). A *detached method* is a special active object method that is not executed on the active object thread. Instead, a new concurrent activity is spawned for every invocation. Detached methods are not externally visible and always return a `void` result. By default, they do not have access to the state of the active object in which they are declared.

As a result of this definition, detached methods allow concurrency *within* an active object. To prevent state inconsistencies through concurrent access, detached methods are not granted direct access to the active object state. This is required to preserve atomic active object semantics. If a detached method nevertheless requires access to the object state, this can be premeditated through conventional active object methods. Such calls are then normally queued and handled asynchronously by the active-object thread in mutual exclusion with all other active object invocations. This mechanism retains the consistency benefits of atomic active objects while allowing more flexibility and increased concurrency where required.

Since detached methods are not on the critical path of the active object thread, blocking calls are allowed. In particular, the `Future.get()` operation is allowed within the context of a detached method since there is no risk of blocking the active object thread.

Declaration of Detached Methods

A detached method is encapsulated by a `Detached` object instance. Code 3-1 shows the `Detached` class interface which must be extended for the declaration of detached methods. The active object can then spawn invocations of the class by calling the `start()` method. Additional parameters can be passed in subclass constructors. Within the detached method, the two `sleep` operations can be used to cause infinite or timed-out delays. The active object can interrupt such delays by calling `wakeup()`.

Code 3-1 Detached Method Interface.

```
protected abstract Detached {

  /** constructors */
  Detached();
  Detached(String name);
  /**
   * overridden by the subclass' call implementation method
   */
  abstract void
  run();

  /**
   * Start this detached method.
   */
  void
  start();

  /**
   * sleep infinitely.
   */
  void
  sleep();
  /**
   * sleep for a given number of milliseconds.
   *
   * @param   millis the sleeping time.
   * @return  true if the sleeping time was completed without
   *          interruption.
   */
  boolean
  sleep(long millis);

  /**
   * wake up a sleeping Detached
   */
  void
  wakeup();
}
```

Use Cases

The use cases for detached methods are as follows:

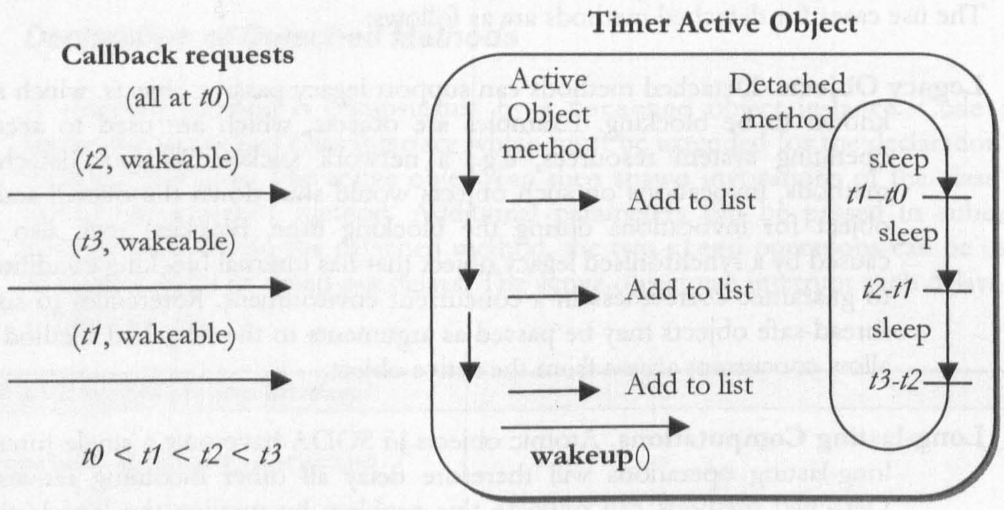
Legacy Objects. Detached methods can support legacy passive objects, which are known to be blocking. Examples are objects, which are used to access operating system resources, e.g., a network socket. Without detached methods, invocations on such objects would shut down the overall active object for invocations during the blocking time. Blocking may also be caused by a synchronised legacy object that has internal blocking conditions to guarantee correctness in a concurrent environment. References to such thread-safe objects may be passed as arguments to the detached method to allow concurrent access from the active object.

Long-lasting Computations. Atomic objects in SODA have only a single thread; long-lasting operations will therefore delay all other incoming requests. Detached methods can mitigate this problem by moving the long-lasting computation off the critical path of the active object thread.

Asynchronous Activities. For some active objects it can be useful to perform asynchronous background activities while still being able to accept incoming requests. As an example, consider a Timer active object that would be used by other active objects to schedule callbacks (see Figure 3-5). In the background, this Timer object would have a detached method that just sleeps until the earliest callback is due. However, the Timer is still able to accept requests for further callbacks since the sleeping is not performed by the active object thread. The active object can interrupt a sleeping detached method via the `wakeup` operation in order to schedule new callback times. This occurs in Figure 3-5 for the scheduling of a callback for time $t3$, where $t1 < t2 < t3$.

Bootstrap Method. The bootstrap method, which is invoked by the runtime system to start up a SODA program, is a detached method. This is required, since the Funnel return mechanism (see below) would not be available in this primordial active object method.

Figure 3-5 A Timer Active Object with Detached Method.

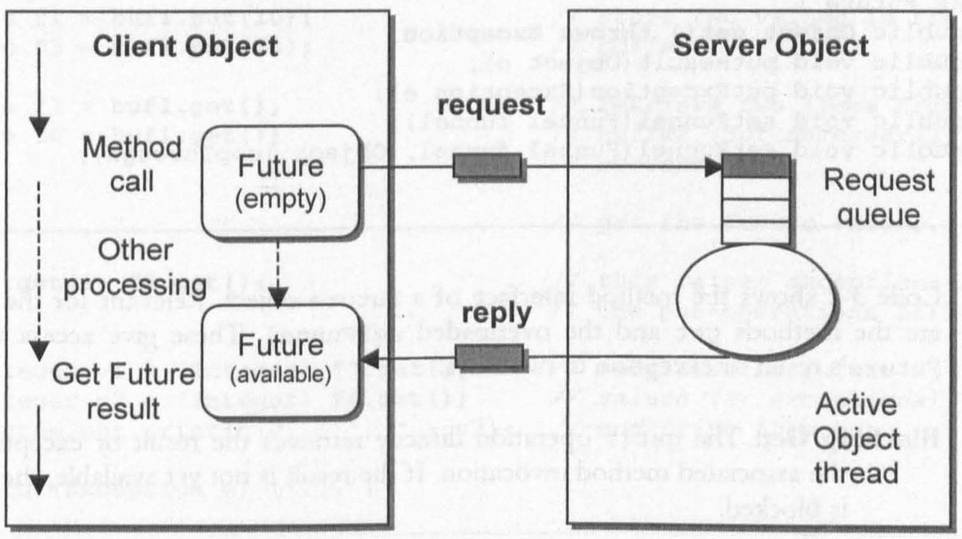


Detached methods impinge on active object mobility: during their execution, weak mobility is not applicable, because the active object state is not check-pointed. As a result, an object cannot be migrated during the execution of a detached method.

3.4 Inter-Object Concurrency and Synchronisation

Every active object method invocation conceptually entails asynchronous message exchange: The client sends a *request* message to the server. This contractually binds the server to eventually process the request and send back a *reply* message that contains the method's result. The client can proceed past the method call *immediately* without the need to wait until the arrival of the reply. SODA provides Futures as a mechanism to nevertheless enable coordinated and structured method execution by matching incoming replies to the appropriate client.

Figure 3-6 Request-Reply Message Passing between Client and Server Active Objects.



3.4.1 Futures

Client-Side

Definition 3-5 (SODA Future). A Future in SODA acts as a placeholder for the result of an active object method invocation until the reply is received. A Future object is *immediately* returned by every method invocation on an active object. This allows the client to proceed past the call although it has not yet terminated. Futures also record exceptions at the server or during network transmission of request or reply messages. Such exceptions are raised when the client tries to establish a Future's value at a later stage.

Purely asynchronous method invocation, as for example in the Actor model, is in conflict with the principles of structured programming. It is difficult to obtain the results or possible exceptions of method calls. The Future mechanism in SODA allows coordinated asynchronous method execution with the guarantee of balanced request-reply chains. Once the client obtains a Future, the exchange of request and reply messages as well as the servicing of the request by the server can occur asynchronously. Active object method calls therefore implicitly create concurrency. In SODA Futures are explicit in order to circumvent problems which are encountered with transparent Futures, e.g., by the wait-by-necessity approach (see §2.2.5).

Code 3-2 Interface of the Future Class.

```
class Future {
   public Object get() throws Exception;
   public void putResult(Object o);
   public void putException(Exception e);
   public void setFunnel(Funnel funnel);
   public void setFunnel(Funnel funnel, Object loopThrough);
}
```

Code 3-2 shows the method interface of a `Future` object. Relevant for the client are the methods `get` and the overloaded `setFunnel`. These give access to the `Future`'s result or exception in two ways:

Blocking Get: The `get()` operation directly retrieves the result or exception of the associated method invocation. If the result is not yet available, the client is blocked.

Non-blocking Funnels: Funnels provide a non-blocking, data-driven way of handling a Future. This is fundamental to the SODA model and described in detail in the next section.

If an exception was recorded on the server-side, this is re-thrown during a blocking get. Exceptions may also be caused by network failures, which is an artefact of the underlying distribution model. The blocking semantics of get introduce a significant liveness and deadlock hazard:

- Over the duration of a blocking `get()` all further pending requests on the initiating object are suspended. This is undesirable, since the initiator object is idle, waiting for replies, while it could do other useful work meanwhile. This reduces liveness and efficiency. Code 3-3 shows how `Futures` are used to invoke methods on an instance of the `Buffer` class. The result is not necessarily `"10, 20"`, since other clients might use the buffer concurrently.
- An atomic active object is not able to accept further invocations while it is blocked on a Future. Deadlock may occur in the presence of direct or indirect self-invocation.

For these reasons, SODA allows blocking gets only in detached methods.

Code 3-3 Asynchronous Calls on an Active Object with Blocking Futures

```
Future f1 = buf1.put(10);          // store two values in the
Future f2 = buf1.put(20);          // buffer

Future f3 = buf1.get();            // retrieve two items from
Future f4 = buf1.get();            // buffer (not necessarily
                                   // successive!)

try {                              // get the Future values...

   f1.get(); f2.get();             // this raises exceptions if
                                   // the put operations failed

   Integer v1 = (Integer) f3.get(); // retrieve the Future
   Integer v2 = (Integer) f4.get(); // values (or exceptions)
   System.out.println(v1 + ", " + v2); // and print them out

} catch (Exception e) { ... }
```

In the current implementation Futures are generic. This is not a restriction of the programming model per se, but an implementation detail. As a result, primitive data types must be encapsulated by Object wrappers (e.g., `java.lang.Integer` instead of `int`). It would be a desirable feature to provide typed Futures instead. This could increase performance and also improve program correctness, since the compiler could perform static type checking. Typed Futures could be implemented as parameterised classes. For example, an `int`-type Future would be declared as `Future<int>`. Parameterised classes are not supported in the current version of the Java language specification. However, this feature will probably be included in version 1.5 and experimental implementations already exists, for example the Pizza compiler.

Server-Side

In the SODA model, Futures are also visible on the server-side. This is different to Future-based models reviewed in the previous chapter. This design choice is a mainstay for the runtime-system's inlined call optimisation. It is also the basis for the Funnel mechanism. The server gains flexibility, since the Future can be shared with or produced by one of its passive objects.

Any active object method must explicitly create and return a Future. In the simplest case, the active object method makes a result or exception directly available to the Future. This is done via `putResult(Object result)` or `putException(Exception exc)` as shown in Code 3-4 and Code 3-5.

Code 3-4 Active Buffer Class Implementation (no protectection against overflow or underflow).

```
active class Buffer1 {

  private String name = null;
  int[] buf = new int[1000];
  int in = 0; int out = 0;

  public Future init(String s) {
    Future f = new Future();
    name = s;
    f.putResult(null);
    return f;
  }

  public Future put(int x) {
     Future f = new Future();
     buf[in++%buf.length] = x;
     f.putResult(null);
     return f;
  }

  public Future get() {
     Future f = new Future();
     int x = buf[out++%buf.length];
     f.putResult(new Integer(x));
     return f;
  }
}
```

Code 3-5 Active Buffer Class (with exceptions to signal overflow/underflow).

```
active class Buffer2 {

  int[] buf; int in = 0; int out = 0;

  public Future init(int size) {
    Future f = new Future();
    buf = new int[size];
    f.putResult(null);
    return f;
  }
  public Future put(int x) {
     Future f = new Future();
     if (in >= out + size)
        f.putException(new OverflowException());
     else {
        buf[in++%buf.length] = x;
        f.putResult(null);
     }
     return f;
  }

  public Future get() {
     Future f = new Future();
     if (in < out + 1)
```

```
         f.putException(new UnderflowException());
      else {
         int x = buf[out++%buf.length];
         f.putResult(new Integer(x));
      }
      return f;
   }
}
```

3.5 Funnels

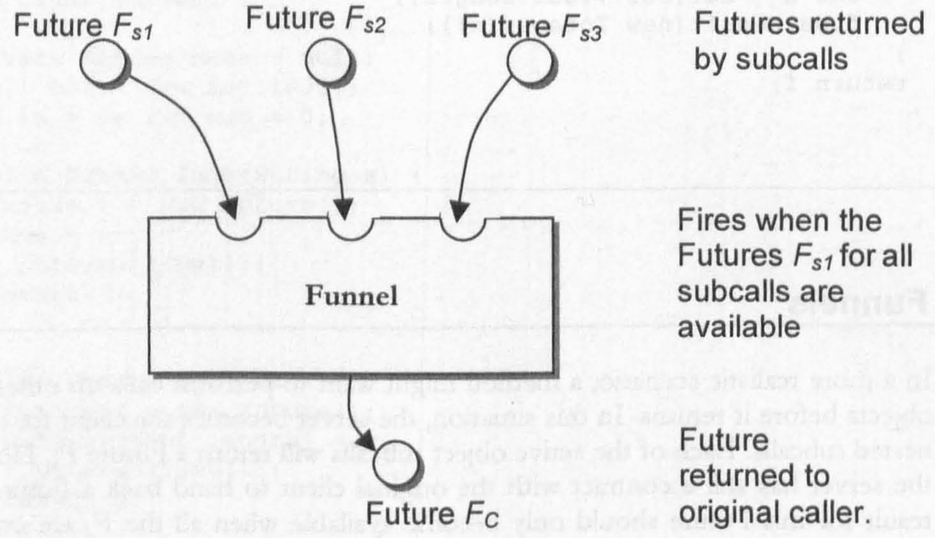
In a more realistic scenario, a method might want to perform calls on other active objects before it returns. In this situation, the server becomes the client for a set of nested subcalls. Each of the active object subcalls will return a Future F_{si}. However, the server has still a contract with the original client to hand back a Future F_c. A result for this Future should only become available when all the F_{si} are available, since F_C depends on these (see Figure 3-7). Blocking `Future.get()` operations could be used to retrieve the results of these subcalls. However, this would introduce a significant deadlock hazard in the context of atomic active objects, as explained above. For example, a direct or indirect self-invocation would deadlock the active object.

The Funnel mechanism directly addresses this problem. Funnels can effectively avoid deadlock despite direct or indirect self-invocation and they increase liveness by reducing the waiting time for pending requests (as shown in the pipeline-example in §3.8.1). With `Funnels`, an active object never gets suspended as long as pending requests are queued.[19]

Definition 3-6 (SODA Funnel). A Funnel is a return mechanism available to active object methods that perform subcalls. The Funnel asynchronously collects the Futures of the method's subcalls as their values become available. Once all required results have been retrieved, the Funnel makes a result available to the method's original client via the server-side Future. Funnels may perform some aggregate function to determine this result. Funnels have full, mutually exclusive (with active object methods) access to the active object's state. By default, if any subcall throws an exception, the Funnel ignores further outstanding subcalls and makes the exception immediately available to the original client.

[19] This statement assumes that the guards associated with the pending requests/methods invocations evaluate to true (see §3.6.1).

Figure 3-7 Dataflow in a Funnel.

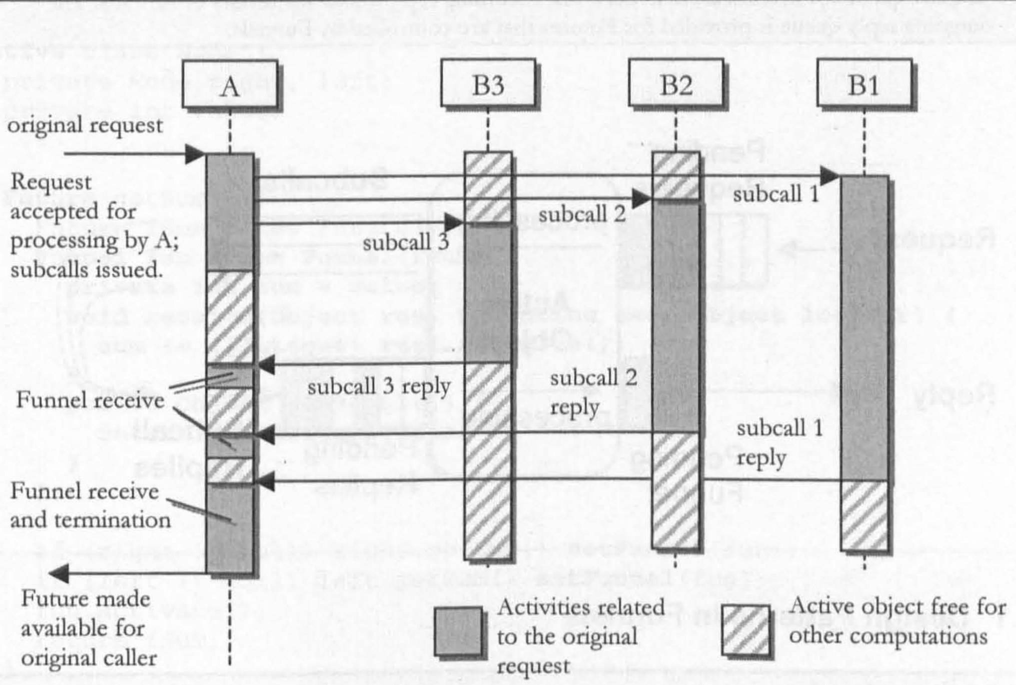

Future F_{s1}
Future F_{s2}
Future F_{s3}
Futures returned by subcalls
Funnel
Fires when the Futures F_{s1} for all subcalls are available
Future returned to original caller.
Future F_C

An active object method with subcalls never makes a result available to its Future directly. This is the responsibility of the Funnel. As Figure 3-7 shows, a Funnel connects together a set of client-side Futures from nested subcalls with a single server-side Future that is handed back to the method's original client. This collection of subcall Futures into the server-side Future is performed by a Funnel in a non-blocking, data-driven manner that resembles dataflow: `Funnels` become activated in a data-driven manner when all the Futures it is set on become available or an exception is thrown.[20] Funnels are therefore similar to nodes in a dataflow graph: they represent aggregate instructions that are triggered by the arrival of relevant data. Each subcall Future acts as a token. The Funnel fires when all the tokens are available. However, while pure dataflow is a very fine-grained approach, treating nodes as individual instructions, this is more in line with recent macro dataflow ideas.

Futures in combination with Funnels avoid unnecessary sequencing among invocations of successive methods. As soon as the original client has issued successive calls and set up corresponding Funnels, it is free to start processing of the next message queued without waiting for a result (see Figure 3-8).

[20] It is not currently possible to "cancel" outstanding subcalls collected by a Funnel, when one of the subcalls throws an exception. Therefore, all subcalls will be executed. In the case of several exceptions thrown in different subcalls, only the first one is by default passed back to the client. However, this behaviour may be overridden to account for all exceptions and perform some aggregate functionality (see 3.5.1 and 3.5.2 below).

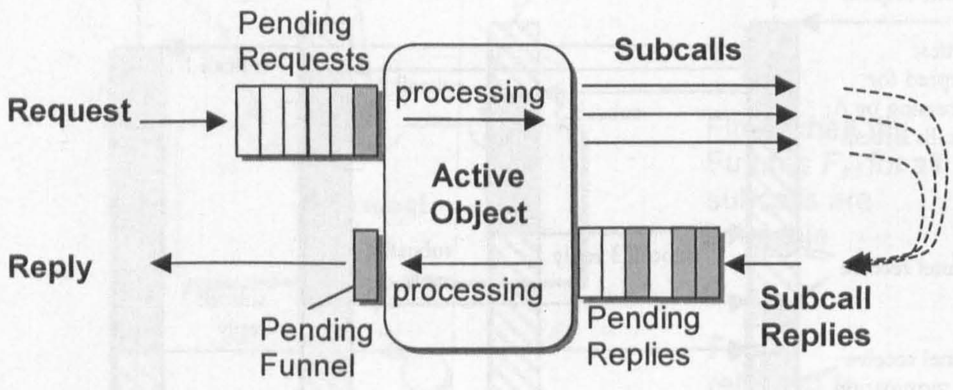
Figure 3-8 Funnel Operation


A
B3
B2
B1
original request
Request accepted for processing by A; subcalls issued.
subcall 1
subcall 2
subcall 3
Funnel receive
subcall 3 reply
subcall 2 reply
subcall 1 reply
Funnel receive and termination
Future made available for original caller
Activities related to the original request
Active object free for other computations

An active object method has a contract with its client to make available a result (or exception) to the Future it hands back. This responsibility is taken over by the Funnel for this method. The single client Future is given as argument to the Funnel constructor. Subcall Futures are associated with this Funnel via the operation `Future.setFunnel(Funnel f)`.

`Funnels` are processed in mutual exclusion with method requests on the active object; they can therefore access object-local data without creating inconsistencies. `Funnels` are given priority over pending requests in order to minimise servicing latency for clients. Requests and replies traverse an active object in opposite directions (see Figure 3-9).

Figure 3-9 Dynamics of a SODA Active Object.
Request queue for new method invocations, Incoming reply queue for results of subcalls. The outgoing reply queue is provided for Futures that are controlled by Funnels.



3.5.1 Design Patterns in Funnels

Code 3-6 Funnel Interface.

```
class Funnel {
    protected void receive (Object result, Exception exception,
                                             Object loopThrough);
    protected void terminated();
    protected Object getResult();
    protected Exception getException();
}
```

Funnels can implement a set of design patterns for collecting a set of subcalls. In the default case, the Funnel simply waits on availability of all subcall Futures. Once all subcall Futures become available, the Funnel produces a `null` result for the client Future.

This default behaviour implies a synchronisation on complete termination of subcalls. In contrast, a `PartialFunnel` might synchronise on the partial availability of `Futures`. Consider for example a search operation in a tree. If the query is successful in one of the branches, there's no need to continue evaluation of the other branches or to wait on their result. Partial availability of results might also be useful to implement replicated active objects, where one reply of a replicated object is sufficient for program continuation.

A useful feature in this context would be the ability to cancel outstanding subcalls, which are queued as requests at foreign active objects, but not yet evaluated. An operation `Future.cancel()` could invalidate calls which do not have side-effects. This would save processing resources. This mechanism is not currently implemented in the SODA runtime-system.

Code 3-7 Aggregate Funnel as coud be used in the Node of a Binary Tree.

```
active class Node {
 private Node right, left;
 private int value;
 ...

 Future getSum() {
   Future fSum = new Future();
   Funnel fun = new Funnel(fSum) {
     private int sum = value;
     void receive(Object res, Exception exc, Object loopThr) {
       sum += ((Integer) res).intValue();
     }
     public Object getResult() {
       return new Integer(sum);
     }
   };

   if (right != null) right.getSum().setFunnel(fun);
   if (left != null) left.getSum().setFunnel(fun);
   fun.activate();
   return fSum;
 }
}
```

Other frequently used patterns in `Funnels` are aggregate operations. Code 3-7 above shows how an aggregate funnel is created as anonymous inner class [72] of the active class `Node`. The `getSum` operation recursively sums all elements in a tree of `Nodes`. As the `getSum` operations of the right and left sub-tree return, their values are added to the Node's value. `getResult` then returns the accumulated result once all outstanding futures have returned (and the Funnel has been `activated`).

By default the first subcall exceptions that a Funnel encounters is propagated towards the root of the call hierarchy immediately, without waiting on the result of further pending subcalls. This behaviour can be modified to provide a context in which to catch and handle the asynchronously encountered exception. This mechanism can be used to handle the situation where more than one exception is raised by a subcall.

3.5.2 Custom Funnels

`Funnel` functionality can be further customised if the above predefined design patterns are not sufficient. For this purpose, `Funnel` subclasses can override the interface in Code 3-6:

- **`receive(Object result, Exception ex, Object loopThrough)`** is called whenever a subcall `Future` becomes available. `result` and `exception` are determined by the subcall. An optional `loopThrough` parameter can be used on a per-subcall (i.e., per-Future) basis. This allows a Funnel to disambiguate between a set of subcall results that it collects. To set up a `loopThrough` parameter the overloaded operation `setFunnel(Funnel funnel, Object loopThrough)` is provided.

- **`terminated()`** is called when all subcalls have terminated.
- Finally, **`getResult()`** and **`getException()`** are invoked after `terminated`, to retrieve a result and exception for the original client's Future underlying this `Funnel`, respectively.

3.6 Message Acceptance Policy

The class in Code 3-5 throws an exception when a buffer underflow or overflow would occur. For example, the `put()` method throws an `OverflowException` when the buffer space is exhausted. This is appropriate when the buffer object is accessed sequentially. Active objects, however, exist in a concurrent environment where they may be used by multiple clients more or less simultaneously. Therefore it makes sense to delay the `put()` request until after buffer space becomes available as a result of a concurrent `get()` request. Such a reordering of method invocations increases the number of successfully served requests. An acceptable reordering or servicing policy is implicitly defined in SODA through *method guards*.

3.6.1 SODA Method Guards

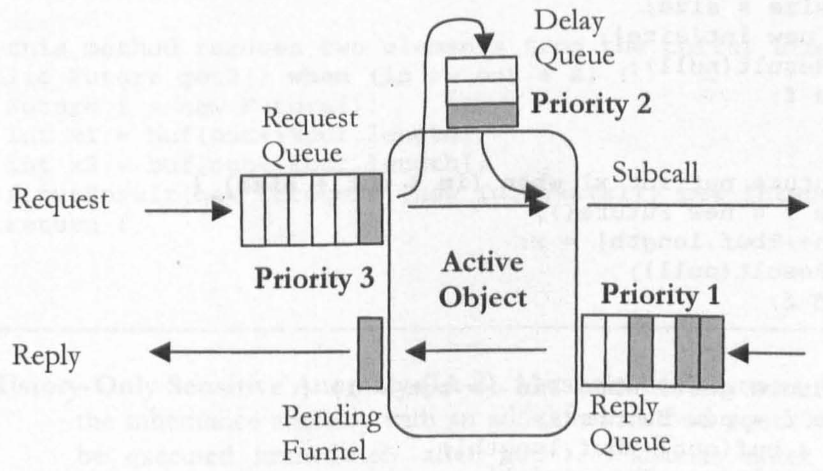
Definition 3-7 (SODA Method Guards). A SODA method guard is optionally attached to an active object method. A guard is a side-effect free Boolean expression that can be based on the active object's state or on the value of the request parameters. At runtime, a call to a guarded method is suspended until the guard expression evaluates to true. Subclass methods may call the superclass only once and as their first statement. Such a method can only proceed, if the logical conjunction of all guard expressions along the inheritance hierarchy evaluates to true.

It is not predictable how often a method guard will be evaluated at runtime. Therefore, the guard expression must be free of side-effects, i.e., it should not cause changes to the active object's state. If guard expressions are overly complex this also has negative consequences on performance.

In an inherited class, an invocation of a guarded superclass method must be the first statement in a subclass method. This guarantees that the invocation of the subclass method and the overridden method occurs as an indivisible operation. Guards are acquired successively along the inheritance hierarchy. If at any one level a guard evaluates to false, the overall method invocation is abandoned and the request is rescheduled. Since up to this point only side-effect free guard expressions have been evaluated, the active object's resource invariant is still intact because it has not yet been modified. With these semantics, method guards avoid the inheritance anomaly (see next section).

If a request does not fulfil its guard in the current object state, it is enqueued in a special *delay queue* (see Figure 3-10). A request in the delay queue is served with higher priority than a conventional request once its accumulated guard evaluates to true. Only pending subcall replies take precedence over the delay queue.

Figure 3-10 Priority of Message Processing at an Active Object.
Pending incoming Replies take priority over pending Requests. Pending Outgoing replies are sent whenever their Funnel has collected the incoming Replies of all subcalls. If the Delay Queue has entries, these are processed after every Request and every Reply.



Syntax

Guards are declared with the keyword **when** as a suffix to the method signature. Code 3-8 shows an example bounded buffer class with method guards. The guards in this class prevent buffer underflow and overflow based on the object's internal state. Code 3-9 shows an alternative method guard for the put method that is based on the value of a request parameter: in addition to checking for availability of buffer space, this modified guard only accepts a request when the value to be stored is less than 100.

Code 3-8 Original Bounded Buffer Class with Method Guards

```
active class BoundedBuffer {

  int[] buf; int size = 0;

  int in = 0; int out = 0;

  public Future init(int size) {
    Future f = new Future();
    this.size = size;
    buf = new int[size];
    f.putResult(null);
    return f;
  }

  public Future put(int x) when (in < out + size) {
    Future f = new Future();
    buf[in++%buf.length] = x;
    f.putResult(null);
    return f;
  }

  public Future get() when (in >= out + 1) {
    Future f = new Future();
    int x = buf[out++%buf.length];
    f.putResult(new Integer(x));
    return f;
  }
}
```

Code 3-9 Revised Method Guard for put

```
  public Future put(int x) when ((in < out + size) && (x < 100)) {
    ...
  }
```

3.6.2 Absence of the Inheritance Anomaly

SODA method guards can be subclassed without causing inheritance anomaly, since their semantics follow Ferenczi's proposal. Subclass methods can call on superclass methods that are guarded. Guards are evaluated recursively for every level of the inheritance hierarchy and acquired as conditional critical regions. If at any stage a guard evaluates to false, the overall request is aborted and rescheduled into the delay queue.

Based on the bounded buffer class from Code 3-8 we show derived classes that provide a solution to all three instances of the inheritance anomaly:

State partition anomaly (IA-1). According to Matsuoka [113] state partition anomaly does not affect method guards. Guards can be composed directly on the object's state space independently of each other, rather than relying on pre-defined subsets of object state space. Code 3-10 shows an example solution for the *X-Buf* subclass that has a method get2 to retrieve two elements atomically from the buffer.

Code 3-10 X-Bounded Buffer in SODA

```
active class XBoundedBuffer extends BoundedBuffer {

   public Future init(int size) { return super.init(size); }

   // this method removes two elements from the buffer atomically.
   public Future get2() when (in >= out + 2) {
      Future f = new Future();
      int x1 = buf[out++%buf.length];
      int x2 = buf[out++%buf.length];
      f.putResult(new Integer[]{new Integer(x1), new Integer(x2)});
      return f;
   }
}
```

History-Only Sensitive Anomaly (IA-2). Matsuoka demonstrates this instance of the inheritance anomaly with an additional method gget() which cannot be executed immediately after put(). Matsuoka notes that guarded methods put and get must be completely reimplemented in the derived class [*G-Buf*] as a result. Code 3-11 shows an alternative which avoids inheritance anomaly, following Ferenczi's proposal. In this example, pre-existing code is reused and the effective guard conditions are accumulated along the inheritance hierarchy. i.e. method gget of the subclass can only proceed if the combined condition (!afterPut) & (in >= out + 1) holds.

Code 3-11 G-Bounded Buffer in SODA

```
active class GBoundedBuffer extends BoundedBuffer {

   private boolean afterPut = false;

   public Future init(int size) { return super.init(size); }

   // this method cannot execute as immediate successor to put().
   public Future gget() when (!afterPut) {
      return get();
   }

   public Future put(int x) when (true) {
      Future f = super.put(x); afterPut = true;
      return f;
   }

   public Future get() when (true) {
      Future f = super.get(); afterPut = false;
      return f;
   }
}
```

State Modification Anomaly (IA-3) occurs as a result of modifying the set of states under which the original methods can be invoked. In the example of the `L-Buf` class, the addition of a lock method introduces finer-grained distinctions for the set of states under which methods put and get can be invoked. Again, Matsuoka proposes a complete reimplementation of the method guards. However, as Code 3-12 shows, this problem can be solved without causing inheritance anomaly, when guards are accumulated along the inheritance hierarchy.

Code 3-12 L-Bounded Buffer in SODA

```
active class LBoundedBuffer extends BoundedBuffer {

   private boolean locked = false;

   public Future init(int size) { super.init(size); }

   public Future lock(boolean lockState) {
      Future f = new Future();
      locked = lockState;
      f.putResult(null);
      return f;
   }

   public Future get() when (!locked) {
      return super.get();
   }

   public Future put(int x) when (!locked) {
      return super.put(x);
   }
}
```

As these examples show, SODA method guards, based on the semantics proposed by Ferenczi do not expose inheritance anomaly for the example cases identified by Matsuoka.

3.6.3 Expressive Power according to Bloom's Criteria

Bloom [27] developed several criteria for evaluating the expressive power of synchronisation constraints. She suggested a set of five criteria by which a server object could be allowed to define its servicing policy. SODA guards fulfil only the last three of Bloom's criteria:

Type of Request. The server object should be able to select messages for execution, depending on the method called (e.g., "calls to hold must be serviced before calls to allocate"). This constraint cannot be expressed with SODA guards, since it would require direct access to the pending request queue in order to determine which request types are available.

Order of Request. An object should be able to accept requests in FIFO order, or priority of caller order. In SODA, invocations for a given method are always accepted in FIFO order, given that their guards evaluate to true. Priority of caller would require a direct manipulation of the pending request queue which is not supported in SODA.

Request Parameters. A method's parameters should be a criterion for acceptance by the server object. This is possible with SODA guards as shown in Code 3-9.

Local State. Message acceptance should be in relation to the object's internal state. Again, this can be achieved via SODA guards.

History Information can be used to allow only certain patterns of subsequent message invocations. Since such history constraints can be expressed through local state (see as an example the *G-Buf* class in Code 3-11), this can be expressed in SODA.

Type-of-request constraints could be supported by counters that keep track of all pending invocations for a given method, similarly to the operators underlying Deontic logic. Method guards could then be based on the value of these counters, which could be implemented without significantly impacting on performance.

Similarly, priority-of-caller could be implemented. This would entail the sending of priority information with every request. Active objects should then have a priority-queue that would serve highest-priority requests first.

3.6.4 Self-Invocation and Guards

Self-invocation occurs when an active object calls a method recursively on itself. Two forms of self-invocation are possible in SODA: *Immediate calls* take priority over conventional requests; they are performed within the context of the original method. In contrast, *indirect calls* follow normal queuing patterns.

Immediate calls are achieved via the reference `this`; however, this mechanism is problematic since it bypasses the object's request queue and leads to unfair scheduling. External pending requests could be delayed for a long time, until the object-internal recursion is finished. The other issue with immediate calls is that they effectively call another method before the current method invocation is finished, which could lead to inconsistencies.

Indirect calls guarantee fairness for external clients and mutual exclusion of active object method invocations. Calls behave like external invocations and are merged into the active object request stream. Such invocations are possible via the self-reference `thisActive` that is implicitly defined in all active objects. Guards can be used to lock the active object for other invocations until the recursion is finished (see the merge-sort example below).

3.7 Parameter Copying Optimisations

The SODA model is based on deep-copying semantics for passive objects when these are passed as parameter or result of an active object method call. This is necessary, because passive objects cannot be shared between active object instances. They can only be referenced from within the context of a single owning active object.

Depending on the object structure, deep-copying can cause significant performance overheads. In some situations where active objects are collocated in the same physical address space it is semantically equivalent to pass passive objects by reference. The runtime-system relies on compile-time information to perform this optimisation. For this purpose, two hints related to the passing of passive objects between active objects can be given to the SODA compiler.

Immutable Objects. If a passive object never changes its value, it can be marked as *immutable*. An immutable object can then be transferred by-reference between active objects within the same physical address space; the semantics are equivalent to deep-copying. Immutability can be defined on class-basis or instance-basis.

Hand-Over Parameters. Another optimisation is available when a passive object is relinquished after it has been passed in a request or reply. For example, a passive object could be created simply as an argument for a method invocation and then be discarded by the client. Vice versa, a return value might not be used by the server once it has been handed back to the client. In this situation, it is unnecessary to deep-copy the passive object at runtime when client and server are collocated. Such parameters can be marked as hand-over parameters.

3.8 Evaluation

This section aims at a brief empirical evaluation of the concepts underlying the SODA programming model. While some authors criticise object heterogeneity for its reduced reuse potential (in rare cases two versions of a class may need to be developed) we think that the opposite is the case: the heterogeneous model enables the reuse of already existing classes written for sequential settings, which actually increases reuse.

3.8.1 Types of Parallelism Supported

Buyya [31] summarises the following as main paradigms of parallel programming. Based on some examples we will examine how well these paradigms can be supported by the SODA model. While SODA supports divide-and-conquer type parallelism in a natural way, it has considerable flexibility for the support of other types of parallelism.

Task-Farming (or Master/Slave). This is the typical message-passing model, based on a set of communicating concurrent processes. This paradigm creates concurrency through the definition of processes and therefore does not map onto SODA's call-based concurrency. Neither does SODA support one-way message passing.

Single Program Multiple Data (SPMD). SPMD or data parallelism can be supported in SODA by creating multiple instances of the same active object. A central controller object can then invoke the same operation on all objects, which would perform slightly different computations as determined by their state. There are two problems with this approach: Firstly, the controller is a central communication bottleneck. Secondly, the SODA model does not currently provide a means to perform a multicast invocation of a set of data-parallel active objects. Thirdly, the transport mechanism for remote calls does not provide an efficient and direct multicast mechanism for requests.

Data Pipelining. Data pipelining can be modelled as chained invocation of a set of active objects as demonstrated below. Note that this is not equivalent to hardware pipelining where every pipeline stage has the same cycle time.

Divide and Conquer. Divide-and-conquer type parallelism can be captured by a recursive subdivision of the problem domain into active objects. A call to the root object would then recursively span the tree of active objects.

Speculative Parallelism. Speculative parallelism is not really a structural approach like the other paradigms mentioned above. For example, a program could devote effort to speculatively trigger active object invocations when resources are idle. One problem here is that SODA Futures do not currently support a functionality that would allow cancellation of pending requests if these are no longer relevant. Such a feature could also be useful for parallel search algorithms: If the element to search for is found in some partition of the search set, the search in other branches could be cancelled.

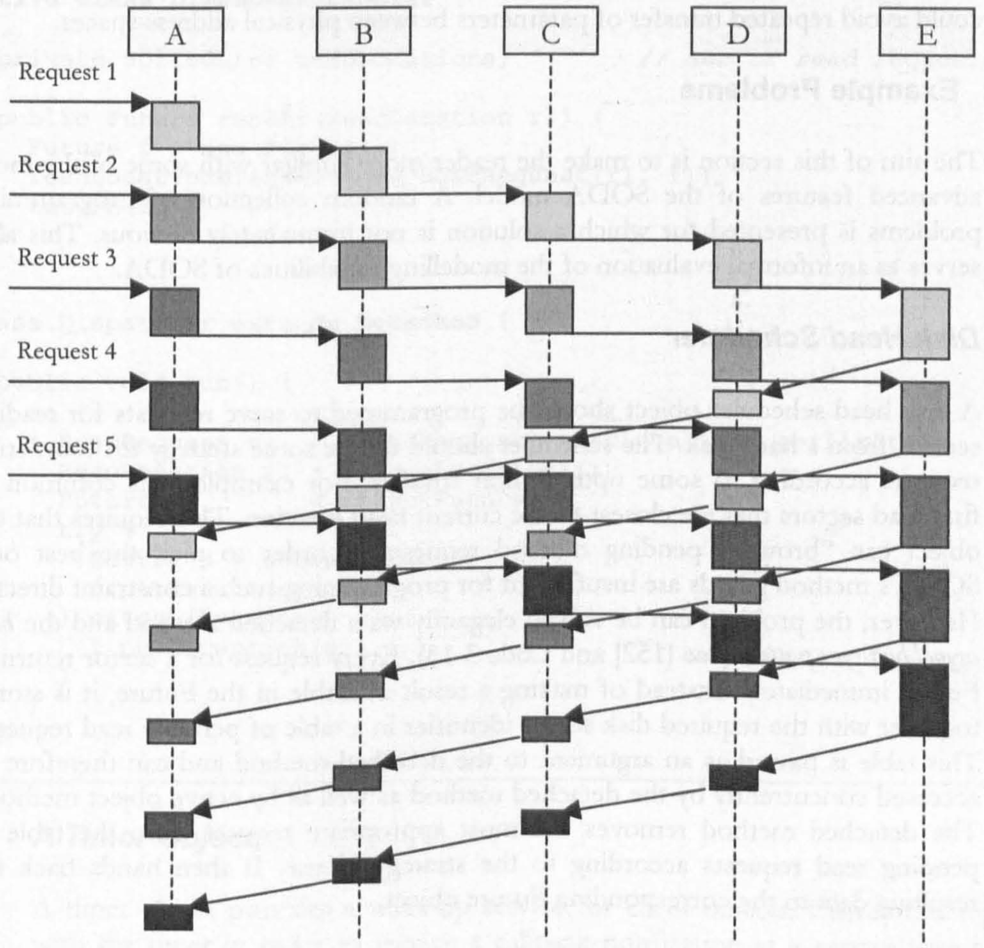
Pipeline Parallelism

It is straightforward to express pipeline-style parallelism across a set of active objects. A chain of active objects is formed, with the leader triggering off a pipeline cycle: any invocation is recursively passed on as a subcall to the right follower active object. Conventional blocking Futures combined with atomic active objects would render such a structure very inefficient. For example, in ProActive PDC, the header object would block all further invocations until the recursive subcalls have terminated.

In SODA, the Futures and Funnels mechanism can be used to achieve parallel processing across all objects in the pipeline for different sets of invocations. Each object sets up a Funnel to collect the result of its subcall and to hand it back to the object on its left. In contrast to blocking Futures, the leader is not blocked, after it has passed off data to its right. Instead, it can immediately accept new data to be processed through the pipeline (see Figure 3-11). Leader and followers can thereby work in parallel, as long as the data stream provides new inputs: in mutual exclusion, each pipeline element handles new requests from the left and replies from the right.

Of course, the same degree of concurrency can be achieved with non-atomic active objects. However, consider the problem of providing mutual exclusion of the invocations for each object in the pipeline. It would then be necessary to provide a monitor that only encloses the section of the method before the subcall occurs.

Figure 3-11 Pipeline Parallelism with Active Objects.



Functional Parallelism (Divide-and-Conquer)

Again, Funnels can be used to program functional parallelism corresponding to a divide-and-conquer strategy. This is demonstrated in Code 3-7, where subcalls to the left and right branch of a binary tree are processed in parallel. Functional parallelism is exploited at every branching level of the tree. In addition, to this horizontal parallelism, multiple invocations on the root can take advantage of vertical parallelism, similar to the above pipelining scenario. This means that multiple tree traversals can be in flight concurrently (vertically), while every single traversal can be internally concurrent (horizontally).

Data Parallelism

Multicasting to Objects is not supported yet in the SODA run-time system. Data parallelism would be easy to integrate into the programming model, using active object arrays as destination for method calls. Results could be collected using Funnels. This functionality could be implemented using Futures and Funnels. However, this does not allow for overhead optimisation. e.g., the runtime system could avoid repeated transfer of parameters between physical address spaces.

3.8.2 Example Problems

The aim of this section is to make the reader more familiar with some of the more advanced features of the SODA model. A random collection of programming problems is presented for which a solution is not immediately obvious. This also serves as an informal evaluation of the modelling capabilities of SODA.

Disk Head Scheduler

A disk head scheduler object should be programmed to serve requests for reading sectors from a hard disk. The scheduler should follow some strategy for reordering requests according to some optimisation strategy. For example, it is common to first read sectors that are closest to the current head position. This requires that the object can "browse" pending method requests in order to pick the best one. SODA's method guards are insufficient for programming such a constraint directly. However, the problem can be solved elegantly via a detached method and the *half-async/half-sync pattern* (see [152] and Code 3-13). Every request for a sector returns a Future immediately. Instead of making a result available in the Future, it is stored together with the required disk sector identifier in a table of pending read requests. This table is passed as an argument to the detached method and can therefore be accessed concurrently by the detached method as well as by active object methods. The detached method removes the most appropriate request from the table of pending read requests according to the strategy in use. It then hands back the resulting data to the corresponding Future object.

Code 3-13 A DiskHeadScheduler Active Object

```
active class DiskHeadScheduler {
  ...
  private SortedList readLocations;          // sector read requests

  public Future readAt(ReadLocation rl) {
    Future f = new Future();
    readLocations.insert(new ReadRequest(rl, f));
    return f;
  }
}

class Dispatcher extends Detached {
  ...
  public void run() {
    do {
      ReadRequest rr  = (ReadRequest) readLocations.getFirst();
      ReadLocation rl = rr.getLocation();
      Future f        = rr.getFuture();
      try {
        Sector s = getSector(rl);
        f.putResult(s);
      } catch (Exception e) {
        f.putException(e);
      }
    }
  }
}
```

A Timer Object

A timer object provides a wake-up service for client objects. Clients can register with the timer in order to receive a callback notification at a certain time in the Future. Such a timer can be implemented in SODA as an active object according to the half-async/half-sync pattern, similar to the disk head scheduler above.

An active object method (the asynchronous part) accepts a reference to the callback object together with the wakeup time. These two elements constitute an event. The method returns a Future result immediately and inserts the event into a scheduling list. A detached method (the synchronous part) operates on this list. The method waits until the wakeup time of the most imminent event is reached and then invokes the callback method. If during the waiting time another event is scheduled with an earlier time, the wait is interrupted and the list re-evaluated. An example Timer object is provided in the `class uk.ac.ncl.soda.util.Timer`. Client objects must implement the `uk.ac.ncl.soda.util.Wakeable` interface for callback notification.

Code 3-14 A Timer Active Object

```
active class Timer {
  ...
  private Detached det;        // reference to an instance of Wait
  private LinkedList events; // scheduling list

  public Future scheduleAbsolute(long millisAbs, Wakeable wa) {
    ((ActiveProxy) wa).setOwner(thisActive);
    WakeUpCall wuc = new WakeUpCall(millisAbs, wa);
    events.add(wuc);
    Collections.sort(events);
    if (wuc.equals(events.getFirst())) {
      det.wakeup();
    }

    Future f = new Future();
    f.putResult(null);
    return f;
  }

  protected Future getFirst() {
    Future f = new Future();
    Object wuc = events.getFirst();
    f.putResult(wuc);
    return f;
  }

  public Future removeFirst() {
    ...
  }
}

class Wait extends Detached {

  public void run() {
    try {
      do {
        do {
          Future f = thisActive.getFirst();
          WakeUpCall wuc = (WakeUpCall) f.get();
          // sleep infinitely if there are no wake-up calls scheduled
          if (wuc == null) sleep(-10);
        } while (wuc == null);

        long millis = wuc.millis - System.currentTimeMillis();
        boolean succeeded = sleep(millis);

        if (succeeded) {
          thisActive.removeFirst().get();
          wuc.wa.wakeUp();
        }
        else continue;
      } while (true);
    } catch (Exception e) {
      ...
    }
  }
}
```

The Dining Philosophers

Model each philosopher (see Code 3-16) and each fork (see Code 3-15) as an active object. Forks can be picked up and put down (lock and release). Every philosopher sends a lock request to the forks on his left and on his right. A Funnel (declared s anonymous inner class [72]) is used to control the results of the lock requests. If both requests succeed, the philosopher can eat. Finally, both forks are released again. One interesting thing here is that Future subcalls are issued not only by the active object method itself, but also by the Funnel. For this reason, the `activate` call is not done in the `eat` method but in the Funnel, after the calls to `Fork.release()`.

The receive method in the Funnel is now responsible for collecting results of the two `lock` methods **and** the two `release` methods. To distinguish which type of response is expected, we introduce an additional variable `state` that has either the value `ACQUIRING_LOCKS` or the value `RELEASING_LOCKS`. Depending on that value, the receive method behaves differently. Extra complexity is therefore introduced in an attempt to restrict concurrency (release calls should only be issued after lock calls). This is appropriate and desirable for a programming strategy that fosters the exposure of concurrency rather than its restriction.

Code 3-15 A Fork Active Object

```
active class Fork {                    //must be active, because shared
  Philosopher current = null;

  public Future lock(Philosopher p) {
    Future f = new Future();
    if (current == null) {
      current = p;
      f.putResult(new Boolean(true));
    } else {
      f.putResult(new Boolean(false));
    }
    return f;
  }

  public Future release(Philosopher p) {
    Future f = new Future();
    if (current.equals(p)) {
      current = null;
    }
    f.putResult(null);
    return f;
}
```

Code 3-16 A Philosopher Active Object

```
active class Philosopher {

 private Fork right, left;

 public Future init (Fork right, Fork left) {
   this.right = right;
   this.left = left;
 }

 public Future eat() {
   Future f = new Future();

   Funnel fun = new Funnel(f) {
     private int received = 0;
     int state = ACQUIRING_LOCKS;
     receive(Object result, Exception exc, Object loopThrough) {
       if (state == RELEASING_LOCKS) return;
       received++;
       if (((Boolean) result).boolValue()) locked++;
       if (received == 2) {
         if (locked == 2) {
           // eat ......
           state = RELEASING_LOCKS;
         } // end if locked == 2
         left.release(thisActive).setFunnel(this);
         right.release(thisActive).setFunnel(this);
         this.activate();
       } // end if received == 2
     }
   };

   left.lock(thisActive).setFunnel(fun);
   right.lock(thisActive).setFunnel(fun);
   return f;
 }
}
```

Finite Element Simulation

Every cell in a finite element simulation can be modelled as an active object. However, the programmer needs to decide on the optimum cell size. Sometimes results at various granularities are required. It is then best to choose the minimum granularity that could be exploited on the lowest-latency distributed machine this algorithm was ever to run on.

Merge-sort Example

How can an active object representing a list of numbers implement a parallel sorting algorithm? We describe the SODA solution based on the merge-sort algorithm. Merge-sort recursively divides the list into two halves and calls `sort()`

on these halves. The two halves are then merged. When a certain threshold size is reached, the remainder lists are sorted via quick-sort and recursion terminates.

We implement lists as active objects. A list object is subdivided by removing half of its entries and creating a new list object based on these entries. This minimises active object creation and copying overheads. Since the shortened list object does not require access to the second half after subdivision, entries can be passed as hand-over parameters.

The original list object (A) is in an inconsistent state until the sorting operation terminates. It is therefore necessary to introduce an additional locking mechanism (through method guards) that prevents concurrent access to the list while sorting is in progress.

Figure 3-12 Recursion with Future Subcalls

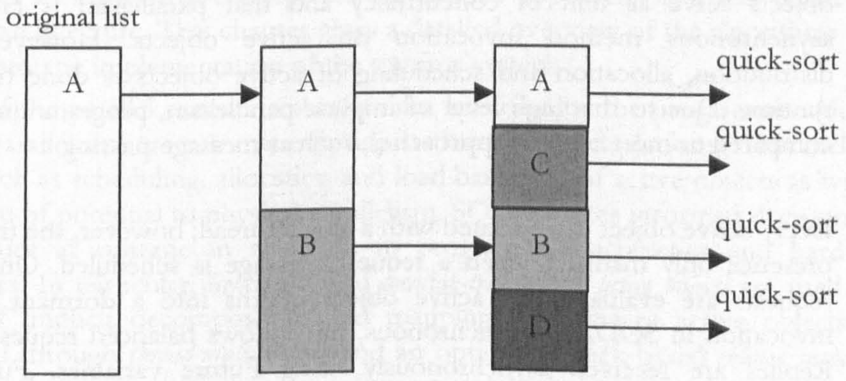
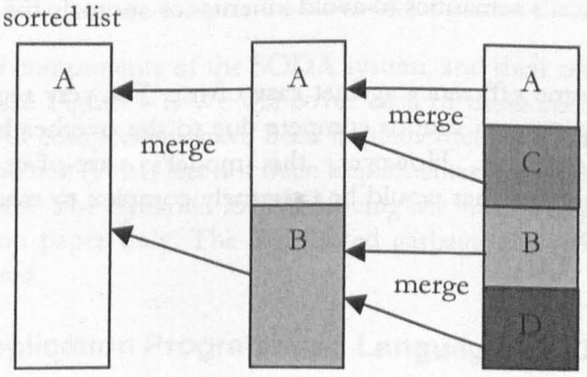


Figure 3-13 Merging of Sublists through Funnel Operation



Of course, one issue here is the finding of a correct threshold value for switching from active object merge-sort to quick-sort. Once the threshold is reached, no further sublist active objects are created. As mentioned before, SODA active objects are very lightweight. Nevertheless, they carry a higher method call overhead that can only be amortised by sufficient method call granularity. Insofar, the programmer has to be aware of this dual-stage processing and find an appropriate threshold value, based on object granularity and system performance.

Summary

This informal collection of examples showed that the SODA model can address a range of programming problems in an easy-to-use manner. Funnels increase concurrency, while maintaining the benefits of atomic active objects. Detached methods can provide functionality similar to early-reply.

3.9 Conclusion

The SODA model allows a programmer to directly write object-oriented, parallel and distributed applications. A programmer must be aware of the fact that active objects serve as unit of concurrency and that parallelism is created through asynchronous method invocation on active objects. However, the actual distribution, allocation and scheduling of active objects is done transparently at runtime. Due to this high level of implicit parallelism, programming is simplified compared to more explicit approaches, such as message-passing.

Every active object is associated with a single thread; however, the thread makes its presence only manifest when a request message is scheduled. Once all pending requests are evaluated, an active object returns into a dormant state. Method invocation in SODA is asynchronous, but follows balanced request/reply chains. Replies are received asynchronously using Future variables. Futures may be resolved in a blocking manner, which however is not safe since a deadlock potential is introduced. Alternatively, Futures and Funnels can be used as non-blocking, data-driven continuation mechanism. Requests arriving at an active object are queued and processed in a sequential order. This order is FIFO by default but reordering constraints may be imposed if method guards are declared. Method guards follow Ferenczi's semantics to avoid inheritance anomaly for active objects.

SODA trades off some efficiency against ease-of-use. For very regular, numerical algorithms SODA programs cannot compete due to the overheads caused by the high level of abstraction. However, the model's ease-of-use enables the construction of programs that would be extremely complex to manage with more traditional approaches.

Chapter 4

Implementing SODA

SODA has been implemented as a source-to-source translator and a runtime system. In this chapter, we illustrate the design choices made in this system. The main concern was to correctly implement the model described in Chapter 3. The secondary concern was to implement the model efficiently. The purpose of our implementation is to serve as a vehicle for further research and development. It also serves as a proof-of-concept for the SODA approach on a real distributed memory architecture. This chapter gives a detailed overview of the algorithms used in the prototype implementation of the runtime system.

Details of the parallel execution that are not specified by the programmer (due to implicit parallelism) are automatically adjusted to the target platform. This includes issues, such as scheduling, allocation and load balancing of active objects as well as adaptation of potential to physical parallelism. SODA makes informed decisions on these issues at runtime in response to program characteristics and hardware capabilities. In particular, *lightweight* and *location-transparent active objects* are used as a means of implicit decomposition and mapping. Lightweight active objects are supported through *thread-multiplexing* and an optimised stack-based *inlined invocation* technique that is used in the case of *unexpected locality*. Location-transparency is provided through *transparent proxies*. Further optimisations reduce the marshalling costs for remote method invocations.

4.1 Overview

The major components of the SODA system, and their relationship to one another, are shown in Figure 4-1. We will cover each of these components in the following sections. All components have been implemented, with three exceptions. The Java Virtual Machine (JVM) has not been implemented by us; instead, any standard JVM can be used. The dynamic load balancing via object migration module has been designed on paper only. The distributed garbage collection scheme has not been implemented.

4.1.1 The Application Programming Language: SODA

The SODA programming model is not tied to a particular application programming language. However, we have implemented it through the SODA language. This language does not invent an entirely new syntax. Instead it adds a few keywords to Java as a host language in order to reuse existing technology as much as possible. The keywords `active` and when as outlined in chapter 3 are used to declare active objects and guards.

A source-to-source translator has been built that converts the SODA language into Java. For every active object declaration, a set of Java classes are generated that call into the SODA runtime system library. The resulting Java class declarations can then be compiled using a standard Java compiler.

At an early stage during this project, we experimented with runtime reflection, especially dynamic proxies, to implement SODA. However, we switched to the translation approach because the overheads of reflection were found to be too high. Alternatively, SODA could also have been implemented through a library-based approach. However, we decided for a language-based approach based on the following factors:

- Direct use of a library would require detailed knowledge of the active object implementation and would make programming more error-prone. The translator can hide many of these difficulties and automatically enforce correct library usage.
- A language based approach retains a clear mapping between design and implementation. Active objects can be directly expressed at the source code level, which provides a clean conceptual model and hides repetitive house-keeping code from the programmer. Therefore, the active object environment is easier to understand and the programmer can think at a higher level of abstraction.
- Code is a more abstract specification of the required computation. Therefore, changes to the underlying runtime system library can be more easily introduced, since they are restricted to modification of the compilation process. This proved especially useful during the experimental phase in which the runtime system underwent frequent modification.
- It is still possible to program directly against the runtime library if so desired. In this case the SODA translation step can be skipped.

4.1.2 The System Implementation Language: Java

Since the SODA model is object-oriented, an object-oriented or object-based language is preferable for the system implementation. Also, support for heterogeneity is desirable, to make the system portable. For these reasons, we selected Java as the system implementation language and as the target for the SODA translator. The advantage of Java is that the language features are rather streamlined and that object-orientation is followed throughout. In contrast, languages that include non-object oriented features, such as C++, pose difficulties for the SODA model. For example, features such as global variables and pointers could not be supported in SODA.

It is true that the performance of Java has not been very good in the past. In HPC, performance is by definition of paramount importance. At a first glance, the combination of Java and HPC therefore appears to be an oxymoron. However, it must be noted that execution-inefficiency is a property of the language implementation, not of the language per se. Java programs are compiled into *byte-code* for a *Java Virtual Machine* (JVM) [104]. On most microprocessors, the JVM is emulated in software, which introduces additional run-time costs compared to native execution. Other costs are incurred by the JVM's automatic memory management. Most authors in [1] however contend that Java's weaknesses are surmountable and that it has a significant potential for HPC. They trace Java's past efficiency problems to naïve implementations and illustrate several HPC applications. Java is a promising vehicle for HPC due to a unique set of properties (see also [1;94] for a detailed discussion):

- Java byte-code is secure and portable between heterogeneous platforms, removing the need to explicitly compile to different target architectures.[21]
- Java has built-in thread support and low-level synchronisation primitives, which yield portable results across all supported platforms; multiprocessor systems can be utilised effectively.
- Strong static typing is guaranteed by the Java compiler and also required by the byte-code verifier built into the JVM before code is accepted for execution.
- Automatic memory management with asynchronous garbage collection.
- Finally, Java programs are much easier, faster and safer to produce than their C/C++ counterparts. One example is the removal of pointer manipulations and explicit memory allocation, a source of many subtle bugs in C/C++ programs, such as buffer overflows.
- Programmers have ready access to an extensive set of libraries that exist for graphics and networking.

We expect to ride the Java technology curve, reaping better performance in SODA as the Java technology platform matures. Improved compilers (especially JIT and native compilers) have already begun to remove Java's performance deficit [94]. JIT compilers perform a partial compilation of Java byte-code into native code for frequently called methods (hotspots) of the program. Native compilers go further and generate native executables ahead-of-time at the cost of giving up Java's compile once, run anywhere, advantage. Examples are the Free Software Foundation's GNU-GCJ compiler [66], TowerJ VM [170], Excelsior JET [57]. These systems usually provide a built-in Java interpreter in order to support dynamic class loading. Unfortunately, the performance advantage does not extend to such dynamically loaded classes.

Another requirement for efficient HPC is the support for high-performance communications. To this end, Java version 1.4 offers a complete reimplementation of the I/O package, with support for socket channels. This allows asynchronous, non-blocking network communication (NIO), greatly improving networking scalability [87].

Amendments to the Java language have been proposed that would allow run-time performance close to FORTRAN for numerically intensive applications [122;123]. For a few years, the Java Grande Forum [88;160;168] has been representing the

[21] This is necessary, e.g., in MPI and PVM if programs are to be run on a heterogeneous cluster. A different executable for every architecture is then required.

interest of HPC programmers for the development of future versions of Java. They have succeeded in adding improved floating-point support into Java 1.2 and continue to provide further proposals for improving Java's efficiency for numerical operations. Due to the activities of the Java Grande Forum and other groups more efficient JVMs can be expected in the future, leveraging Java's benefits without significant performance costs. While some JVM improvements require the rewriting of programs to fully exploit their benefits, others will transparently leverage existing code. Other projects which examine Java for HPC are [40] and [64;70]. Welsh and Culler [181] propose modifications to the JVM and native compilation to interface Java with high-performance networking interfaces and to improve communication between distributed JVMs. Hyde [85] lists a set of parallel programming models which can be implemented on top of Java.

4.1.3 The SODA Abstract Machine

The SODA model has been integrated into a portable, efficient and flexible middleware infrastructure, implemented as a Java library. This runtime system library allows to write object-oriented, parallel and distributed applications and to implement higher-level programming systems. SODA applications are executed by a SODA abstract machine (SAM) built on top of a variable collection of heterogeneous computers communicating by means of a multi-protocol transport layer. The SAM appears as a logically fully-interconnected set of *bases*, each one wrapping a Java Virtual Machine. Each physical computer may host more than one base. This is useful for testing purposes, if a cluster is not available.

Externally, the physical distribution of active objects is transparent to the programmer–the SAM provides a global namespace for active objects across the cluster and automatically mediates remote method invocations if client and server objects are not collocated on the same base. The transport layer attaches a globally unique VMID[22] identifier to every participating base in the system. An active object is uniquely referenced through a combination of VMID and object identifier OID.

On every base, the SAM efficiently schedules active objects and multiplexes them onto available threads in order to match potential and physical parallelism. Lightweight active objects are provided through two mechanisms: Optimised stack/heap-based SODA method invocation (§4.3.4-§4.3.5) and multiplexing of active objects (§4.4).

The SAM is completely decentralised. No central component, such as a naming service, exists. All bases work in symmetric peer-to-peer fashion, which avoids centralised scalability bottlenecks. The absence of any central points of failure could also provide a strong basis for a future fault-tolerant version of SODA. For example, active objects could be replicated on different SODA bases. However, no such fault-tolerance is provided in the current version of SODA.

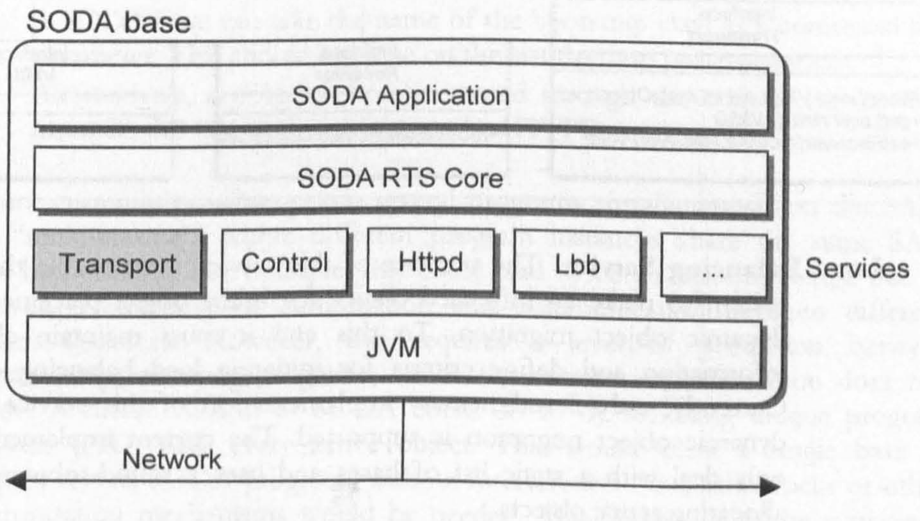
[22] The implementation of VMID depends on the transport implementation in use. As an example, in the TCP/IP transport implementation, a VMID comprises an IP address and a port number.

4.1.4 Object Factories and Bases

The SAM enables the location-transparent invocation of active object methods. However, we also need a mechanism for the dynamic creation of active objects. For such bootstrap purposes, every base provides a factory object, which is an active object itself. The factory method for creating new active object instances has the signature `Future makeNew(ConstructorCall)`. A `ConstructorCall` is manufactured through one of a set of static, overloaded methods in the proxy's interface, which take the place of constructors. These constructors do not follow the pattern of conventional Java constructors. The exact semantics and usage of such constructors is explained in §4.2.4 below.

A program can explicitly request a list of all factories through the call `RTS.getAllBases()`. However, this makes the physical distribution explicit to the application level. Therefore, a user program should employ the load balancing service (see below) to determine a new active object's allocation. This is achieved through the call `RTS.newActive(ConstructorCall)`. The load balancing service will then select an appropriate base according system load and allocation policy.

Figure 4-1 SODA Runtime System Structure

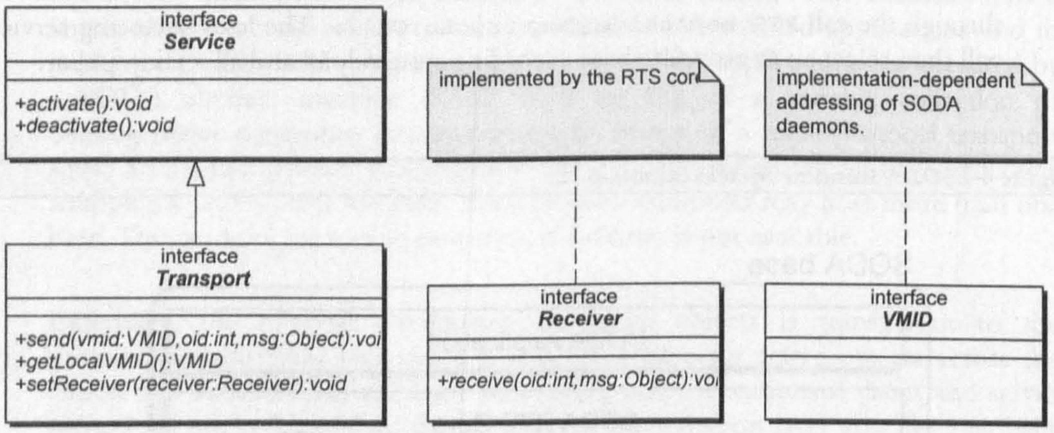


4.1.5 Plug-in Services

The SAM is built in a modular way using *plug-in services* for the implementation of various subcomponents. Services are defined in terms of their interface and can easily be replaced by alternative implementations. This allows an easy experimentation and quick evaluation of different strategies. Each service implementation resides in a separate package and is instantiated through a *factory* class (see [152]). These services are now described.

Transport Service. The transport service implements inter-base communication. It is responsible for the set up and management of connections. Three implementations of the transport service currently exist: they are using TCP/IP sockets, unicast datagrams and Java RMI as a transport protocol, respectively. Other implementations could use CORBA, MPI or other any other protocol for the network communication, possibly even using native code to interoperate with high-performance network interfaces (such as ATM or VIA). Most important is the method `send`, which allows the transfer of an object `msg` to a remote active object identified by `<VMID:OID>`. At the receiver side, the object msg is handed over to a `Receiver` object together with the destination active object's local `OID`. The `Receiver` is then responsible to despatch `msg` to the correct active object. The type of `msg` is either `Request` or `Reply`.

Figure 4-2 Transport Service Interface Class Diagram



interface
Service
+activate():void
+deactivate():void
implemented by the RTS core
implementation-dependent addressing of SODA daemons.
interface
Transport
+send(vmid:VMID,oid:int,msg:Object):voi
+getLocalVMID():VMID
+setReceiver(receiver:Receiver):void
interface
Receiver
+receive(oid:int,msg:Object):voi
interface
VMID

Load Balancing Service. The purpose of this service is to enable the run-time system to make an informed choice for initial object placement and for dynamic object migration. To this end it must maintain cluster load information and define criteria for initiating load balancing operations. Currently, only a rudimentary implementation of this service exists; no dynamic object migration is supported. The current implementation can only deal with a static list of bases and uses a round-robin strategy for allocating active objects.

Control Service. This service allows external control of the SAM, e.g., through an applet downloaded via the HTTPD service. This can be used to monitor the available bases and to control program execution. Currently, only monitoring functionality is provided and the standard output stream of the base can be examined. Future implementations could provide the ability to select a subset of available hosts for the execution of a program and support for several concurrent programs (see §4.1.6).

HTTPD Service. This is used to provide an administrative interface to the SODA runtime system via a conventional web browser. An embedded Java applet communicates with the control service and allows the user to perform administrative and auditing functions via the control service. The administrative interface could be improved to include the deployment of JAR files over all participating machines in the SAM.

FastSerialisation. This is not a service per se, but rather a set of helper classes used by some implementations of the transport service. It provides an effective means of serializing Objects into byte arrays and therefore reducing overheads when transferring rich data types across the network (see §4.5.3).

4.1.6 Program Execution

Execution of a SODA program starts with a single active object that has a method public `void main(String[] args)`. This is the primordial detached method (see 3.3.2). This is not a class method as is the case in Java. Instead, the SVM will first create an active object instance of the bootstrap class and then invoke its main as an instance method. Since `main` does not return a Future, it is treated as a detached method (see §4.4.3). The main method can then spawn secondary active objects which are allocated across the set of available SODA daemons. The allocation policy is defined by the load balancing service. The primordial method defines the lifecycle for a SODA program. Two options exist for the start-up of main:

- A SODA base can take the name of the bootstrap class as a command line parameter. This should be done on the last daemon process started.
- Alternatively, a program can be started through the control service. For instance, this can be done through the HTTPD service.

It is not currently possible to run several programs simultaneously on the SAM. Such "multi-tasking", where different program instances share the same SAM would be desirable. For example, this could lead to better resource usage due to improved overlapping of computation and communication between different program instances. However, this requires a level of protection between simultaneously executing programs, which the current implementation does not provide. A future version of SODA could do so by including unique program identifiers (PID) with every active object. This would allow a single base to multiplex several SODA programs without interference. No further locks or other synchronisation mechanisms would be needed. This feature could be supported without change to existing programs.

4.2 SODA Source-to-Source Translator

SODA programs are converted into Java source code through the SODA translator; in a second stage, the resulting Java source code can then be compiled into bytecode using a standard Java compiler. The source-to-source translator has been written with the JavaCC compiler-generating tool [179]. This makes SODA independent of Java compilers, in the same way that it is independent of Java Virtual Machines.

The result of the translation process is a collection of Java classes for every SODA active object declaration. These include a `Proxy` and a `Body` class and meta-classes for every active object constructor and every active object method. A Proxy *appears* like an instance of an active object: their class name matches that of the active object declaration and the Proxy replicates the active object interface. The Proxy is the only object that is directly accessible to application code. All other generated Body classes and meta-classes are only used internally by the SODA runtime system.

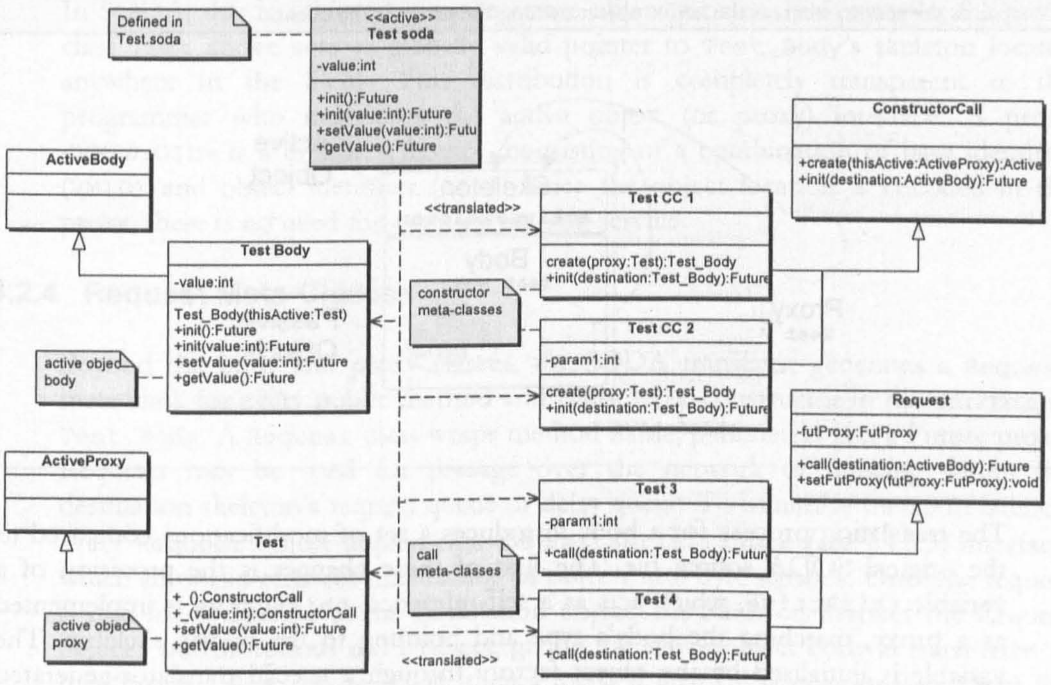
4.2.1 Simple Translation Example

As an example, consider Figure 4-3 below. The SODA translator converts an active object definition `Test.soda` into a set of Java classes. The result comprises the proxy class, `Test.java` and the body class `Test_Body.java`. The body class contains only slight modifications compared to the original `Test.soda` file.

The class name and method interface of the proxy class definition, `Test.java`, match that of the active object class definition in `Test.soda`. The Test proxy can therefore be used as drop-in replacement for the active object. All classes in a system can use the `Test` class, as if it were an active object with an interface as defined in `Test.soda`. For example, passive objects can create and use active objects through Proxy classes, without requiring any translation step. Of course, `Test.java` must be generated and compiled before any code depending on that class can be successfully linked.

In addition to proxy and body, a set of call meta-classes and constructor meta-classes come into existence. There is one meta-class for all every method and every constructor that is defined in the public interface of `Test.soda`. These meta-classes are used internally by the SODA runtime system for network transmission and queuing purposes.

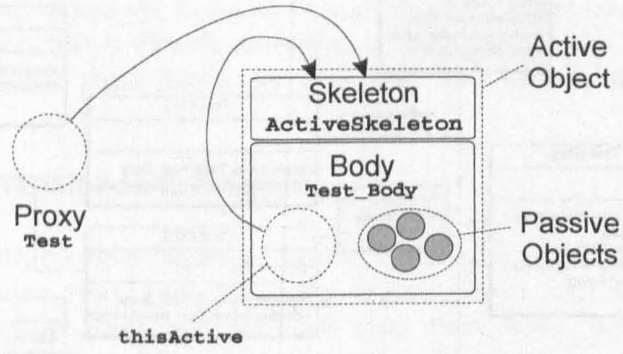
Figure 4-3 Meta-class Generation by the SODA Translator


Defined in Test.soda
<<active>>
Test soda
-value:int
+init():Future
+init(value:int):Future
+setValue(value:int):Futu
+getValue():Future
ConstructorCall
+create(thisActive:ActiveProxy):Active
+init(destination:ActiveBody):Future
ActiveBody
<<translated>>
Test CC 1
create(proxy:Test):Test_Body
+init(destination:Test_Body):Future
Test Body
-value:int
Test_Body(thisActive:Test)
+init():Future
+init(value:int):Future
+setValue(value:int):Future
+getValue():Future
constructor meta-classes
Test CC 2
-param1:int
create(proxy:Test):Test_Body
+init(destination:Test_Body):Future
active objec body
Request
-futProxy:FutProxy
+call(destination:ActiveBody):Future
+setFutProxy(futProxy:FutProxy):void
ActiveProxy
Test 3
-param1:int
+call(destination:Test_Body):Futur
Test
+_():ConstructorCall
+_(value:int):ConstructorCall
+setValue(value:int):Future
+getValue():Future
call meta-classes
Test 4
+call(destination:Test_Body):Futur
active objec proxy
<<translated>>

4.2.2 Skeletons and Bodies

At runtime, an active object instance is comprised of a body and a skeleton. The body in our example is `Test_Body`. The skeleton class `ActiveSkeleton` is generic and performs various maintenance activities on the skeleton's behalf. This includes provision of request queue, delay queue and reply queue. Also, scheduling and synchronisation is the responsibility of the skeleton. The skeleton's activities and its interplay with the body are described in detail in §4.4. All application access to an active object is through its proxy. Passive objects are contained within the body, as specified in the `Test.soda` file. They are owned by the `Test` active object.

Figure 4-4 Relation between Proxy, Skeleton and Body.
`Test_Body` is the "owning active object" for the passive objects; `thisActive` is a self-reference proxy to the active object, with type `Test`. In a method invocation, the skeleton-body pair is the server, whereas the active object owning the proxy is the client.



The translation process for a body introduces a set of modifications compared to the original SODA source file. The first of these changes is the provision of a variable `thisActive`, which acts as a self-reference. `thisActive` is implemented as a proxy, matching the body's type and pointing to the body's skeleton. The variable is initialised by the object factory through a special translator-generated Java constructor in the body. For example, the body `Test_Body` contains the constructor `Test_Body(Test thisActive)` for this purpose.

The only other modification in `Test_Body.java` compared to `Test.soda` is related to method guards. The translator extends every method signature with a `GuardException` throw-clause. This is important for SODA's implementation of Ferenczi guarded methods (see §4.4.2). In addition, if a method in `Test.soda` defines a guard, evaluation of this guard is inserted as the method's first statement in `Test_Body.java` (see Code 4-1 and Code 4-2). SODA's `init`-style active object constructors are adopted without modification from `Test.soda`.

Code 4-1 SODA Guard Definition (as in e.g., Test.soda)

```
public Future method ({params}*) when (cond) {
    //method body
}
```

Code 4-2 Guard Definition translated into Java (as in e.g., Test_Body.java)

```
public Future method ({params}*) throws GuardException {
    if (!cond) throw guardException;
    //method body
}
```

4.2.3 Active Object Proxies

The SODA programming model requires the ability to dynamically create active objects and pass them by reference to other active objects. This is useful to create complex, distributed data structures and to write programs that operate on these structures in a homogeneous, uniform fashion, independent of object location.

In SODA, this is achieved through active object proxies. For example, the proxy class `Test` above acts as globally valid pointer to `Test_Body`'s skeleton located anywhere in the SVM. This distribution is completely transparent to the programmer who only uses the active object (or proxy) interface. A proxy `<VMID:OID>` is a unique reference, consisting of a combination of base identifier (`VMID`) and object identifier (`OID`). Since the object location is encoded in the proxy, there is no need for a central naming service.

4.2.4 Request Meta-Classes

Beyond the body and proxy classes, the SODA translator generates a `Request` meta-class for every public method and every `init`-constructor in the interface of `Test.soda`. A `Request` class wraps method name, parameters and a Future proxy. Requests may be used for passage over the network or queuing within the destination skeleton's request queue or delay queue. To minimise transport latency, every `Request` object implements the `FastSerializable` (see §4.5.3) interface, which allows an efficient marshalling of objects into byte streams. Once the request is due for evaluation at the destination object, the Skeleton invokes the Request object's polymorphous `call` object, giving the active object body as parameter. If this throws a `GuardException`, then a method guard was not satisfied. Once the request completes successfully, the skeleton will wrap the resulting Future into a reply. It then sends the reply to the Future given in the request's Future proxy.

4.2.5 Dynamic Object Creation

Conventional Java constructors do not have return parameters per se. The result, instead, is a reference to the defining class. If this concept is transferred to active objects, a problem emerges: no asynchronous subcalls are possible. This makes the usage of Funnels impossible, since they rely on Future type returns in the calling method. Without Funnels no asynchronous subcalls can be programmed.

Blocking subcalls would lead to deadlock whenever the constructor was to perform direct or indirect recursion, since SODA active objects are atomic. This approach is therefore inherently unsafe. We also reject the idea of simply disallowing subcalls in constructors, as this is an important concept required to set up complex data structures. While the task of constructors could be delegated into dedicated initialisation methods, there would be no way to enforce once-only semantics for the execution of these methods at object creation time.

The approach adopted in SODA abandons the conventional Java constructor syntax in favour of `init` methods. A set of overloaded `init`-methods behaves exactly like constructors, except that they return `Future` type results. Therefore, asynchronous subcalls are possible, as for any other SODA method. The `init` methods are not directly exposed at the proxy's interface. Instead, the execution time of `init` methods is managed by the object factory from which the active object is created. This allows SODA to enforce the execution of exactly one `init` method immediately after active object allocation time.

As an example, consider our example class `Test.soda` with overloaded constructors `init(int a)` and `init(double a)`. The SODA translator maps these onto corresponding static methods in the proxy class `Test` with signatures `ConstructorCall _(int a)` and `ConstructorCall _(double a)`. An instance of `Test` is created through `RTS.newActive(Test._(5))` or `RTS.newActive(Test._(5.0))`. As a result, a `ConstructorCall` object of type `Test_CC_1` or `Test_CC_2` is created, respectively. As a result, the object factory of the destination base performs the following actions:

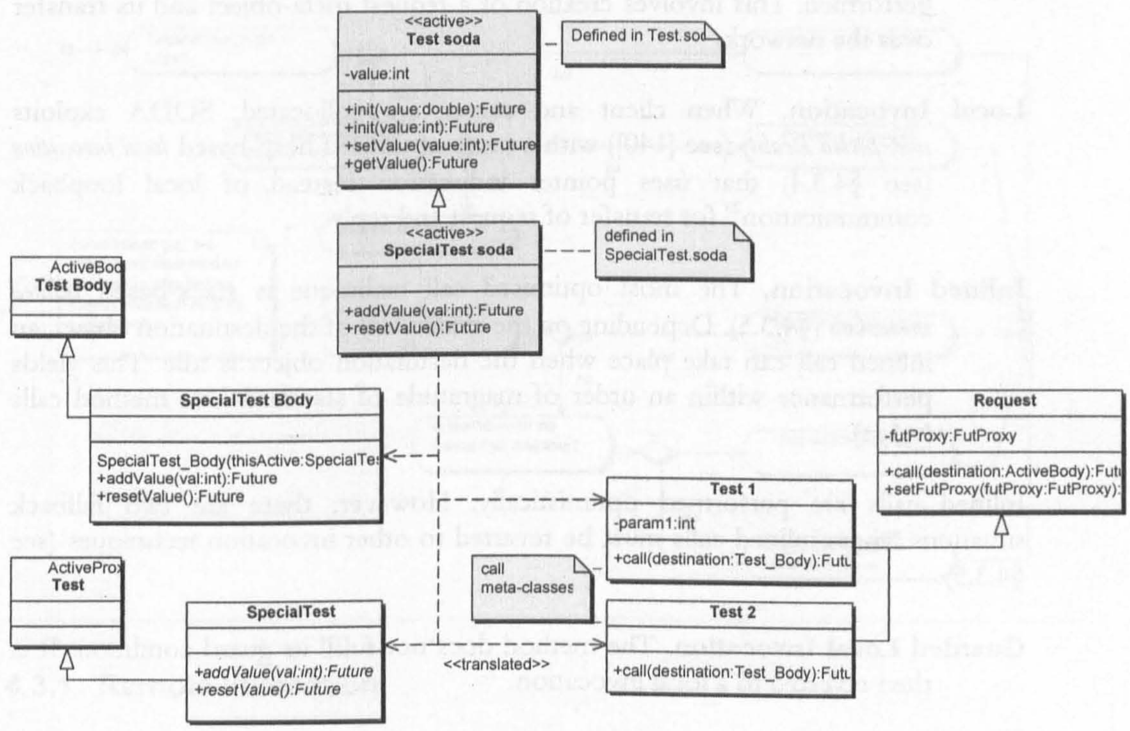
- Create a new active object skeleton and a generic proxy p to that skeleton
- Call `ConstructorCall`'s create method, which will initialise a new active object body using the translator-generated constructor `Test_Body(ActiveProxy p)`. This initialises the `thisActive` variable in the body with the proxy p obtained above.
- Wrap the body in the previously created skeleton.
- Call `ConstructorCall`'s init method, which invokes one of the overloaded init methods in the active object body as appropriate.
- Set up a Funnel on termination of this `init` method.
- The Funnel sets the result to the newly created proxy and catches any exceptions from `init`, before handing back a reply to the original client.

4.2.6 Dealing with Inheritance

The SODA compiler fully supports dynamic binding and inheritance of active objects, which are important concepts in object-oriented programming. As an example, consider an active object class `SpecialTest` which inherits from Test. In this case, the proxy class `SpecialTest` inherits from `Test`, while the body class `SpecialTest_Body` inherits from `Test_Body`. Therefore, proxies of type `Test` can be used to invoke methods on `SpecialTest` active objects. However, `SpecialTest's` additional functionality is only available through the `SpecialTest` proxy. Casting between `Test` and `SpecialTest` is supported.

One particular feature of SODA's inheritance support is that it circumvents the inheritance anomaly. The approach we take is based on Ferenczi's proposal for the semantics of guarded methods (see §3.6 for a detailed description of the mechanism).

Figure 4-5 Inheritance Support in the SODA Compiler


<<active>>
Test soda
-value:int
+init(value:double):Future
+init(value:int):Future
+setValue(value:int):Future
+getValue():Future
Defined in Test.soda
<<active>>
SpecialTest soda
+addValue(val:int):Future
+resetValue():Future
defined in SpecialTest.soda
ActiveBod
Test Body
SpecialTest Body
SpecialTest_Body(thisActive:SpecialTes
+addValue(val:int):Future
+resetValue():Future
Request
-futProxy:FutProxy
+call(destination:ActiveBody):Futi
+setFutProxy(futProxy:FutProxy):
Test 1
-param1:int
+call(destination:Test_Body):Futu
call meta-classes
Test 2
+call(destination:Test_Body):Futu
ActiveProx
Test
SpecialTest
+addValue(val:int):Future
+resetValue():Future
<<translated>>

4.3 Method Invocation in SODA

In this section, we focus on active object calls from the client's point of view. Essentially, this is a discussion of the proxy's algorithms. Section 4.4 discusses the implementation of the active object skeleton, which represents the server-side of a call.

In the SODA runtime system, method invocation is the only communication path between active objects. Method invocation is always initiated through a proxy, which forwards the call to the active object body. The proxy is transparent, since its interface matches the active object's interface as defined in the SODA source files. This also enables location transparency for active objects, since only the proxy need be concerned about distribution. The proxy further guarantees SODA parameter-passing semantics by preventing a client from handing out private objects to a server by enforcing by-copy semantics. The execution model must provide this guarantee, regardless of whether or not proxy and body (client and server) reside in the same physical address space (same JVM). This allows uniform, homogeneous access to local and remote objects, transparently to the programmer.

A proxy can use one of three different invocation techniques. Each proxy contains an algorithm to choose the most appropriate technique according to current runtime conditions. We discuss these invocation techniques, from the most general to the most efficient and we give an approximate cost measure for empty method calls on our test platform.

Remote Invocation. In the most general (and least efficient) case, where client and server reside on distinct machines, a *remote invocation* (see §4.3.1) is performed. This involves creation of a request meta-object and its transfer over the network.

Local Invocation. When client and server are collocated, SODA exploits *unexpected locality* (see [140]) with a more optimised heap-based *local invocation* (see §4.3.4) that uses pointer indirection instead of local loopback communication[23] for transfer of request and reply.

Inlined Invocation. The most optimised call technique is stack-based *inlined invocation* (§4.3.5). Depending on the availability of the destination object, an inlined call can take place when the destination object is idle. This yields performance within an order of magnitude of standard Java method calls (~1μs).

Inlined calls are performed optimistically. However, there are two fallback situations where inlined calls must be reverted to other invocation techniques (see §4.3.5):

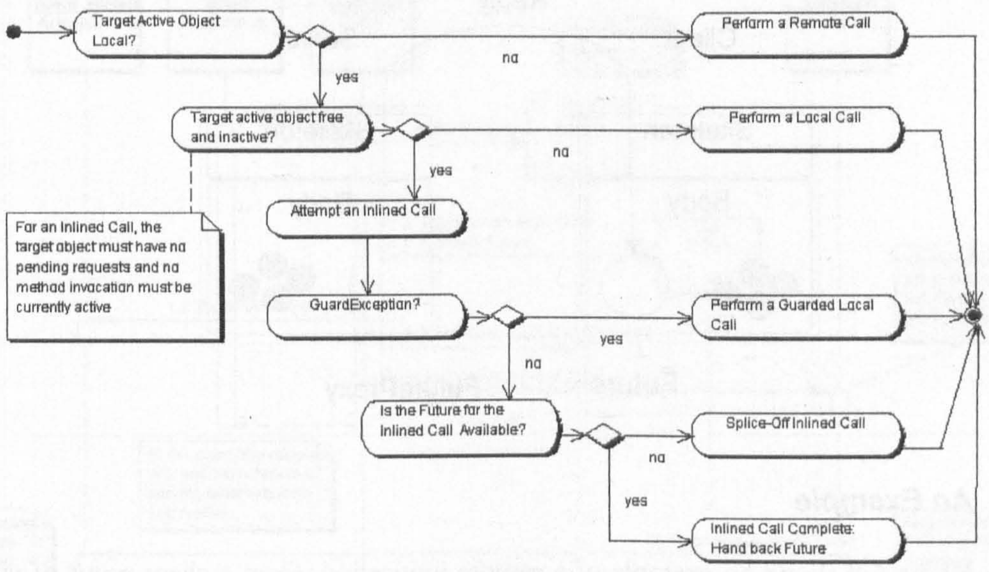
Guarded Local Invocation. The method does not fulfil its guard condition. It is then reverted to a local invocation.

Splice-Off Inlined Invocation. When an inlined subcall is blocking there is a risk of *parent-child welding* [54]. In this situation, SODA "*splices off*" the invocation to proceed asynchronously and avoid deadlock.

In the following sections, these invocation techniques are explained in detail, interspersed with the description of client- and server-side Futures.

[23] As Java RMI does for calls to a collocated RMI object. Pointer indirection is several orders of magnitude faster than local loopback communication.

Figure 4-6 Activity Diagram for an Active Object Method Invocation.


Target Active Object Local?
no
yes
Perform a Remote Call
Target active object free and inactive?
no
yes
Perform a Local Call
For an Inlined Call, the target object must have no pending requests and no method invocation must be currently active
Attempt an Inlined Call
GuardException?
yes
Perform a Guarded Local Call
no
Is the Future for the Inlined Call Available?
no
Splice-Off Inlined Call
yes
Inlined Call Complete: Hand back Future

4.3.1 Remote Invocation

For distributed active objects, the method invocation, along with its parameters and results, is transferred across the network, to and from the serving object on a remote JVM. The method call is performed in two phases: first the proxy creates a pending Future in the client and transmits a request to the remote JVM where the server resides (Figure 4-7). In the second phase, the destination object on the remote JVM turns back a reply containing result and exception of the terminated call (Figure 4-8). The transport service handles the transfer of `Request` and `Reply` objects between different JVMs.

Figure 4-7 Request path for method execution from client to server active object

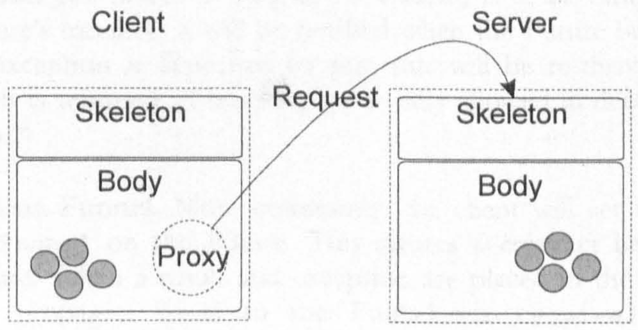
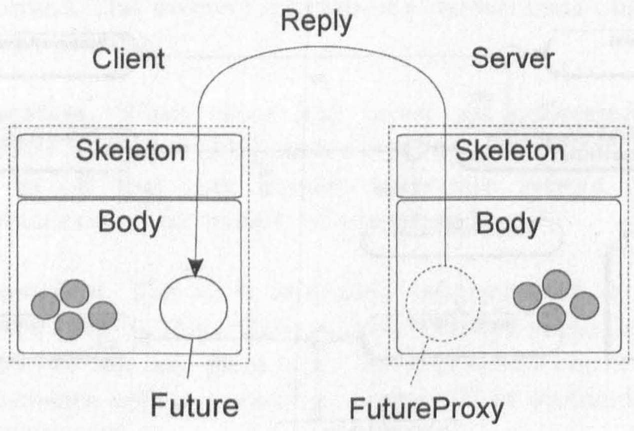


Figure 4-8 Reply from server to client after method termination

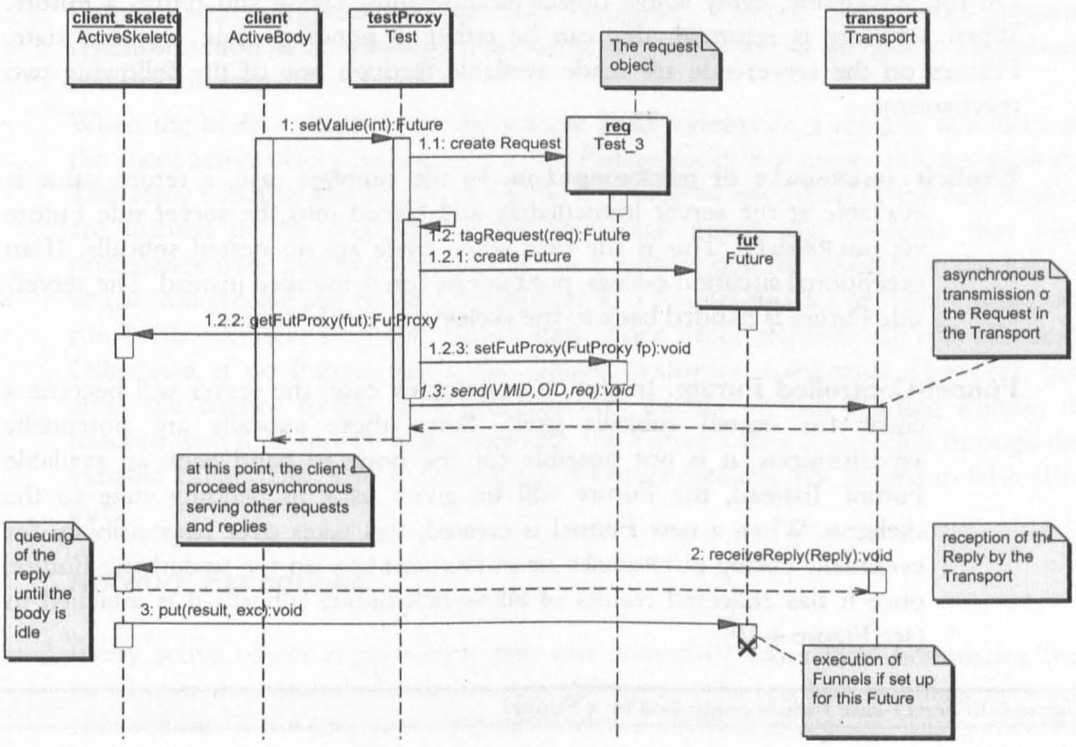


An Example

Figure 4-9 shows an example of a remote invocation, from a client point of view. The client has a proxy `testProxy` that references a remote active object `Test`. When a method is invoked, the proxy creates a `Request` object `req` (sequence ID 1.1), as an instance of the meta-class `Test_3`. In addition, it sets up a pending `Future` within the client object and a corresponding Future proxy `fp` (ID 1.2). The Future proxy `fp` is included in the request (ID 1.2.3), and then the request is dispatched via the `Transport` service (ID 1.3). At this stage the client can proceed asynchronously past the method call.

Eventually, a `Reply` is received back from the server (ID 2) and added to the client's reply queue. The evaluation of the reply is serialized in the client active object's activity stream (i.e., in exclusion with the processing of other requests or replies). When the body becomes available, the skeleton `client_skeleton` removes the reply from the reply queue and deposits the result and possible exception of the call to the client-side Future using the `put` method (ID 3).

Figure 4-9 Client-side view of a remote invocation


client_skeleto
ActiveSkeleto
client
ActiveBody
testProxy
Test
The request object
transport
Transport
1: setValue(int):Future
1.1: create Request
req
Test_3
1.2: tagRequest(req):Future
1.2.1: create Future
fut
Future
asynchronous transmission o the Request in the Transport
1.2.2: getFutProxy(fut):FutProxy
1.2.3: setFutProxy(FutProxy fp):void
1.3: send(VMID,OID,req):void
at this point, the client proceed asynchronous serving other requests and replies
queuing of the reply until the body is idle
2: receiveReply(Reply):void
reception of the Reply by the Transport
3: put(result, exc):void
execution of Funnels if set up for this Future

4.3.2 Implementation of Futures

Client-Side Usage

Futures allow asynchronous local and remote calls. While the proxy creates a pending Future and hands it back to the client, a request is queued for later execution at the server's skeleton. The client can deal with a Future in one of two ways: either it queries for a result in blocking fashion, using `get` or it sets a Funnel.

Blocking Get. `get` blocks as long as `isPending` is `true`. Since `get` waits on the Future's monitor, it will be notified when the Future becomes available. If an exception is deposited by put, this will be re-thrown. Otherwise, the result is returned. A blocking get is only allowed in detached methods (see §3.3.2).

Asynchronous Funnel. More commonly, the client will set up a Funnel, using `setFunnel` on the Future. This creates a contract between Funnel and Future. When a result and exception are placed in the Future, the Future will propagate these to the Funnel via `receive(Object result, Exception exc)`. Every Funnel can collect a set of pending subcall Futures in this data-driven manner.

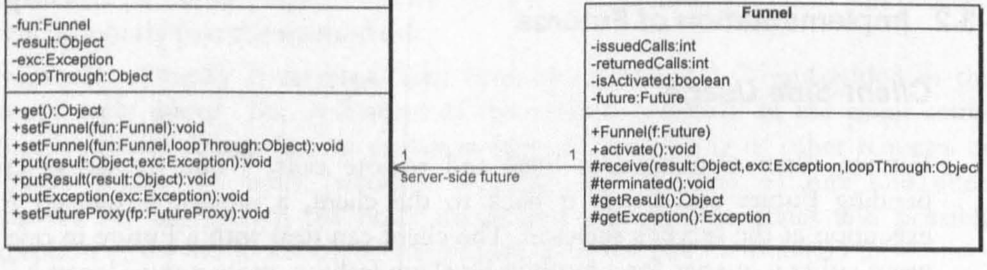
Server-Side Usage

On the server-side, every active object method must create and return a Future. When a Future is returned, this can be either in pending or in available state. Futures on the server-side are made available through one of the following two mechanisms.

Explicit `putResult` or `putException`. In the simplest case, a return value is available at the server immediately and placed into the server-side Future via `putResult`. This is the case when there are no nested subcalls. If an exceptional situation occurs, `putException` is invoked instead. The server-side Future is handed back to the skeleton in available state.

Funnel-Controlled Future. In the more complex case, the server will become a client for nested subcalls itself. Since these subcalls are potentially asynchronous, it is not possible for the body to hand back an available Future. Instead, the Future will be given back in pending state to the skeleton. When a new Funnel is created, this takes over responsibility for eventually calling `putResult` or `putException` on the underlying Future, once it has collected results of all asynchronous subcalls it is attached to (see Figure 4-10).

Figure 4-10 Server-side Future controlled by a Funnel


Future
-fun:Funnel
-result:Object
-exc:Exception
-loopThrough:Object
+get():Object
+setFunnel(fun:Funnel):void
+setFunnel(fun:Funnel,loopThrough:Object):void
+put(result:Object,exc:Exception):void
+putResult(result:Object):void
+putException(exc:Exception):void
+setFutureProxy(fp:FutureProxy):void
1
server-side future
1
Funnel
-issuedCalls:int
-returnedCalls:int
-isActivated:boolean
-future:Future
+Funnel(f:Future)
+activate():void
#receive(result:Object,exc:Exception,loopThrough:Objec
#terminated():void
#getResult():Object
#getException():Exception

Once the skeleton obtains the Future, it will call `setFutureProxy` on it, giving the Future proxy contained in the original request as argument. Only when the result (or exception) has been set on the server-side Future, the Future is converted into a reply object and sent back immediately. Otherwise, the Future remains at the skeleton until `putResult` or `putException` is invoked on it by the Funnel.

Futures Implementation

`Futures` cannot exist independently, but are always attached to a single "owning" active object. They behave like passive objects, except that they do not have "first-class object" status. The reason for this lies in the implementation of Future identities. Such identities, called *Future proxies*, are a combination of active object proxy and Future identifier `<FID>`. A Future proxy `<VMID:OID:FID>` uniquely identifies a Future within the owning active object `<VMID:OID>`. Since Futures do not have an identity outside of the context of their owning active object, this explains why they cannot be passed as parameters between active objects. If Futures were implemented as fully qualified objects, this would require synchronisation of distributed Future instances, which would be hard to implement in a scalable way.

When the body invocation on the remote JVM terminates, a reply is sent back to the client active object (as encoded in the Future proxy that came with the request). This reply contains the `<FID>`, as well as the result or exception of the call. Replies are queued in the reply queue for sequential processing, however they take precedence over pending requests. When a reply is due for evaluation, the Future `<FID>` will receive any result or exception. If a Funnel was set up, this Funnel will run (with exclusive access to object data, since other requests are still blocked). Otherwise, if no Funnel exists, the request evaluation terminates. However, this gives the chance to any detached methods waiting on the pending Future, to proceed with their `get()` calls. Internally, the Future's state is reflected through the variable `isPending`. This is set to `true` at Future creation, but flipped to false after `put`.

Network Exceptions

Every active object request-reply pair can potentially cross JVM boundaries and travel over the network. In this case, problems may be sparked by failing network connections. If the transport layer encounters a network exception, this is encoded into the client's Future and re-thrown on access to the Future. The client can then take appropriate action, e.g., retrying, abandoning overall program execution or delegating the call to another server.

There is still an outstanding problem: if the server's JVM crashes after receiving the request, the Future would never receive an exception, since requests are one-way messages. In a future version of SODA, a timeout-mechanism should be integrated, which puts an exception in the Future of a call which does not return for a long time.

4.3.3 Funnel Implementation

A Funnel collects the client-side Futures of several asynchronous subcalls which a server performs and deposits them into a server-side Future. This server-side Future is given to the Funnel at instantiation time. Internally, a Funnel keeps track of the number of asynchronous calls that it set up through `setFunnel`. A Funnel can produce a result/exception for the server-side Future, as soon as the number of asynchronous calls made on it and the number of replies it received match; in addition, the Funnel must have been `activated`. Activation is necessary, since otherwise a subset of completed asynchronous calls may already fire the Funnel.

`setFunnel` accepts an optional parameter Object `loopThrough`. This is attached to the Future and later made available to the Funnel when the associated method returns. This is useful for distinguishing between a set of Futures that are controlled by a Funnel.

The default Funnel will issue a `null` result to the underlying server-side Future, as soon as it is activated and all asynchronous calls have been collected. This behaviour can be modified in Funnel subclasses by overriding `receive`, `terminated` and `getResult/getException` methods (see Figure 4-10 and §3.5.2):

- **`receive`** is called whenever a reply from an asynchronous subcall arrives. This could be used to program aggregate functions, e.g. summation of results. The `loopThrough` parameter is set to null if no `loopThrough` object was set up.
- **`terminated`** is called whenever the Funnel fires, e.g., when all results are available and the Funnel has been activated. By default, this is an empty method.
- **`getResult`** returns `null` by default, but can be programmed to provide a different result for the underlying Future.
- **`getException`** returns `null` by default, but can be programmed to provide a different result for the underlying Future.

4.3.4 Local Invocation

If both, client and server reside in the same physical address space (same JVM), no network communication is required and the above approach can be optimised. In this case, the proxy performs a local invocation, bypassing the transport service. Otherwise, the invocation sequence is identical to Figure 4-9. The client's proxy will invoke `receiveRequest` on the destination skeleton directly, which enqueues the request and then gives back control to the client. The server then handles the invocation asynchronously. Since only a simple pointer indirection occurs, local calls are faster than loop-back network calls. When the result of the method call becomes available, the server skeleton will notify the client skeleton through a call to `receiveReply`.

Parameter Cloning and Hand-Over Parameters

In a local or inlined call, client and server are located in the same address space. Therefore, the proxy must take care of creating deep copies of passive object parameters to enforce SODA parameter passing conventions. Similarly, results from the server are deep-copied before being wrapped in the reply. These deep copies are created by calling the `clone()` method of the relevant parameter.

Deep-copying is necessary, as otherwise both client and server may modify the same instance of a passive object. For example, if a parameter object is modified by the server, this modification would be visible to the client. This would obviously violate the SODA programming model which does not allow sharing of information between active objects. A problem here is that deep-copy operations can be very cost-intensive, depending on object size.

In the case of handover parameters (see §3.7) this copying overhead can be avoided. Consider a situation where a client creates a passive object for the sole purpose of using it as an argument for the server invocation. After the asynchronous method invocation such a parameter object is abandoned. In this situation inconsistencies through false sharing cannot occur since once the parameter has been transmitted to the server, it is not modified any more by the client. Changes by the server do not affect correctness, since the client does not have any interest in the parameter after passing it to the server. It is the programmer's responsibility not to use handover type parameters on the client past the method call. It would be good practice to invalidate the client's reference by setting it no null after the call.

Handover is specified on a per-parameter basis. Other optimisations are possible on a per-passive-class basis. For example, a passive object that never changes its state does not need to be copied; neither client nor server can change such an immutable object. This is detailed in §4.5.

In a remote call, an accidental sharing of private data is largely evaded, since client and server reside on physically disjoint memory areas. Still, there is a risk, that parameters are modified *after* the request in which they are contained is handed over to the transport service, but before the transport service had a chance to actually serialize and despatch their internal data representations over the network. It is the responsibility of the transport service implementation to protect data from modification after it has been passed to the transport service. This guarantee must be observed when designing a transport service implementation that maintains a queue of outgoing messages for batched, asynchronous transmission in larger network packets.

4.3.5 Inlined Invocation

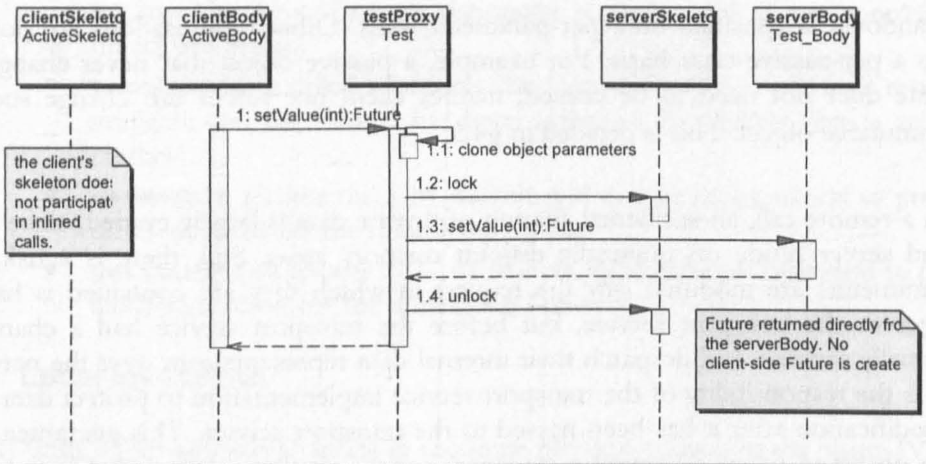
An active object is *idle* if its request queue, reply queue and delay queue are all empty. In this situation, the first arriving request can be executed immediately without queuing. SODA exploits this situation with a third invocation mechanism: the so-called *inlined invocation* bypasses queuing algorithms for idle server objects. SODA aims to perform every invocation between collocated client and server as an inlined execution as this is the most efficient of the three invocation mechanisms. Inlined calls are a mainstay of SODA's lightweight active object support, since they greatly reduce the method call overhead compared to queued local invocations. Overhead is within an order of magnitude of conventional Java method calls. Therefore, inlined invocation encourages a developer to utilise a large number of active objects in a program.

In an inlined invocation the proxy performs execution of the request directly and synchronously on the destination body (Figure 4-11). This stack-based invocation on the client thread appears like an asynchronous invocation to a programmer. The semantics are indistinguishable from queued, local method calls. This condition is based on the premise that SODA Future calls are non-blocking.[24] An inlined invocation does not rely on request or reply objects and bypasses the server's skeleton. Instead, the proxy delegates a call directly to the body. For this purpose, the proxy contains code for direct access to a body of matching type. This code is compiled into the proxy during the Soda-to-Java translation phase. In remote and local calls, two Future instances exist on the client and server side, respectively. For

[24] Only detached methods (see §4.4.3) may be blocking.

inlined calls, no client-side Future exists; instead, the body's Future is directly handed back to the client.

Figure 4-11 Sequence diagram for an inlined invocation


clientSkeleto
ActiveSkeleto
clientBody
ActiveBody
testProxy
Test
serverSkeleto
serverBody
Test_Body
1: setValue(int):Future
1.1: clone object parameters
the client's skeleton doe: not participat in inlined calls.
1.2: lock
1.3: setValue(int):Future
1.4: unlock
Future returned directly fro the serverBody. No client-side Future is create

Additional costs for inlined compared to standard Java method calls are caused by two factors:

- As for the local invocation mechanism, non-primitive parameters must be deep-copied to guarantee SODA call semantics for passive, mutable objects (v, sequence ID 1.1). For this purpose, the proxy applies `clone` on every non-primitive parameter as well as on the result.
- Thread synchronisation is required to prevent concurrent client access to a body. The body's skeleton serves as synchronisation monitor (Figure 4-11, sequence ID 1.2 and 1.4). Future objects are also synchronised to control access by client (put result/exception) and server (obtain result/exception).

Preconditions for an Inlined Call

In summary, an inlined call can take place if *all* the following preconditions are fulfilled:

- **Client and server objects are collocated** on the same base. Otherwise a remote call is performed.
- The **server object is not busy** (e.g., not currently evaluating another request or a reply). This is guaranteed by acquiring a monitor on the server's skeleton and locking the busy variable for the duration of the inlined call (see also §4.4.1).
- The **server has no requests in its delay queue**. This is necessary to guarantee the processing of the delay queue after every state change, which is not triggered by inlined calls.
- **No additional concurrency is needed** on the base. e.g., on a multiprocessor system, all processors are busy. If base-local parallel

slackness has to be increased, inlined calls revert to locally queued invocation technique in order to employ more worker threads on the local base (see also §4.4).

When the above preconditions are fulfilled, an inlined call is performed optimistically. However, there are two circumstances where a fallback technique is necessary.

Fallback to Guarded Local Invocation

The first situation relates to method guards. If a method guard evaluates to false the inlined invocation reverts to a **guarded local invocation**. While the monitor on the skeleton is maintained, the request is added directly to the server's delay queue. This is possible, because guard expressions appear as the first statement in a method and are side-effect free. Therefore, the method can be rescheduled without inconsistencies.

Fallback to Spliced-Off Invocation

The second situation occurs when the server itself becomes a client for a set of nested subcalls. This is fine as long as all subcalls are performed as **synchronous** inlined calls. However, if at one point in the invocation chain a subcall is reverted to an **asynchronous** (local or remote) call, the server hands back a *pending* Future to the client. Without any further precautions, a set of problems would arise:

- The server could asynchronously put a result or exception in the Future that is shared between client and server. If a Funnel is set up on this Future, it would then execute without serialization on the client's other activities. This is a violation of the SODA programming model that guarantees mutual exclusion of request and reply processing at active objects.
- Neither server nor client could be migrated until the asynchronous call terminated, since they share the same Future instance. If the client was moved, the server would not be able to notify it about the Future's availability. If the server was moved, the link is severed in a similar way.
- There would be a conflict between setup of a server-side Funnel and client-side Funnel on the same shared Future instance.
- In a stack-based inlined call, a blocking server would result in a blocking client. This is an instance of *parent-child welding* [54]. In SODA, active objects are atomic. Therefore, the client would not be able to accept any further requests, with a deadlock as likely outcome.

To avoid these problems, SODA adopts the following solution: When an inlined call returns with a pending Future, this is not directly given back to the client. Instead, the proxy–having access to the client skeleton–creates a new client-side Future and sets the corresponding Future proxy on the existing (server-side) Future via `setFutureProxy`. In this way, an inlined invocation reverts to an asynchronous reply. This is propagated up towards the root of the invocation chain and affects all Futures from the original inlined caller down to the asynchronous caller.

SODA splice-off calls allow a decoupling of the client and reversion of an attempted inlined call into a heap-based call. Parent-child welding (see §2.3.1) therefore does not affect SODA. Splice-off calls are similar to the optimised heap/stack invocation scheme implemented in ABCL1/AP-1000 [167]. The difference is that their technique reschedules a call into the mail queue and is therefore likely less efficient.

Limitations and Trade-Offs

Inlined calls are processed in an expedited fashion. This increases efficiency, but it also introduces a problem of fairness. While requests and replies are usually scheduled in order of their arrival across the base, inlined calls take precedence. i.e., an object which satisfies the conditions for inlined calls is taking precedence over other active object invocations on this base.

This trade-off of efficiency against fairness is accepted, since the SODA programming model does not make any fairness guarantees towards the order in which active objects receive processing power. FIFO semantics for a set of invocations are observed between the same client/server pair.

Any active object method may be invoked as an inlined call. This explains the requirement in the SODA programming model for active object methods to be non-blocking. If this is not the case, an inlined call is blocked for the duration of the method execution. This could delay a whole invocation chain and a set of active objects. It is therefore the programmer's responsibility to ensure this property. If blocking calls are required, these can be isolated in detached methods (see §4.4.3).

4.3.6 Self-Invocation

As mentioned before, all active objects in SODA define a variable `thisActive` that implements a self-reference, similar to the keyword `this` in Java and C++. `thisActive` points to the active object's skeleton and is automatically created during the compilation process and initialised upon active object instantiation. `thisActive` can be used to hand out references to the current instance to other active objects. It can also be used for an active object to perform indirect self-invocation and iteration within an active object.

When an active object performs an indirect self-invocation, client and server object are identical. Consequently, the server will be busy with the current level of iteration. As a result, self-invocation is therefore always performed as a local invocation, queued on the server's skeleton. No inlined call can be performed. The current iteration will run to completion with a Future, setting up a Funnel on the subcall, which represents the nested iteration.

4.4 Active Object Multiplexing

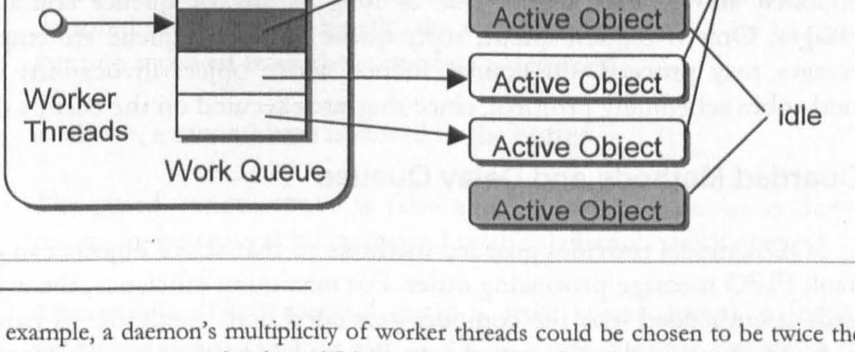
Many other Java-based active object systems [39;132] map active object instances onto threads in a one-to-one manner. This incurs a high overhead if a large number of active objects happen to be located on a single base. It also fixes a program's potential parallelism to the number of active objects. To avoid these problems, SODA uses a multiplexing scheme to associate active objects with system resources. Besides inlined invocation (§4.3.5), this is the second mainstay for SODA's lightweight active objects.

Multiplexing is based on a fixed number of worker threads per daemon. These are allotted to active objects on a round-robin basis. The number of threads can statically be adapted to a daemon's physical parallelism.[25] The skeletons cooperate locally to handle scheduling and multiplexing of the daemon's active objects. This includes queuing of incoming requests and replies as well as temporary suspension for requests whose guard conditions are not currently satisfied. If a request can be serviced, the skeleton collects the resulting server-side Future object and returns it to the client skeleton to be despatched to the client-side Future.

4.4.1 Base-Local Scheduling

Active objects that are non-idle are scheduled in a round-robin manner per base. This is possible through a work queue. Active objects enter this work queue if they have pending requests or replies to execute. If no further activities are pending for an active object, it is removed from the work queue. A pool of worker threads operates on the scheduling queue (see Figure 4-12). Each worker thread locks the next active object in the work queue and devotes processing power to it. This guarantees that worker threads access objects in mutual exclusion as is required for atomic active object semantics. Under the control of a worker thread, the active object then performs the following actions:

Figure 4-12 Base-Local Thread-Multiplexing of Active Objects



[25] For example, a daemon's multiplicity of worker threads could be chosen to be twice the number of available processors on that host. This would guarantee that the potential parallelism is higher than the physical parallelism without overwhelming local resources through the creation of an excessive number of threads.

(1a) If the reply queue is not empty, a reply is removed from it and the contained result and exception made available to the client-side Future. If a Funnel has been set up on the Future, then the Funnel's `receive` method is executed.

(1b) Otherwise, a request is removed from the request queue and executed on the body. After execution terminates the skeleton is responsible for sending the `Future` result back to the client. However, this is only possible when the Future has an actual value set, either explicitly by `setResult()` or `setException()` or implicitly through a Funnel.

(2) Since activity (1a) or (1b) may have changed object's internal state, the next step lies in rechecking the guards of any delayed requests. If any of the queued requests are satisfied, they are processed.

(3) If any of the requests in the delay queue have been processed in (2), the object's internal state may have changed. Therefore, other queued requests might have their guards fulfilled. Therefore, repeat (2) until no delayed requests can be processed any more.

(4) Reinsert the active object into the work queue unless both the reply queue and request queue are empty. If reply queue, request queue and delay queue are empty, the skeleton's `busy` flag is set to `false` which is an indication that inlined calls are acceptable. Finally, control is given back to the worker thread, which joins back into the pool of idle workers.

Fairness and Synchronisation

The SODA model does not make any fairness assumptions towards the scheduling of active objects. It is just an artefact of our implementation that scheduling is round-robin amongst active objects on a single base. An earlier version of SODA made the scheduling order dependant on the request arrival order. However, this approach was abandoned as it was found to give an unfair advantage to frequently invoked objects.

Skeleton driven scheduling must be synchronised with inlined calls to prevent concurrent access to the body. This coordination is achieved through the `busy` flag mentioned above. This flag is true as long as any of queues contain pending messages. Only if request queue, reply queue and delay queue are empty, inlined messages may proceed. Of course, inlined active object invocations violate the round-robin scheduling protocol, since they are executed on the client's thread.

4.4.2 Guarded Methods and Delay Queues

The SODA model provides guarded methods so that active objects can change the default FIFO message processing order. For maximum efficiency, the evaluation of guards is embedded into the compiler-generated body methods. In particular, the guard condition is directly copied into the body's method as the first statement. The body throws a `GuardException` if the guard's condition is not satisfied. This serves as a signal for the skeleton. In this case, the request is appended to the delay queue for later re-evaluation. Java's exception handling mechanism gives good efficiency here, since try/catch blocks carry a near-zero overhead in most JVM implementations [150].

In the current implementation, all requests in the delay queue are re-evaluated every time the active object undergoes a change of internal state. This is the case after successfully serving a request, a reply or another delayed request, as outlined in the previous section. Although guard evaluation is cheap, this is not an optimal solution, since it could lead to frequent guard re-evaluation cycles (see §5.6).

Since SODA guard conditions are not allowed to have side-effects, a re-evaluation of the guard does not cause any inconsistencies to the object state. It is also negligible in terms of performance, if the guard is based on a simple Boolean expression, based on the object's state or on the value of any of the request's parameters.

Support for Ferenczi Semantics in Guards

In §3.6.2 we gave an overview of how the three instances of inheritance anomaly are circumvented following Ferenczi's proposal of guard semantics of inherited objects [59]. We will now go on to describe how these semantics are implemented in SODA.

In chapter 3, Code 3-11, we gave an example solution for history-only sensitive anomaly (IA-2) Code 4-3 shows the class `GBoundedBuffer_Body` as created by the SODA source-to-source translator. The method `gget` cannot be executed as an immediate successor to a `put` method. This history information is recorded through an additional variable `afterPut` in the subclass. Code 4-4 is the equivalent body for the original bounded buffer class, `BoundedBuffer_Body`.

Consider the following sequence of invocations on a newly created instance of the `GBoundedBuffer` active object: `put`, `put`, `gget`, `get`.

`put` `GBoundedBuffer`'s method guard is true; `super.put(x)` is invoked. The superclass guard is true because buffer space is still available and the item x is stored in the buffer. Finally, the variable `afterPut` is set to true and the `Future` received from the superclass is returned.

`put` as above, a second item is stored in the buffer.

`gget` The guard `(!afterPut)` is false and a `GuardException` is thrown. This results in the request being moved to the skeleton's delay queue.

`get` `GBoundedBuffer`'s method guard is true; `super.get()` is invoked. The superclass guard `(in >= out + 1)` is `true` because two items are in the buffer. Finally, the variable `afterPut` is set to `false` and the `Future` received from the superclass (containing the first item deposited by put above) is returned.

Since the get call did succeed without being moved to the delay queue, it is assumed that the active object state has changed. Therefore, the delay queue is re-evaluated. This only concerns the previously queued request to gget. The guard (!afterPut) is now false and the get method is invoked. This results in a call to super.get(). The corresponding guard is true, because there is still one remaining item in the buffer.

Code 4-3 Body for the G-Bounded Buffer after translation into Java

```
public class GBoundedBuffer_Body extends BoundedBuffer_Body {

    private boolean afterPut = false;

    < ... initialisation of thisActive, and other householding    >
    < ... constructors                                            >

    // this method cannot execute as immediate successor to get().
    public Future gget() throws GuardException {
        if (!(!afterPut)) throw guardException;
        return get();
    }

    public Future put(int x) throws GuardException {
        if (!(true)) throw guardException;
        Future f = super.put(x); afterPut = true;
        return f;
    }

    public Future get() throws GuardException {
        if (!(true)) throw guardException;
        Future f = super.get(); afterPut = false;
        return f;
    }
}
```

Code 4-4 Body for the Bounded Buffer superclass after translation into Java

```
public class BoundedBuffer_Body extends ActiveBody {

  int[] buf; int size = 0;
  int in = 0; int out = 0;

  < ... initialisation of thisActive, and other householding    >

  public Future init(int size) {
    Future f = new Future();
    this.size = size;
    buf = new int[size];
    f.putResult(null);
    return f;
  }

  public Future put(int x) throws GuardException {
    if (!(in < out + size)) throw guardException;
    Future f = new Future();
    buf[in++%buf.length] = x;
    f.putResult();
    return f;
  }

  public Future get() throws GuardException {
    if (!(in >= out + 1)) throw guardException;
    Future f = new Future();
    int x = buf[out++%buf.length];
    f.putResult(new Integer(x));
    return f;
  }
}
```

As this example shows, if at any level of the inheritance hierarchy a guard evaluates to false, the overall invocation is abandoned and rescheduled. The SODA translator inserts the guard expression as the first statement in a method. Superclass method invocations must be the first statement after that. A call can be abandoned at any point during guard evaluation, because no side-effects to the active object state have occurred yet.

4.4.3 Detached Methods

Active object methods recruit threads from the base-local worker pool to process incoming requests and replies in a multiplexing fashion. The execution of detached methods may be long-lasting and blocking. In order not to tie up active object worker threads, a separate thread pool is used for detached methods. This allows blocking activities, such as timers, blocking operating system call and integration of legacy passive objects which perform blocking synchronisation. During execution of a detached method, the associated active object cannot be migrated. This would require strong mobility, which is difficult to implement [32].

Detached methods are implemented as separate classes. This restricts detached methods to their own private data but prevents direct access to the active object's data. This guarantees that no inconsistencies through concurrent access to the declaring active object's data can occur. The only communication path is given through an active object proxy, which allows the invocation of active object methods like for any other client.

4.4.4 Liveness Issues

SODA active object methods are not allowed to invoke a blocking `get` on a Future. This ensures that an active object instance does not enter a livelock where it cannot receive further requests, due to blocking of its allocated worker thread. Instead, Funnels are provided as a non-blocking, data-driven mechanism to collect the results of Futures as they become available.

Therefore, a SODA program is free of deadlock, except if one of the following situations occurs:

(1) A detached method (not the primordial method) uses a blocking `Future.get` call.
(2) Guards can cause livelock, where all active objects in a system are idle while all pending requests are bound to delay queues.
(3) Deadlock or blocking can occur in a contained passive object that is used by the active object. If a private, passive object blocks the active object's thread, this temporarily suspends the owning active object instance from processing any more requests or replies.

Situation (3) can occur, when the passive object performs some internal synchronisation and suspends the active object thread while in a monitor. Such a passive object is not compatible with SODA, since it is designed as a synchronisation point for several threads. Since in SODA, only a single thread can enter an active object at any time, a deadlock is inevitable. Another example is a blocking call into the operating system, performed by the passive object. For instance, a `receive` call could query a socket's input stream for further data from the network. If no data arrives on the stream, this call may timeout; still it delays the owning active object for the timeout interval. In the worst case, `receive()` has an infinite timeout, rendering the owning active object useless.

To avoid such situations, potentially blocking calls in SODA are restricted to detached methods of an active object. Detached methods have a separate thread allocated to them and therefore cannot interfere with the owning active object's operation.

4.5 Improved Object Serialisation

An invocation on an active object may use passive objects as parameters. These can be of primitive or any complex data type. Any such invocation may potentially cross network boundaries, wrapped in a request or a reply. Therefore, we require the ability to flatten complex data types (i.e., any type of Java Object) into an equivalent byte array for network transmission.

Java *object serialization* [164] provides this capability. Java Object serialization is a powerful and very flexible technique that requires little extra coding from a programmer. However, this flexibility comes at the price of a large amount of overhead, a serious impediment for high performance cluster computing. For example, object serialization takes about 25% to 50% of the time needed for a method invocation in RMI [139;174]. Since all this overhead is in software, the relative overhead increases as networks become faster.

We will examine object serialization in the following sections and present an alternative serialization mechanism, called SODA fast serialization, which provides more specialised serialization but significantly better performance. This is achieved through more explicit encoding and decoding routines and stateful streams that can cache previously-sent information.

4.5.1 Overview of Java Object Serialization

Java Object serialization [164] is a mechanism that allows the "flattening" of Java objects onto byte streams and vice versa. This is a significant functionality that enables the transmission of objects across the network or their persistent storage on file. The byte array representation of an object includes all its primitive instance variables and the complete graph of objects to which its non-primitive instance variables refer.

Usage

A class must implement the `Serializable` *marker* interface in order to be considered for serialization. A `Serializable` object can then be passed to the `writeObject` method of the class `ObjectOutputStream`. This method can deal with complete object graphs, even if these contain cyclic references. Multiple references to the same instance are encoded as *back references* to prevent infinite loops during serialization.

The `ObjectInputStream` class provides a matching `readObject` method for deserializing the object. Serialisation does not transmit the byte code of the class, which must be available to the receiving JMV's class path. If this is not the case, a `ClassNotFoundException` is thrown. However, the version of the class may be different at the receiver and sender JVM, since serialization can gracefully deal with evolving classes as explained below.

Serialization Overheads

Class evolution is supported by a unique version identifier contained in the serialization wire protocol. Newer class versions can be explicitly compatible with their predecessors. In addition, the name, type and value are encoded for every instance variable[26]. Information about an instance's type and variable structure is discovered at runtime, using Java's introspection features. This operation however, carries a significant runtime cost and moreover, must be repeated for every instance sent.

The design of Java serialization is clearly focussed on ease-of-use and flexibility rather than high performance. In fact, [139] contend that object serialization yields "catastrophic performance". This is not tolerable in a HPC environment. Usually, the lifetime of an object in HPC programs is shorter than the runtime of a parallel program; no long-term persistence is required. It is also fair to assume that identical versions of every class are available on every participating base. Therefore, the full generality default Java object serialization is not necessary. This yields room for the following optimisations to serialization:

(1) No class metadata, such as structural and versioning information, is required in a byte stream. Therefore, it is sufficient to transmit an object's class name and the values of all primitive types in order. No name or type information for primitive values is required. This process is repeated recursively for nested objects, using back references where necessary.
(2) Instead of using introspection, the values of all primitive variables can be written out in an explicit marshalling routine and read back by a matching unmarshalling routine. Such routines can be generated manually for every class, or automatically by a compile-time tool. Another approach is to use load-time class transformation.

The `Externalizable` Interface

Java provides the `Externalizable` interface that can yield the functionality of (1) and (2) above. `Externalizable` defines the `writeExternal` and `readExternal` method to give complete control over the format and contents of the stream for an object. These methods must explicitly coordinate with the superclass. If an `Externalizable` object is passed to an `ObjectOutputStream`, only its identity is written to the stream, before the `writeExternal` method is called. On the receiver side, an `Externalizable` object is reconstructed by creating an instance of the transmitted class identity, using the public no-argument constructor.[27]

Since no reflection is required to find the names, types and values of all fields, the serialization process for `Externalizable` objects is much faster than for `Serializable` objects. In addition, the stream representation is more compact. However, introspection is still used to a small degree: on the sender side, to get an instance's class name, and on the receiver side, to create a new instance based on the received class name. The required methods `Object.getClass().getName` and `Class.forName(<name>).newInstance()` have relatively high overhead on most JVMs due to the internal use of native methods. Therefore, these types of introspection are not well suited for performance sensitive code.

[26] Static fields and transient fields are excluded.

[27] This constructor must be provided by every class that implements `java.io.Externalizable`.

4.5.2 Other Approaches

Several authors have examined alternatives to Java's default serialization scheme. The **JavaParty** project has produced an improved serialization scheme [128;139] for Java that can yield a performance gain of up to 97% over standard Java serialization. The implementation is based on explicit serialization/deserialization routines that manage byte buffers explicitly. This reduces the runtime overhead for reflection; however, the routines are tedious to write. Also, the implementation is not portable across different JVMs, since a change to the standard class files is required.

Manta [108;172] implements an RMI that is a further improvement over that of JavaParty. Much of Manta's performance improvement derives from their implementation of a native compiler, and the whole-program analysis used by that compiler. Furthermore, the compiler takes special actions when compiling RMI code so that JNI (Java Native Interface) calls are avoided. In fact, communication is inlined into the code, increasing the speed and responsiveness of the system still further.

Matt Welsh's **Jaguar** system [181] is based on a JVM extension that enables direct access to native memory regions outside of the Java heap.[28] This functionality can be used to allocate Java objects, so called *pre-serialized objects*, into the native heap. All state changes to the Java object are transparently replicated to this memory area.

4.5.3 SODA Serialization

Common to all the above serialization mechanisms is that they require changes to either the JVM or the standard class libraries. In contrast, SODA provides a serialization scheme that can run on an unmodified JVM. In addition, further optimisations are exploited by *caching serialization streams*. Efficiency arises in large part from abandoning the official Sun protocol in favour of a more compact, but less versatile, protocol. SODA Serialization is based on explicit encoding and decoding routines, similar to `Externalizable`. In the current implementation, these methods must be explicitly coded, but it is possible to generate these automatically at compile-time.

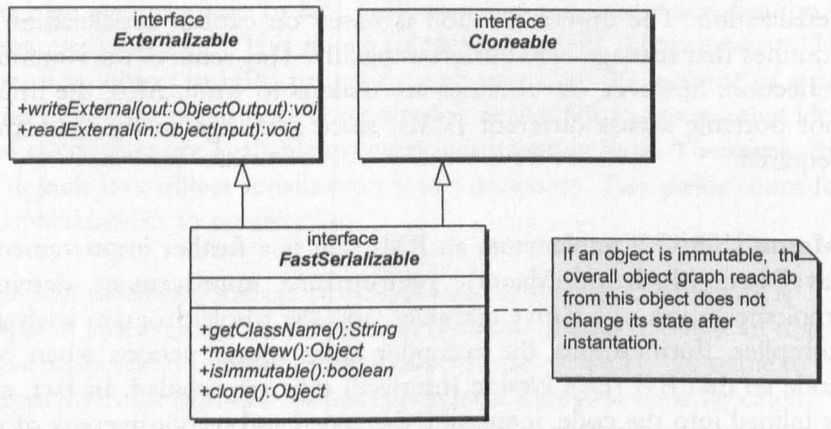
The `FastSerializable` Interface

SODA Serialization is supported through the `FastSerializable` interface which extends `Externalizable` (see Figure 4-13). For `FastSerializable` objects, runtime reflection is almost completely avoided. To determine the class a virtual method `getClassName()` is provided that returns the fully qualified class name. Only for the first occurrence of a previously unencountered class is the `Class.newInstance()` method invoked on the receiver side. The returned instance is then stored as a "prototype" by the receiving stream and all further transmissions use this prototype's `makeNew()` method to manufacture new

[28] A similar functionality is provided by the `nio` package in Java version 1.4.

instances. This virtual method dispatch yields much better performance than introspection.

Figure 4-13 SODA `FastSerializable` Interface


interface
Externalizable
+writeExternal(out:ObjectOutput):voi
+readExternal(in:ObjectInput):void
interface
Cloneable
interface
FastSerializable
+getClassName():String
+makeNew():Object
+isImmutable():boolean
+clone():Object
If an object is immutable, the overall object graph reachab from this object does not change its state after instantation.

If a `FastSerializable` object is immutable, it guarantees not to change state over the course of its lifetime. This guarantee includes the instance's primitive variables as well as all recursively referenced objects.

Caching Serialization Streams

At the core of SODA serialization are stateful, caching serialization streams. The usage of these streams is identical to default Java object streams. However, `FastOutputStream` and `FastInputStream` improve performance by indexing frequently sent information via receiver-side caches. If an object to be serialized implements `FastSerializable`, these streams offer a vast performance improvement over default Java Object serialization. As a fallback mechanism the `Externalizable` and `Serializable` are still supported, albeit with lower, default performance.

The caching streams avoid repeated transmission of frequently sent information. Data which has already been sent before is cached at the receiver side. For example, the fully qualified class name of an object is cached, as is the state of any immutable object. If a set of objects of the same type is sent, the second and subsequent transmissions therefore carry only very slim type information. Circular references are correctly resolved as back-references, using hash-tables to identify already sent instances.

4.6 TCP/IP Transport Service

The transport service is responsible for exchanging request and reply messages between participating bases in a SODA system. This is a crucial and extremely performance-sensitive component of the runtime system. The overriding design concern was the absolute minimisation of remote method call overhead for typical workstation clusters. Here we describe the default transport service, built on TCP/IP sockets.

The TCP/IP transport implements the `VMID` interface as a combination of IP address and port number. Several bases can therefore be collocated on the same host, bound to different port numbers. Messages are transferred as a result of a call to `send (VMID, OID, msg)`. The SODA serialization protocol described above is used for serialization of `VMID` and `msg`, whereas `OID` is just an integer. Socket connections to remote bases are acquired lazily and released on an LRU basis. Race conditions, where a pair of bases tries to connect to each other simultaneously are resolved correctly. This dynamic connection management allows better scalability than a fully-connected approach and better reflects the SODA design philosophy which aims at creating localised clusters of activity without global communication. One reason is that, prior to version 1.4, Java sockets do not support the `select()` call. Therefore, a separate thread must be attached to each open socket.

SODA can perform batching optimisations, if a set of calls is in flight simultaneously between a pair of machines. In this situation, several request/reply messages are combined into a single network packet, which reduces overheads in the operating system's TCP/IP stack. For this purpose, a double buffering scheme is used. While one buffer is being filled, the second one is written onto the TCP/IP network socket by an asynchronous thread. When the second buffer has been written completely, the two buffers are swapped. This is beneficial if many small messages are sent since the number of required network packets is reduced.

One limitation of the current transport service implementation is that only a single transport layer can be used at any time. Modern networked systems often have several network interfaces at their disposal. Therefore, ***multi-path communication*** [19] would be a valuable future addition to the runtime system. Multiple networks or transport mechanisms could then be used for simultaneous data transfer between communicating bases. This would increase overall bandwidth and reduce the number of collisions in non-switched networks.

A further improvement would be useful for SODA systems that extend over a WAN network. Here, it would be useful to define a hierarchic communication structure. For example, it would be sufficient if only a single socket connection existed between two bases in geographically separate clusters. Messages exchanged by other bases could then be multiplexed and de-multiplexed over the single WAN socket by the gatekeepers.

4.7 Load Balancing Service

SODA facilitates load balancing through dynamic object migration through the autonomy and location transparency of SODA active objects. The current implementation of the load balancing service is at a rudimentary stage. Currently, only instantiation-based assignment according to system load is supported. Dynamic object migration has only been designed on paper.

For object allocation purposes, every base reads a list of the available bases from a configuration file (`conf/basefile`). An identical version of this file must be available to all bases. At regular intervals, every base sends out a multicast packet that conveys local load information to other bases. Based on the incoming multicast packets, every base can assemble a picture of the system's load situation.

A load balancer implementation can currently get a rich variety of information about the active objects and their interaction on the local node. This information is obtained from both base meta-object and active objects located on that base. The following information can be accessed:

- Number of active objects on the node.
- Number of pending messages overall on the node.
- Load of the node (obtained by measuring the time needed for performing a small benchmarking loop).
- Average message queue length for an active object.
- An active object's acquaintances and interaction frequency.

Object Migration

Two possibilities exist for support of object migration. If an object can be migrated while a method call is in progress, this is called ***strong mobility***. In contrast, ***weak mobility*** allows migration only at certain points during the lifetime of an active object. Weak mobility is much easier to implement, since no activity information of the object (such as thread stacks, etc.) need to be transferred [32].

Active objects in SODA support weak mobility; possible migration points exist whenever processing of the currently active method has terminated. At this point, the object migration algorithm suspends further pending messages. The object can now be transferred to another base. The full information required for the transfer includes the following:

- Request queue, delay queue and reply queue.
- Any pending Futures that have not yet been sent back to their clients and associated Funnels.
- The active object's state (i.e., member variables)

When an active object is migrated, it is necessary to update the stale proxy references so that they point to the new location. One way of doing so is for the active object to keep track of all its proxies and explicitly update them upon migration. This approach requires newly created proxies to register with the active object and vice versa. This can create a large amount of communication and therefore impede on scalability.

A lazy approach to updating stale proxies when they are used seems more appropriate. One solution is given through *forward pointers* [140] left behind at the original location after an active object migrates away. However, if an object migrated several times, the forward chain could be very long.

To avoid this problem, we propose the concept of a *master locator* for every active object. This solution is similar to the one implemented in the NIP runtime system

[177]. The master locator is located on the original location where an active object was instantiated and is referenced through a tuple `<VMID_master, OID_master>`. Whenever an active object migrates, it informs the master locator about the new location. If the client proxy points to a stale location the master locator can be queried for the new location. For this approach, client proxies must be updated to include information on the destination object's master locator, e.g., `<VMID_current, OID_current, VMID_master, OID_master>`. A stale client proxy is detected if the server object is not found at the location `<VMID_current, OID_current>`. In this case, the client object receives an exception and in response blocks all further requests and queries the master locator for the new location. The master locator will only reply when the new location of the server active object is known. The client proxy then updates its `<VMID_current, OID_current>` tuple and resends the queued requests to the new location. For a client proxy of a newly instantiated active object (which has not yet migrated), the following is true: `VMID_current = VMID_master` and `OID_current = OID_master`.

When an active object migrates from its current base to another base, it takes the following steps:

(1) Wait till the last invocation has terminated and then suspend the pending requests queue.

(2) Invalidate the master locator (i.e., all further requests to the master locator will be blocked until it receives the updated location).

(3) Remove object from the base dispatcher (i.e., any incoming requests to the current `<VMID:OID>` will cause exceptions in the original clients

(4) Serialize the object state and the queue of pending messages and transfer to the new base.

(5) Find a free `OID` on that base and register with the base dispatcher.

(6) Update the master locator with the new position.

(7) Schedule processing of any queued method invocations in the pending requests queue.

After step 6, any client proxies that tried to send a request to the active object during migration, will receive the new location from the master locator. They can then resend any queued method invocations to the new location.

4.8 Future Work and Conclusions

While the SODA programming model is similar to ProActive PDC [39], its implementation is geared towards exploitation of more fine-grained parallelism. Since active objects have comparatively small overheads, this encourages a programmer to write programs with a large number of active objects. This is beneficial, because programs become more portable across parallel architectures with different degrees of parallelism. If little physical parallelism is available, client and server active object will be collocated in most cases. This yields a high probability for low-latency inlined invocations. Of course, this depends on an efficient runtime allocation of active objects that collocates frequently interacting objects. In a large system with many bases and high physical parallelism, the available active objects can be spread out more widely and the program can make full use of all available processors.

The RTS described here is by no means the most efficient implementation of the SODA model. It has been explicitly designed to allow easy experimentation and inclusion of new ideas which we find useful. Nevertheless, the experience gained shows that the model is easy to use and can capture a wide range of problem domains. Future versions of the runtime system should address the following issues.

Load Balancing. Load balancing through dynamic object migration as described in §4.7 above is only designed on paper. However, this is an important feature for a runtime system that supports active objects with changing load characteristics. This would require monitoring and analysis of object interaction and activity patterns and heuristics that decide on object migration based upon this information.

Statically Typed Futures. Statically typed Futures are not yet supported, although this would be possible. For example, using a version of Java with support for parameterised classes, Future classes could be generated that match the return type of the associated method. Also, Futures do not directly support primitive data types at current.

Heuristic Inlining. Currently all local, non-guarded method invocations on idle active objects are performed as inlined call. This stack-based invocation is efficient for single-processor nodes. However, when multiple processors are available per node, potential concurrency is lost.

Distributed Garbage Collection. SODA does not currently implement garbage collection. This could be provided through an additional plug-in service (see §4.1.5). Various hooks in the master locator and in the active object proxies exist to retrieve information about new proxy/reference generation, etc. This could be used e.g., to create a reference counting garbage collector. The garbage collector can register handlers for the hooks in the proxies/master locator.

Chapter 5

Performance Results

The SODA runtime system provides a high-level abstraction layer on top of the physical distributed-memory hardware. This incurs efficiency loss compared to a lower-level programming model. The aim of this chapter is to determine estimates for these runtime costs. Based on these, we can assess the ease-of-use vs. efficiency trade-off and to determine SODA's practical value.

A number of micro-benchmarks are devised to obtain estimates for the overhead of basic SODA operations. These include the cost for the various active object invocation techniques and serialization of passive object parameters. We also examine the performance of invoking calls on various types of complex data structures that are composed from active objects. The results obtained are useful to predict SODA's scalability and to give design guidelines for more complex SODA programs.

The experiments presented in this chapter do not attempt to evaluate SODA's ease-of-use benefits. This point is deferred to the next chapter, which presents a large-scale real-world application built with SODA.

5.1 Experimental Environment

Two separate test bed environments were used for the performance analysis. The first was Mill, a network of workstations, consisting of 40 nodes. The second was Mega, a dedicated rack-mounted cluster with 16 dual-processor nodes. Table 5-1 gives a detailed description of the hardware and software. Unless otherwise mentioned, we only use a single worker thread per node, onto which all local active objects are multiplexed. This avoids node-internal speedup on multi-processor nodes, such as those of Mega.

Our main interested lies in the **speedup** that can be achieved by a SODA program compared to a sequential Java program for the same problem. All speedup measurements cited in the following sections are compared to the fastest possible sequential implementation in Java, rather than running the parallel algorithm on just a single processor. This gives a more realistic assessment of the benefits of SODA. In most experiments we use an active object with a method that has adjustable granularity as a test target object. Based on the execution time for this method, and the number of times it is invoked, it is simple to obtain a lower bound for the sequential execution time. Sometimes we also describe **efficiency**. This is the speedup divided by the number of processors in use.

Execution time for the experimental programs was measured based on readings of the system time. However, these can only be obtained in millisecond resolution. To measure very short times, we therefore use a loop that repeatedly executes the code to measure and automatically calibrates the number of iterations until they span over at least a 10 second interval. The exact runtime can then be divided through the number of iterations to obtain the atomic execution time. All measurements were done repeatedly, in order to obtain a 95% confidence interval. This is represented through error bars in the following diagrams.

The current version of the SODA runtime system has prototypical character. More efficient implementations of the programming model are feasible. Nevertheless, the experiments show good speedup values and therefore validate the SODA approach for the chosen example benchmarks.

Table 5-1 Overview of experimental systems and software configurations.

	Mill cluster	**Mega cluster**
Nodes	40 desktop type	16 node rack-mounted Dell PowerEdge 1550
Processor	Pentium III 650 MHz, 256 kb cache	Dual Pentium III 866 MHz, 256 kb cache
Memory	256 Mb	512 Mb
Network	10Mbit Ethernet, shared	100Mbit Fast Ethernet, switched
Op/System	Linux 2.4.18	Linux 2.4.7-10smp
JVM	Sun JVM 1.3.1 (b24)	Sun JVM 1.3.1 (b24)
heap settings	-Xms256Mb –Xmx640Mb	-Xms256Mb –Xmx640Mb

5.2 Remote Method Invocation Overheads

The cost of remote method invocation is crucial to any distributed object system. This is closely related to the minimum computational granularity that can amortise the overhead of network communication. Therefore, our first interest is to obtain the roundtrip latency for a SODA method call. We are only interested in the overheads of Future creation, data marshalling and demarshalling, queuing at the server, method dispatch, synchronisation and network latency for sending request and reply. Therefore the target method does not perform any work and has an empty parameter list (see Code 5-1).

Code 5-1 The empty active object method used for call latency measurements

```
  /** an empty active object method.
   *
   * @return a null-valued Future without any processing.
   */
  public Future testCall() {
    Future f = new Future();
    f.putResult(null);
    return f;
  }
```

We measure the time from Proxy method invocation until the blocking `get()` operation on the corresponding Future returns. For this we obtain a value of ~800 *µs* on the Mill cluster and ~350 *µs* on Mega. On both systems, this corresponds to approximately twice the network roundtrip latency (see Figure 5-1). This overhead is relatively small compared to other Java-based distributed object systems. For example, the same call using RMI takes ~1.3 *ms* on the Mega cluster. This improvement is due to the more efficient serialization technique used in SODA compared to standard Java object serialization.

When several messages are sent between a pair of hosts, SODA's default TCP/IP transport service will attempt to collate several messages into every exchanged network packet. This is done through a buffering scheme. A buffer of accumulated messages is written asynchronously over the wire. To measure the benefits of this buffering mechanism in terms of network efficiency, we modified the above experiment to perform a batch of calls (see Code 5-2). As expected the average per-call latency decreases as the batch size increases. At a batch size of four, the average per-call latency already drops below network latency. For very large batches we asymptotically reach a latency of 110 *µs* for the Mega cluster and 120 *µs* for the Mill cluster (see Figure 5-1). Of course, the wire cost can't decrease; instead, the cost for sending and receiving network packets is shared between several messages.

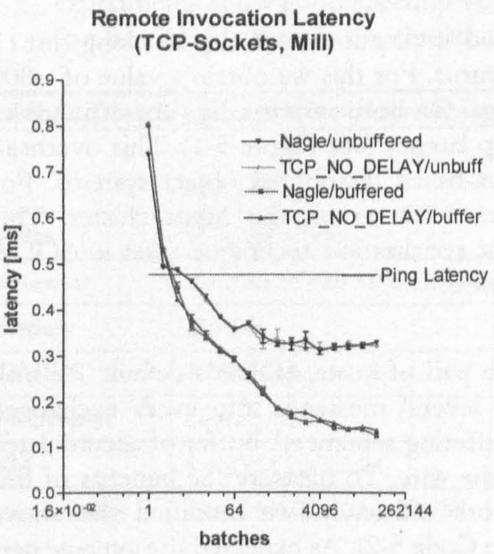
Code 5-2 Call batching and blocking get.

```
   Future calls[] = new Future[nrBatches];

   for (int i = 0; i < nrBatches; i++) {
       calls[i]  = remoteObject.testCall();
   }
     for (int i = 0; i < nrBatches; i++) {
       calls[i].get();
     }
```

We repeated these experiments with a combination of different settings on the TCP/IP transport: the buffering (asynchronous writing) as well as Nagle's algorithm [127] can be controlled independently. Disabled buffering significantly decreases batch performance on the Mill (asymptotically 360 *µs* for batched calls). Surprisingly, the effect is negligible on Mega. This could be explained with a multithreaded TCP/IP stack that makes efficient use of the nodes' second processor. On neither system could we detect a noticeable performance impact when disabling Nagle's algorithm. One exception is the Mega cluster, for a batch size of 32-256 calls. Here the combined use of Nagle's algorithm and unbuffered writing yields the best results.

In general, however, latency is smallest with Nagle's algorithm and buffered writing both switched on. This is therefore the default setting for the TCP/IP transport service.

Figure 5-1 Remote method invocation latency



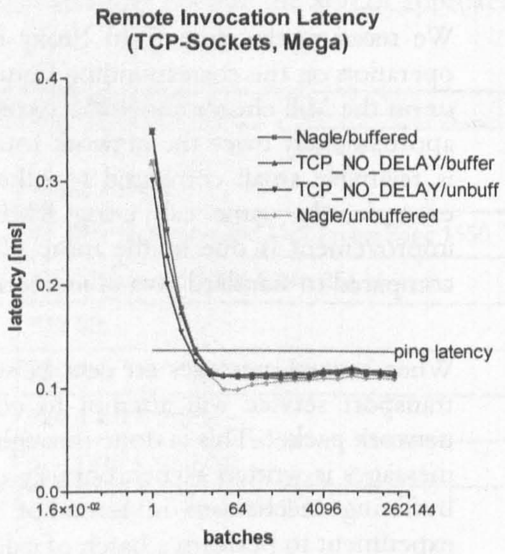
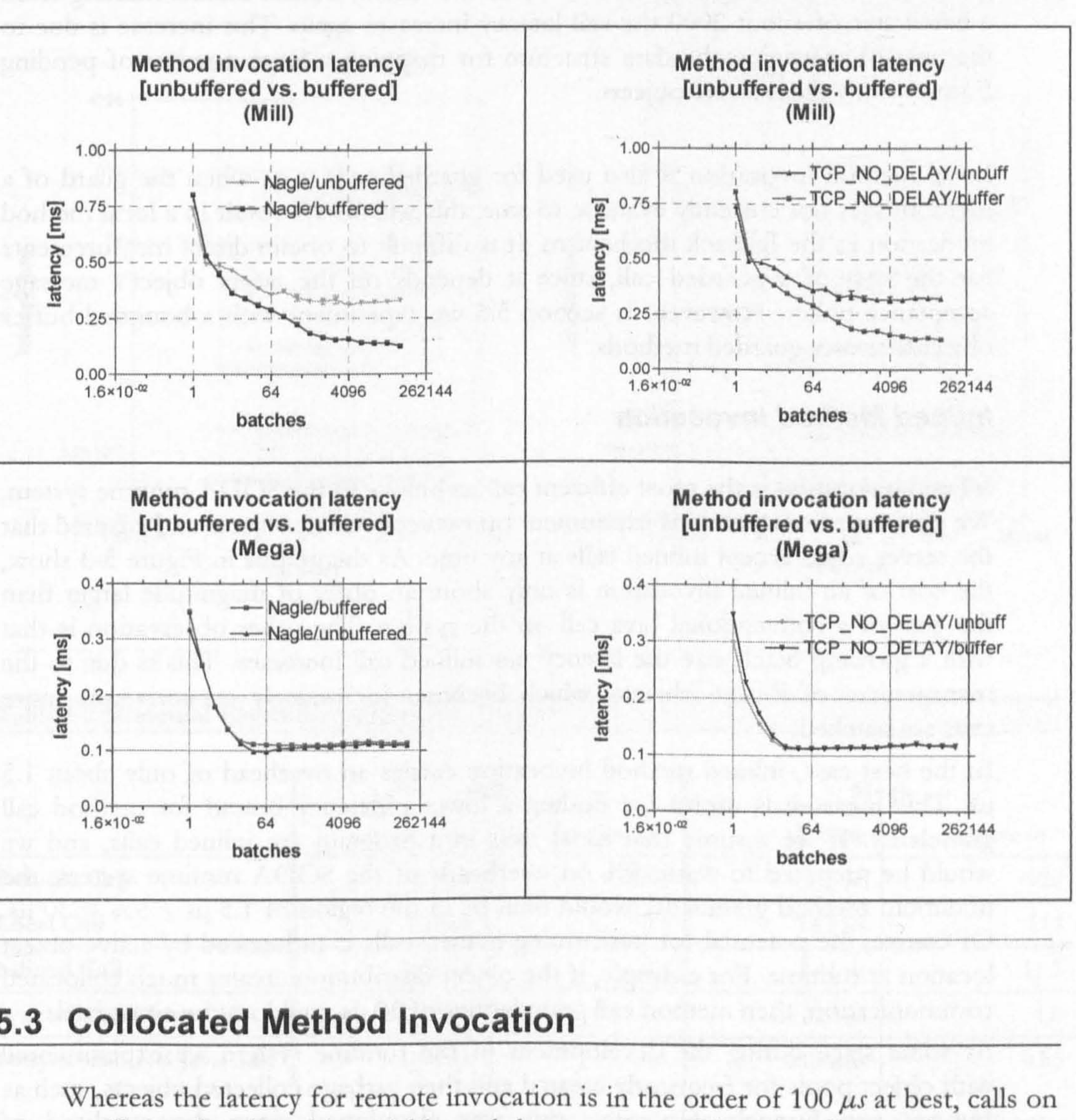


Figure 5-2 Effect of Nagle's algorithm and buffered sending on remote invocation latency



5.3 Collocated Method Invocation

Whereas the latency for remote invocation is in the order of 100 *μs* at best, calls on a collocated active object yield much better values. When the client and server are collocated in the same JVM, SODA has two invocation techniques at its disposal, each optimised for a different runtime situation. In decreasing order of overhead, these are **local** and **inlined** method invocation (see section 4.3).

Local Method Invocation

In the local invocation technique, request and reply messages are created and exchanged between client and server object. Both messages are queued in the request and delay queue respectively and the client and server active object run in separate threads. In order to enforce a local method invocation we manually modified the object's proxy.

As shown in Figure 5-3, the results are similar on both systems. The latency initially drops from about 30(Mega)/50(Mill) *μs* to 15 *μs*, then increases again. The statement block in Code 5-2 is executed within a detached method (this is the primordial method of the test active object). Therefore, the initial latency drop can be explained by a reduced amount of context switches. A set of calls can be generated before delegating work to the active object worker thread. Starting from a batch size of about 2000 the call latency increases again. This increase is due to the cost of managing the data structure for mapping a large number of pending Futures onto client active objects.

Local method invocation is also used for guarded calls. e.g., when the guard of a method does not currently evaluate to true, this will always result in a local method invocation as the fallback mechanism. It is difficult to obtain direct measurements for the cost of a guarded call, since it depends on the target object's message acceptance policy. However, in section 5.6 we experiment with a bounded buffer object that uses guarded methods.

Inlined Method Invocation

Inlined invocation is the most efficient call technique in the SODA runtime system. We performed our previous experiment on two collocated objects and ensured that the server could accept inlined calls at any time. As the graphs in Figure 5-3 show, the cost of an inlined invocation is only about an order of magnitude larger than the cost of a conventional Java call on the system. The other observation is that with a growing batch size the latency per inlined call increases. This is due to the management of Future objects, which becomes increasingly expensive, as more calls are batched.

In the best case, inlined method invocation carries an overhead of only about 1.5 μs. This measure is useful for finding a lower efficiency bound for method call granularity. If we assume that most calls in a program are inlined calls, and we would be prepared to waste 5% on overheads of the SODA runtime system, the minimum method granularity would then be in the region of 1.5 μs / 5% = 30 μs. Of course, the potential for performing inlined calls is influenced by active object location at runtime. For example, if the object distribution creates much collocated communication, then method call granularities of 30 μs could not be amortised.

At some stage during the development of the runtime system we experimented with object pools for frequently created and then garbage-collected objects, such as Futures and Funnels. However, this was abandoned since the overhead of maintaining the object pool could not be amortised with the savings in garbage collection and object creation time. This is due to the aggressive runtime optimisation of modern JVM implementations, which renders object pools inefficient for everything but very heavy-weight objects, such as threads or database connections.

Figure 5-3 Overheads for Different Invocation Techniques

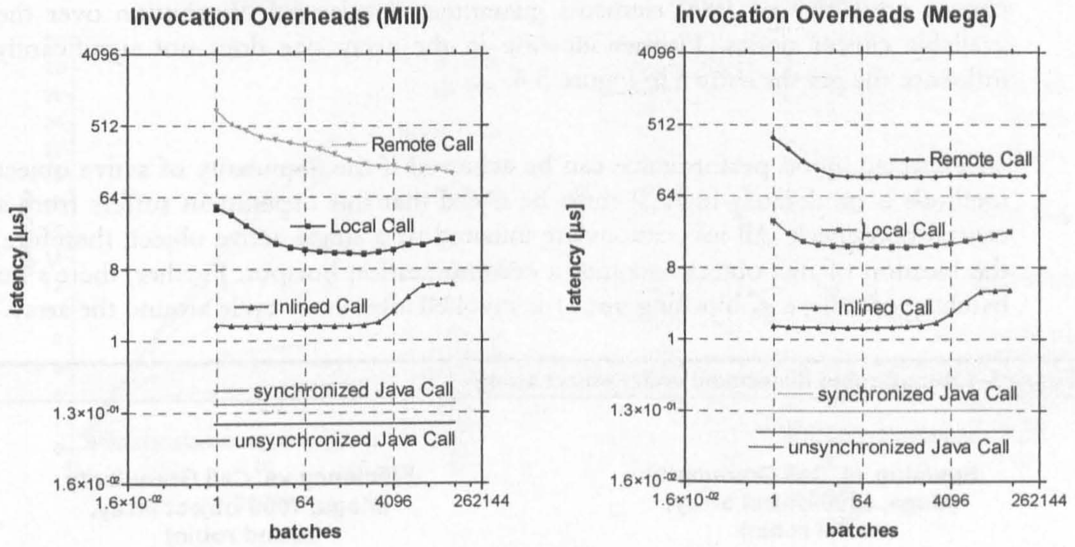


Table 5-2 Minimum Execution Overheads

	Mill		Mega	
	absolute	Relative to *	absolute	Relative to *
Remote Call	328 µs	~2000	115 µs	~920
Local Call	12.0 µs	75.5	14.3 µs	114
Inlined Call	1.50 µs	9.4	1.41 µs	11.3
Synchronized Java Call*	0.159 µs	1.0	0.125 µs	1.0
Unsynchronized Java Call	0.091 µs	0.57	0.069 µs	0.55

5.4 Active Object Data Structures

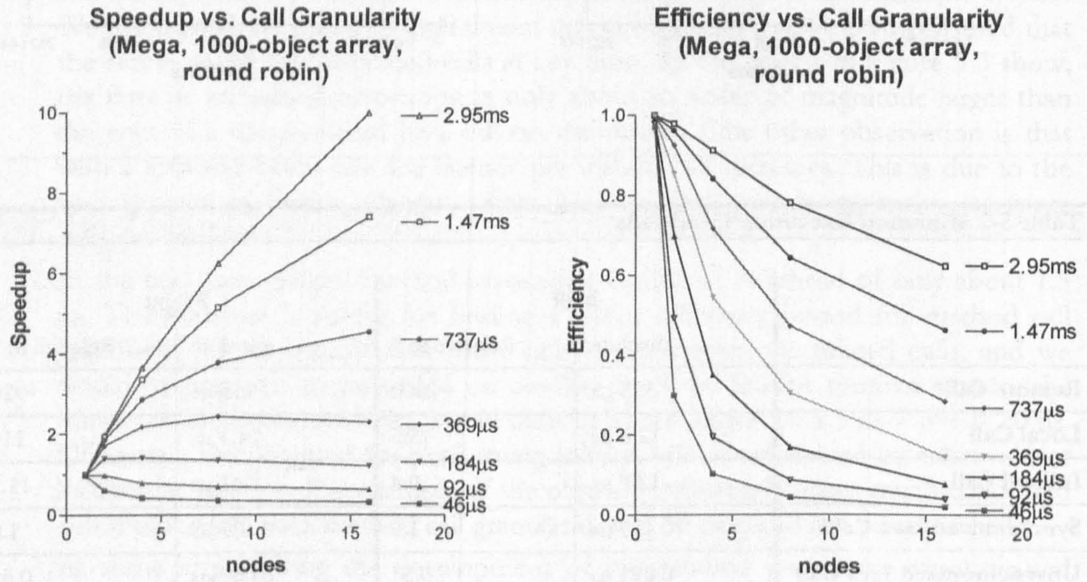
The previous experiments were based on a pair of active objects: a client (and primordial) active object and a server active object. The subsequent experiments focus on more complex data structures, comprising large sets of active objects.

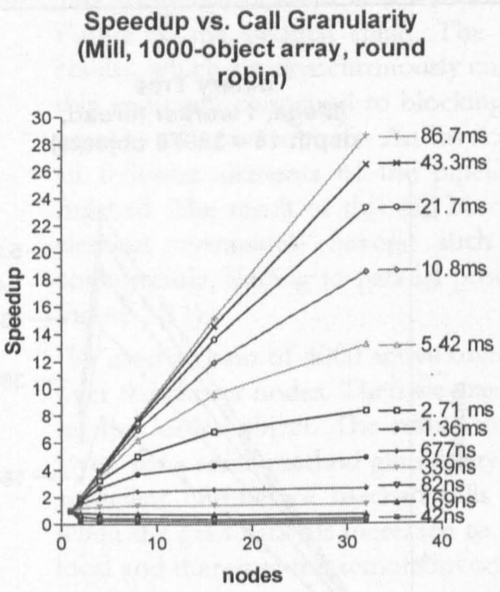
Object Array

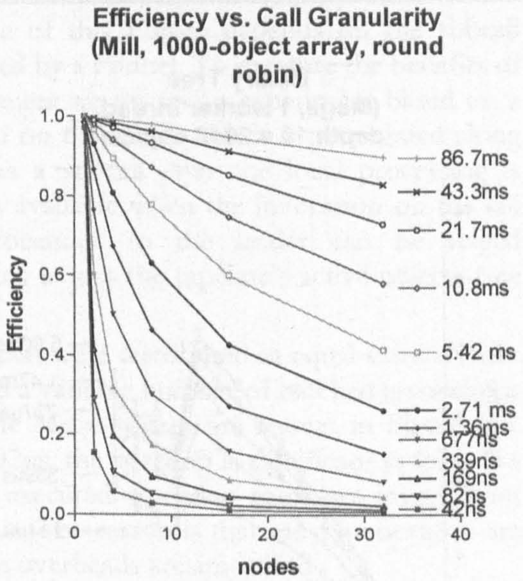
The first experiment involves an array of active objects. The objects in the array are round-robin distributed across nodes in the cluster. A single client object then invokes a method on each of the objects in the array. The granularity of this method can be adjusted. We also vary the number of participating cluster nodes. This experiment therefore gives an indication of potential speedup in relation to the method call granularity. Most method invocations in this experiment are remote. However, for a size *n* of the array, *1/n* invocations will be inlined invocations. As the basis for the speedup value we took a lower bound on the execution time: This is the product of method call granularity and array size. The chosen array size of 1000 elements guarantees a balanced distribution over the available cluster nodes. Further increase in the array size does not significantly influence the graphs shown in Figure 5-4.

As expected, good performance can be achieved if the granularity of active object methods is moderately high. It must be noted that this experiment suffers from a central bottleneck: All invocations are initiated by a single active object; therefore, the location of this object becomes a communication hotspot. Further, there's no batching of calls; e.g., blocking `get()` is invoked after every cycle around the array.

Figure 5-4 Round-robin distributed active object array







Object Tree

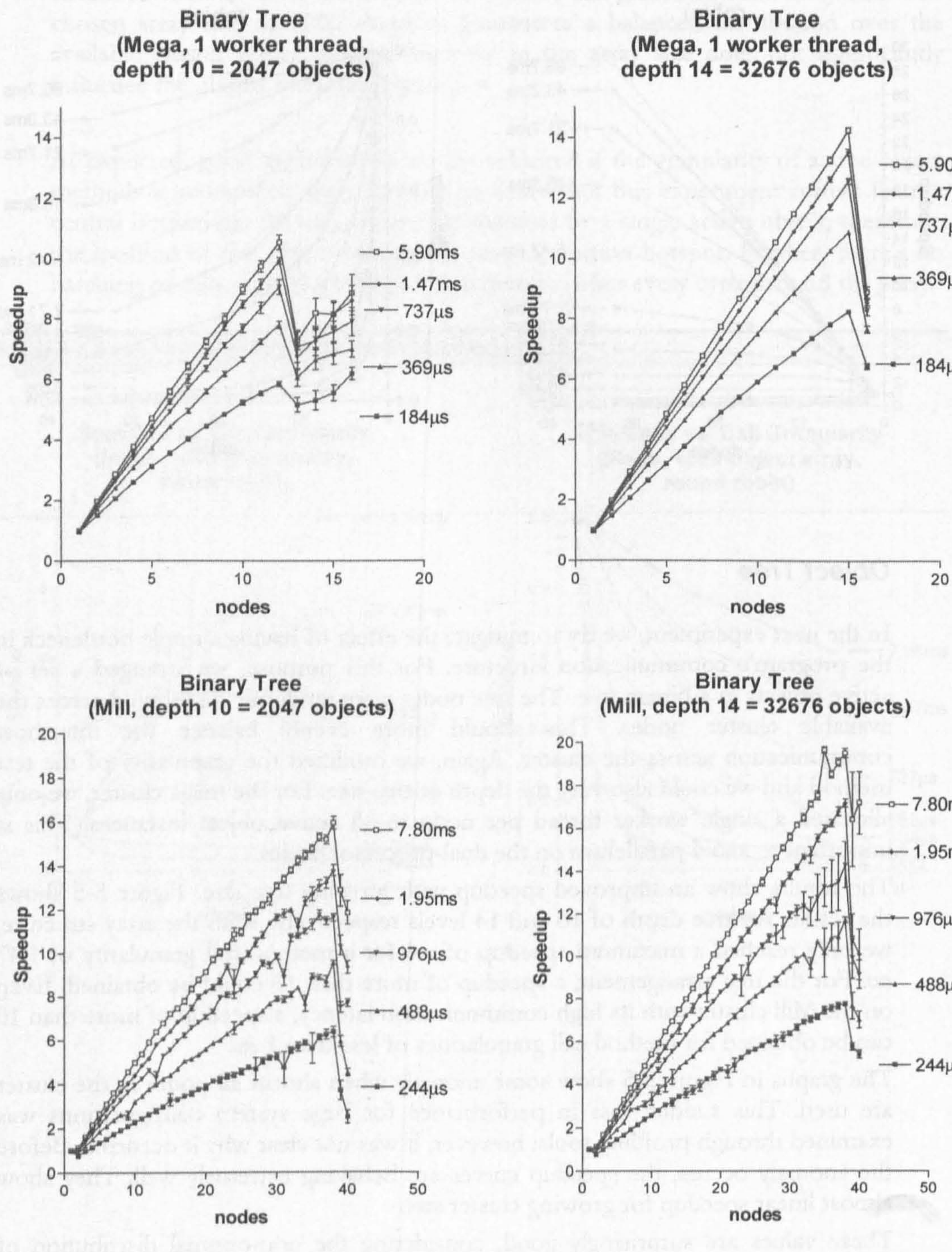
In the next experiment, we try to mitigate the effect of having a single bottleneck in the program's communication structure. For this purpose, we arranged a set of active objects in a binary tree. The tree nodes were randomly distributed across the available cluster nodes. This should more evenly balance the inter-host communication across the cluster. Again, we modified the granularity of the test method and we could also vary the depth of this tree. For the mega cluster, we only allocated a single worker thread per node to all active object instances. This is important to avoid parallelism on the dual-processor nodes.

The results show an improved speedup with growing tree size. Figure 5-5 shows the results for tree depth of 10 and 14 levels respectively. With the array structure, we only reached a maximum speedup of ~6 for a method call granularity of 1.47 *ms*. For the tree arrangement, a speedup of more than 13 could be obtained. Even on the Mill cluster with its high communication latency, a speedup of more than 10 can be obtained for method call granularities of less than 1 *ms*.

The graphs in Figure 5-5 show some anomaly when almost all nodes in the cluster are used. This sudden loss in performance for large system configurations was examined through profiling tools; however, it was not clear why it occurred. Before the anomaly occurs, the speedup curves are behaving extremely well. They show almost linear speedup for growing cluster size.

These values are surprisingly good, considering the non-optimal distribution of active objects over the cluster nodes: Much communication between nodes takes place at every level of the tree due to the random allocation of active objects onto cluster nodes. Ideally, no remote calls should take place below a certain tree depth, but only local or inlined ones. This could be achieved by distributing the tree-nodes breadth-first until saturation of all machines, and then do a local depth-first expansion for each node. This should further improve the speedup values obtained.

Figure 5-5 Tree recursion for random-distributed nodes.



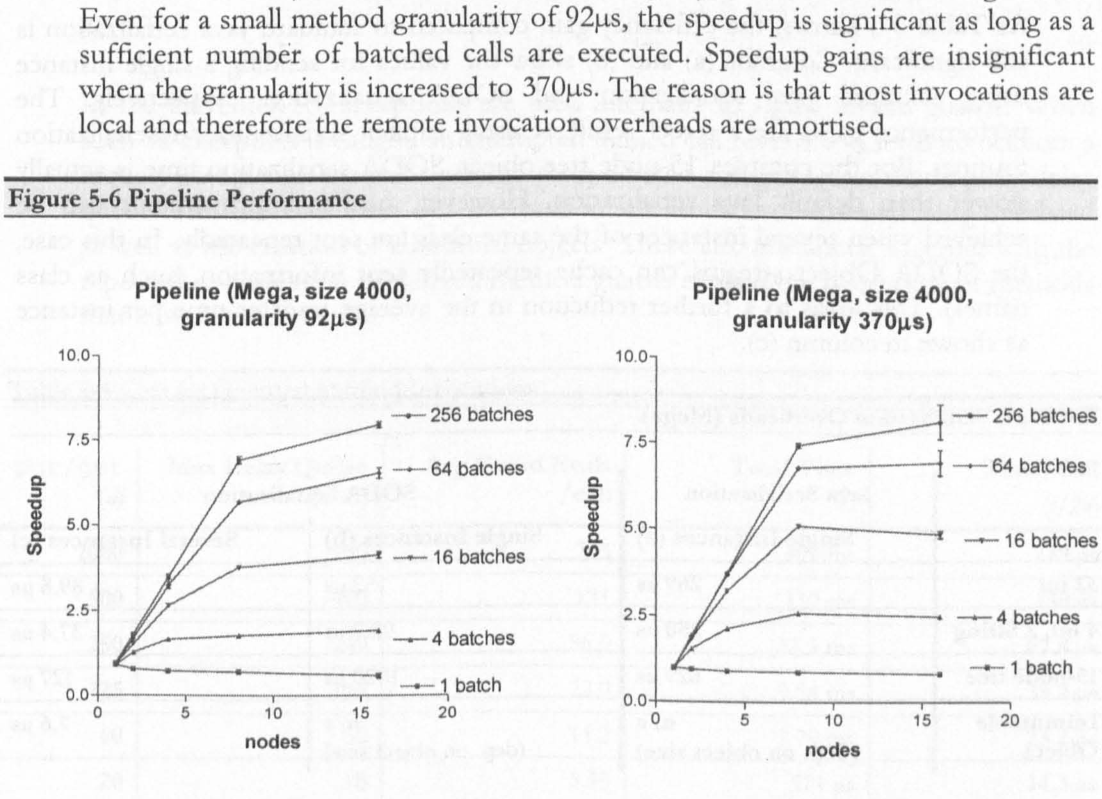
Pipeline

The advantage of the Futures and Funnels mechanism in SODA is that atomic active objects that perform subcalls are not blocked during the time of the subcall.

The active object method will proceed past any subcalls immediately and returns a Future to the original client. The value of this Future depends on the subcall results, which are asynchronously collected by a Funnel. To measure the benefits of this approach compared to blocking Futures we set up an experiment based on a pipeline of active objects. An invocation on the leader object is propagated along all follower elements of the pipeline as a subcall after the local processing is finished. The result of this call becomes available when the invocation on the tail element terminates. Several such invocations to the leader can be staged concurrently, leading to parallel processing across the pipeline's active objects (see Figure 3-11).

We used a chain of 4000 active objects that were distributed in equal-sized chunks over the cluster nodes. Then we executed a variable number of batched invocations on the leader object. The results for the Mega system are shown in Figure 5-6. Even for a small method granularity of 92μs, the speedup is significant as long as a sufficient number of batched calls are executed. Speedup gains are insignificant when the granularity is increased to 370μs. The reason is that most invocations are local and therefore the remote invocation overheads are amortised.

Figure 5-6 Pipeline Performance



5.5 Serialisation Performance

In this section we compare the serialization performance of SODA Serialisation with the overheads of the standard Java object serialization mechanism. For this purpose, we used a locally connected loopback stream provided by the Java API. Any data sent in through this stream is buffered and can be read out by another thread. We measured the passage time for various object types through this construct. This time includes object serialization and deserialization. We compared the performance of standard Java object serialization with SODA serialization. We experimented with various classes. These were an object containing 32 integer values and object with 4 integers and two 10-character Strings. Finally, we also tested a data structure that consisted of 15 objects arranged in a binary tree. Each of these objects had a private integer value. The results are shown in Table 5-3.

The last column of this table shows the start-up overhead for SODA fast serialization. This overhead is due to the cost of establishing the cache at the receiver side of the data transfer. For example, class names are cached and indexed the first time they are encountered. Objects that are marked as immutable can be cached by the stateful SODA serialization streams. This means, that after an initial deep-copy transfer of the object, every Future transfer is only an index value into the receiver-side cache.

The performance of SODA serialization is not as good as Ninja [181] Manta [172], or UKA-serialization [128;139]. The advantage of our approach, however is, that it does not require any modifications to the JVM or class libraries.

As Table 5-3 shows, the efficiency gain compared to standard Java serialization is still significant. Columns (a) and (b) show the values for sending a single instance using standard Java serialization and SODA serialization, respectively. The performance gains are a direct result of using explicit serialization/deserialization routines. For the complex 15-node tree object, SODA serialization time is actually slower than default Java serialization. However, further improvements can be achieved when several instances of the same class are sent repeatedly. In this case, the SODA Object streams can cache repeatedly sent information (such as class names). This leads to a further reduction in the average transfer time per instance as shown in column (c).

Table 5-3 Serialisation Overheads (Mega)

	Java Serialisation	**SODA Serialisation**	
	Single Instances (a)	**Single Instances (b)**	**Several Instances (c)**
32 int	**269 µs**	152 µs	**89.8 µs**
4 int, 2 String	**250 µs**	98.2 µs	**37.4 µs**
15-node tree	**629 µs**	1020 µs	**127 µs**
Immutable Object	**n/a (dep. on object size)**	n/a (dep. on object size)	**7.6 µs**

5.6 Guarded Method Invocation

In order to assess the runtime cost of invoking guarded methods, we use a bounded buffer active object. The buffer has a capacity of 10 cells and supports the guarded methods `put` and `get` as shown in Code 3-8. We instantiate one such buffer object and perform *n* `put` invocations, followed by *n* `get` invocations. The buffer accepts the first 10 `put` invocations, but then runs out of capacity and must delay the following *n*-10 invocations. Afterwards, *n* `get` requests arrive at the buffer. After each `get` request, buffer capacity becomes available and one of the pending `put` requests in the delay queue can be scheduled.

Table 5-4 shows the performance of the guard implementation in the SODA RTS. The measurements are based on a varying number *n* of `put`/`get` requests. As long as a guard expression evaluates to true, the extra overhead for guarded methods is limited to guard evaluation. This is the case for *n*=10; since the buffer capacity is never exhausted, all guards evaluate to true. The call overhead is therefore negligible compared to inlined calls.

For *n* > 10, *n*-10 `put` requests will be accumulated in the delay queue before they can be removed by corresponding `get`s. The current implementation is clearly not optimised for larger values of *n* as Table 5-4 shows. The reason is that *all* messages in the delay queue are re-evaluated, after a state change occurred to the active object. This is the case each time after a request, delayed request or reply is processed by the active object. A more optimised scheme would group together requests with identical guard conditions and only re-evaluate guards once per group.

In the current RTS, exceptions are used internally to signal invalid guards. When such an exception is caught, an attempted inlined call reverts to a local invocation: a request object with the call's parameters is created and queued in the object's delay queue. This involves extra synchronisation and thread context switching overhead as well as the creation of additional objects. These circumstances, together with the repeated re-evaluation of delayed method guards explain the overheads of methods that have invalid guards.

Table 5-4 Cost for Guarded Method Invocations.

`put/get` (*n*)	Max Delay Queue (*n*-10)	Avg Guard Evals /call	Total Time (*t*)	Time/Call (*t*/2*n*)
1000	990	233	499 ms	**249 µs**
500	490	121	130 ms	**130 µs**
250	240	56.0	33.9 ms	**67.8 µs**
125	115	22.0	9.54 ms	**38.2 µs**
60	50	11.2	2.79 ms	**23.2 µs**
20	10	3.55	571 µs	**14.3 µs**
15	5	3.00	359 µs	**12.0 µs**
11	1	1.05	151 µs	**6.87** µs
10	0	1.00	28.3 µs	**1.42 µs**

5.7 Conclusion

We presented a performance analysis for a set of benchmark programs designed to measure various basic operations of our prototype runtime system. The example programs yield real speed-ups over the best sequential algorithm even for method call granularities below the network latency. This is due to SODA's efficient transport service that bundles several requests and replies into network packages.

Due to the limits of the available platforms, scalability has only been examined up to a maximum of 40 hosts. However, there is no indication in the empirical results obtained or in the theoretical foundation, why appropriate SODA programs could not scale to much larger systems. One precondition for such scalability is the absence of program designs that require global lockstep processing, for example a central controller object. In this situation, communication with the controller will eventually become a bottleneck and outweigh the benefit of higher processing power. Therefore, high scalability can only be expected for loosely coupled programs which form "clusters" of active objects and only rarely engage in global communication.

Altogether, we conclude that the SODA programming model and implementation is a valid tool for parallel programming in a distributed memory, off-the-shelf environment. In particular, SODA's advantages become obvious for large-scale, object-oriented programs with complex patterns of interaction. Traditional approaches would make the management of parallelism increasingly complex, error-prone and ultimately inefficient. Performance trade-offs compared to more traditional parallel programming techniques become more and more negligible as the complexity of a program increases. Further evidence for this argument is given in Chapter 6 (VRML server). In order to gain more confidence in our findings and their relevance for real-world applications, we implement a large-scale application on top of SODA in this chapter. The communication structure and computational requirements of this application are unpredictable at compile-time. It will therefore demonstrate the scalability of the programming model as well as scalability of the prototype implementation for a real-world application.

Chapter 6

Scalable VRML Execution in SODA

The primary goal of this chapter is to evaluate SODA's usability in the light of a real-world application. To amortise SODA's overhead, an appropriate target application must be sufficiently complex in terms of communication and module structure. In this chapter we present the design and implementation of a scalable, parallel VRML server that has been built on top of SODA. This server can support large-scale VRML worlds and through efficient information filtering techniques make these accessible to low-powered clients. Parallelism is exploited in the VRML execution model as well as in the client-server communication.

6.1 Overview

SODA is geared towards the support of object-oriented applications with irregular communication structure and modular architecture. Traditional test suites for parallel programming prototypes would not capture SODA's benefits, since they are commonly based on regular, numerical transformations with a rather simple program structure. Low-level tools, such as PVM or MPI are more appropriate for this domain, since they allow an extremely efficient mapping onto the physical hardware.

In comparison, SODA induces extra performance overheads due to its high degree of abstraction over the hardware. These overheads can only be justified in the context of a large-scale application with complex communication patterns between its constituent modules. SODA's ease-of-use advantages then outweigh the loss in runtime performance. An appropriate choice of example application is therefore vital for a successful demonstration of SODA's benefits. For this purpose we have chosen to implement a parallel execution engine for VRML97 on top of SODA.

In this chapter we give some background information on VRML97; opportunities to exploit parallelism in the underlying execution model are identified. Finally, we describe a mapping of the parallel VRML execution model onto SODA. The VRML-related findings presented in this chapter are relevant in their own right. The examination of parallelism in VRML is novel and enables the scalability of VRML to larger and more dynamic worlds. This overcomes a set of limitations currently hindering VRML. Appendix A gives further background information on related work.

6.1.1 Problem Statement

The Virtual Reality Modelling Language (VRML) [35;36] is a platform-independent and standards-based *scene description language*. It was designed to allow "*3D worlds*" to be delivered over the Internet. VRML files are analogous to HTML in the sense that they are simple ASCII text files, which are interpreted by *browsers*. A VRML browser parses the VRML file and delivers an audio-visual rendering of the world. Typically, VRML browsers make a set of *navigation paradigms* available, which describe physical constraints, such as degrees of freedom, on the user's movement. For example, a "walk" paradigm would allow two-dimensional world exploration, binding a user to the ground with simulated gravity. Within the limitations of the active navigation paradigm, users can freely explore a world, zoom in and out, move around and interact with the virtual environment. Rendering of VRML content occurs in real-time according to the user's point of view. Graphical quality is therefore lower than what can be achieved with pre-processing techniques, such as ray-tracing or radiosity solutions. For example, shading is typically based on flat- or Gouraud-shading.

The first version of VRML provided only static and non-interactive worlds, but the later VRML97 ISO standard supports "moving worlds" that are responsive to both user interaction and the passage of time. It is therefore possible to envisage the creation of huge, complex worlds with thousands of interacting users. For example, models of cities could be built to include moving vehicles as well as static buildings. In the future, with advances in traffic sensing technology, it may even be possible to build models of real cities that show accurate traffic flows in real-time.

A primary design aim for VRML was the delivery of worlds over the Internet. Conventional web servers can be used to host a world description, which is then downloaded into the VRML browser. Despite the obvious potential, however, VRML worlds available on the Internet have so far been relatively limited in their behavioural and geometrical complexity (dynamic and static complexity). The reason can be attributed to the monolithic nature of the VRML usage model. All VRML content is downloaded to the browser and the browser carries the sole responsibility for audio-visual rendering. In addition, the browser must also perform *behaviour evaluation* necessary to drive the dynamics of a world, for example a physics-based simulation. A set of scalability limitations arises from this conventional usage model:

Bandwidth Requirements. Downloading the world description into the VRML browser takes a long time for large worlds. Even if the user ever only sees a small fraction of the world, the complete world description must be downloaded to the client. Available network bandwidth therefore limits world size; this is of particular significance for mobile devices, such as wirelessly connected PDAs. VRML addresses download time through the partition of a world over separate files, which can be downloaded individually.[29] However, this is insufficient, since behaviours cannot span multiple files.

Memory demands. Once downloaded, the client must store the world's geometrical state locally. The possible extent of a world is therefore limited by the client-side available memory size.

Parsing and Rendering time. Once downloaded, the VRML browser is responsible for parsing the VRML file and performing the audio-visual presentation of the world. Rendering can be prohibitively expensive for large scenes, despite the culling optimizations performed by most VRML browsers.

Behaviour execution costs. The browser must continually update the state of the world–as objects move and the user interacts–and also render a view of the world in the browser window. If processing power is insufficient for behaviour evaluation, this will slow down the client frame rate, resulting in a low-fidelity presentation.

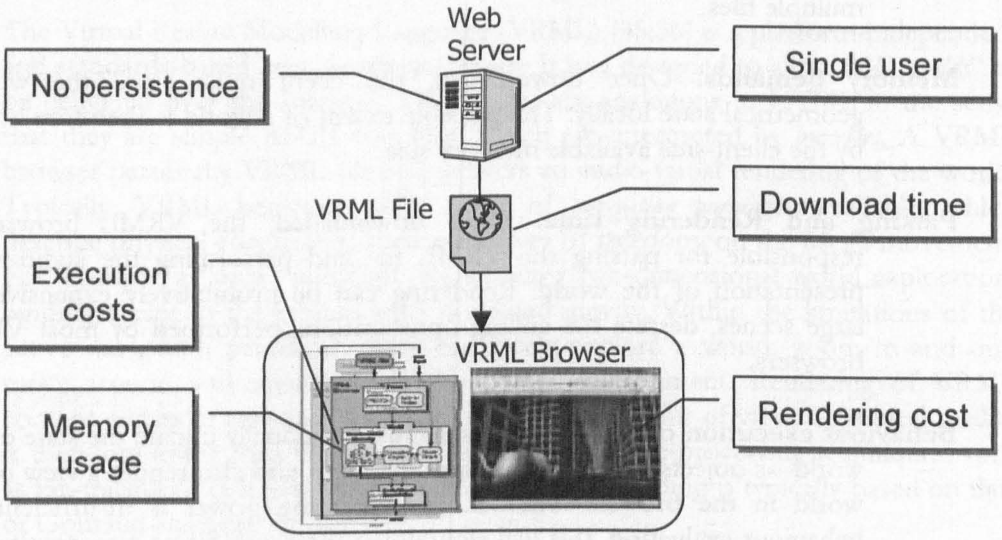
Multi-user interaction and persistence of changes. Because the world runs locally–in the user's browser–there is no possibility of interaction between different users in the same world. This precludes both, direct interaction between users who have visible contact, but also indirect interaction. For instance, one user cannot permanently introduce a new object in the world to be seen by a later passer-by. The VRML usage model does not allow persistence of state: a world reverts to its initial state every time it is downloaded.

External updates. It is difficult to arrange for the world to change in response to external events. For example, if a virtual world models the current state of part of the real world (e.g., traffic flow in a city, footballers on a pitch) then we would wish to move objects in the virtual world to reflect real-time changes in the real world.[30]

[29] As controlled by Anchor, Inline and LOD nodes.

[30] [20] points out another limitation in VRML, which is the limited precision of 32-bit coordinate values. However, this does not influence the generality of the approach presented here, since it would be easy to support 64-bit values instead.

Figure 6-1 Conventional usage model.



6.1.2 A Client-Server based Usage Model for VRML

This chapter presents the design of a system that has been built with the aim of directly addressing the above problems, and so supporting huge, active worlds filled with large numbers of interacting users. Our design provides a novel usage model for VRML that is based on a client-server paradigm. In this usage model, the server assumes a much more active role, maintaining the evolving state of the world and communicating bi-directionally with the client. This is a major deviation from the conventional usage model, where the web server remains essentially passive, once a VRML file has been downloaded to the VRML browser.

In the new usage model, all world state is held at the server, encapsulated as values of VRML object attributes. Behaviour evaluation on the server-side affects the state of these attributes. The server expends the computational effort required for the evaluation of behaviours. Clients, relieved from the task of computing the world dynamics, can therefore expend more resources towards the rendering of a view onto the world.

The design decision to leave rendering on the client side is motivated by two factors. Firstly, it makes sense to perform rendering in hardware. In recent years, high-performance graphics cards have become commonplace in desktop PCs. We therefore reduce server load. Secondly, we assume that the geometrical description of a client's area-of-interest (see the following paragraph) is more compact than equivalent bitmapped frames. This is a measure to conserve bandwidth consumption compared to rendering frames on the server and sending these to the client.

It would be inefficient to dispense exhaustive information on all world objects to the client. In the real world, perceptual limitations reduce our visual awareness to the near-by spatial area. This concept of limited *spatial awareness* can be exploited in computer-generated worlds and is a fundamental component of most networked virtual environment systems. In a sufficiently large virtual world, most objects are beyond the client's visibility or interest range. Therefore, a client can cull objects beyond its *area-of-interest* (AOI)[31], without significantly impeding visual perception.

Our client-server protocol allows an incremental download of objects. This is useful so as to dynamically manage a client's area-of-interest and to add and subtract objects from it as required. Download occurs only for objects that intersect with the client's area-of-interest. This can be triggered by two conditions: either a client moves about and new, stationary objects become visible; or an object autonomously changes its position and therefore moves into the client's visibility range. When objects leave the client's AOI they are removed from the client-side partial world replica.

Beyond client AOI management, further exchange of information between client and server is required. This is necessary for the following (see also Figure 6-16):

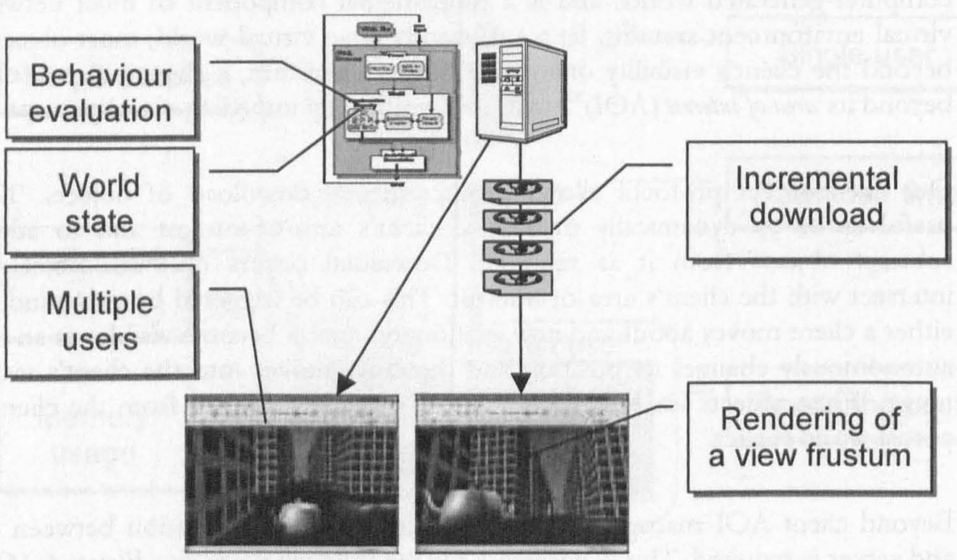
AOI Replication is required to update the client-specific AOI fraction of the server-side world state. This allows a client to participate in the dynamic state of a large world without being aware of the world's overall extent.

Notification Messages enable the propagation of client interaction to the server. For example, a notification message would be sent when the client presses a button in the client-view of the world. This event can then be fed into the server-side behaviour evaluation and it can also be made available to other clients.

Frustum Updates are sent from client to server when the client's area-of-interest changes. Typically, this would occur as a result of the client moving or rotating.

[31] According to the aura/nimbus model [21;73] the area-of-interest is dependent on the medium type. For example, while objects behind a viewer are not visible, audio sources can be perceived. In this case study we are only interested in the visual medium. Therefore, we define the AOI as the client's view frustum onto the geometric content in the world.

Figure 6-2 Client-server based usage model.



6.1.3 Summary

In summary, the proposed usage model with its decoupling of rendering and behaviour execution can overcome the limitations inherent to the conventional, monolithic VRML usage model:

Bandwidth Requirements. Download time is reduced to the fraction of the world that lies within the client's **area-of-interest** (AOI). Relevant geometry is downloaded to the client on a fine-grained per-object basis. Further improvements are possible by making an object available at multiple **levels-of-detail** (LOD)[32].

Our design requires replication of the area-of-interest state from the client to the server. The frequency with which AOI replication occurs is affected by the available network bandwidth and latency between client and server. If network quality is insufficient, the AOI replication rate will be reduced. This will be lead to noticeable visual artefacts, in particular for objects that undergo continuous changes, such as a car travelling along a road. This problem is well-known and can successfully be addressed by *dead reckoning* techniques. However, for simplicity, dead reckoning is not implemented in our system.

Memory demands and **Parsing and Rendering time**. The client's knowledge of a world is restricted to objects within its area-of-interest at any given time. Moreover, few of these objects will be available at the highest level of

[32] The level-of-detail technique [145] is based on the observation, that in the real physical world, objects far away from a viewer are perceivable in less geometric detail than objects close by. This circumstance is exploited in virtual environments by providing low-detail versions of distant objects without significantly compromising their visual appearance.

detail. This greatly reduces the amount of geometrical information known to the client. The benefits are threefold: Reduction of parsing time for VRML objects, reduction of information that is sent down the rendering pipeline and reduction of memory consumption on the client. By modulating their area-of-interest, low-powered clients can participate in complex worlds; the server is responsible for tailoring a view adapted to their capabilities (see next section).

Behaviour execution costs. The server carries the bulk of behaviour execution costs. If resultant changes affect a given client's area-of-interest they are propagated to this client's AOI. The replication frequency adapts to available network latency and bandwidth.

Multi-user interaction and persistence of changes. The server maintains all world state. Driven by user interaction and behaviour execution this state evolves over time. Multiple clients can connect concurrently and interact in a world; the presence of other clients is conveyed through avatars.

External updates. The server-side behaviour evaluation may sample the readings of real-world sensors in order to reflect part of the state of the real world within the virtual world.

6.1.4 System Scalability

Our system leads to a concentration of activities at the server. This includes world behaviour execution, maintenance of world state and area-of-interest filtering and replication for connected clients. As a result of this architecture, the server can easily become overloaded when supporting large worlds with many users and complex behaviours. We do not want to just move the bottleneck from the client to the server. In order to achieve system scalability, our server is implemented as a SODA parallel program

This consideration is the justification for implementing our server as a parallel program, based on the SODA programming system. Parallelism is exploited in the following areas:

Behaviour computation according to the VRML97 execution model is performed in parallel. We examine in §6.2.2 how parallelism can be exploited while remaining compliant with the VRML97 specification.

Area-of-interest management amongst multiple clients is performed in parallel. For every client a dedicated *client-proxy* object performs the task of maintaining a view on the world, tailored to the client's performance characteristics and fidelity requirements.

Due to the abstraction layer provided by SODA, the VRML server can run across a workstation cluster or other distributed memory architecture. The benefit of this centralised, scalable VRML execution is that much larger and more dynamic worlds can be generated. The cluster's overall memory and computational performance is available for maintaining and updating the evolving world state.

The VRML specification does not encompass low-powered clients. VRML does not scale to the combined requirements of small screen size and limited processor, graphics and network performance. The limitations of the VRML usage model become exacerbated. By contrast, the scalability of our usage model extends to devices, such as wirelessly connected PDAs. In our usage model the following measures are available to compensate for limited client-side resources:

Area-of-interest Modulation. An effective means to shrink the number of objects known to a given client lies in the contraction of this client's area-of-interest. This reduces client load and bandwidth consumption at the expense of a reduced number of visible objects.

Level-of-detail degradation. To further conserve bandwidth, the client can request sub-optimal levels of detail. The trade-off here is a less-detailed object presentation.

Replication frequency adaptation. As noted in §6.1.3 (1), AOI changes are propagated from server to client at a variable replication frequency. This allows adaptation to client power and networking quality as a fidelity trade-off.

6.2 Parallel VRML Execution

This section describes potential parallelism in the VRML execution model, which can be extracted in full compliance with the VRML97 specification [124;147;148]. A mapping of this new, parallelised execution model onto SODA active objects is described. Additional parallelism can be obtained between multiple client-proxies (see §6.3).

6.2.1 Fundamental VRML97 Concepts

We begin with a brief overview of the VRML97 specification, pertaining to static and dynamic world description. This includes a discussion of VRML's scene graph structure and its event execution model.

Scene Graph Structure

VRML comes with a pre-defined set of building blocks that are fundamental to all worlds. These building blocks, or ***nodes***, serve as abstractions for a variety of real-world objects and concepts. For example, node types exist that describe simple 3D geometry, sound data, a light source description, a JPEG image, and so on. Nodes are defined by their type and their *field* values. The type is a name, such as `Box`, `Color`, `Group`, `Sphere`, `Sound`, `SpotLight` and so on. Field values define a node's state and they distinguish node instances of the same type. For example, each

Sphere node might have a different radius, and different spotlights may have different intensities, colours and locations.

The acceptable fields for a node are defined in a ***node specification.*** For each field, a name, type and default value are supplied. The default value is used if a value for the field is not specified in the VRML file instantiating the node. Figure 6-3 to Figure 6-5 show some examples of node specifications and their instantiation. An analogy can be drawn between VRML nodes and object-orientation concepts. Node specifications and their instances are analogous to classes and objects. Fields in a node specification correspond to member variables in a class declaration.

These examples reveal a variety of field types, such as 3D float vectors, Boolean values, float scalars, and quaternion rotations. The `SF` and `MF` prefixes distinguish between fields that contain single-field and multi-field types[33]. The `SFNode/MFNode` type can reference other nodes in the scene. This self-reference type is fundamental to the structural description of the *scene graph*. This is a directed, acyclic graph which binds all nodes in a world into a hierarchy. A node's position in the scene graph defines its scope and describes which child nodes are influenced by any of its transformations.

Figure 6-3 Specification and Example Instantiation of the Box node

```
Box {                                 | Box {
  field     SFVec3f size   2 2 2      |   size 1 1 1
}                                     | }
```

The Box node specifies a rectangular parallelepiped box in the local coordinate system centered at (0,0,0) in the local coordinate system and aligned with the coordinate axes. By default, the box measures 2 units in each dimension, from -1 to +1. The Box's size field specifies the extents of the box along the X, Y, and Z axes respectively and must be greater than 0.0.

[33] Single-field types contain scalar values, multi-field types are arrays.

Figure 6-4 Specification and Example Instantiation of the Cylinder node

```
Cylinder {                                       Cylinder {
  field    SFBool     bottom   TRUE                height 2.0
  field    SFFloat    height   2                   radius 1.5
  field    SFFloat    radius   1                 }
  field    SFBool     side     TRUE
  field    SFBool     top      TRUE              # default values are used for
}                                                # the omitted fields.
```

The Cylinder node specifies a capped cylinder centred at (0,0,0) in the local coordinate system and with a central axis oriented along the local Y-axis. By default, the cylinder is sized at -1 to +1 in all three dimensions. The radius field specifies the cylinder's radius and the height field specifies the cylinder's height along the central axis. Both radius and height must be greater than 0.0.

The cylinder has three parts: the side, the top `(Y = +height)` and the `bottom (Y = -height)`. Each part has an associated `SFBool` field that indicates whether the part exists (`TRUE`) or does not exist (`FALSE`). If the parts do not exist, they are not considered during collision detection.

Figure 6-5 Specification of the Transform node

```
Transform {
  eventIn       MFNode       addChildren
  eventIn       MFNode       removeChildren
  exposedField  SFVec3f      center            0 0 0
  exposedField  MFNode       children          []
  exposedField  SFRotation   rotation          0 0 1  0
  exposedField  SFVec3f      scale             1 1 1
  exposedField  SFRotation   scaleOrientation  0 0 1  0
  exposedField  SFVec3f      translation       0 0 0
  field         SFVec3f      bboxCenter        0 0 0
  field         SFVec3f      bboxSize          -1 -1 -1
}
```

A Transform is a grouping node that defines a coordinate system for its children that is relative to the coordinate systems of its parents. The *bboxCenter* and *bboxSize* fields specify a bounding box that encloses the Transform's children. The *translation*, *rotation*, *scale*, *scaleOrientation* and *center* fields define a geometric 3D transformation consisting of (in order) a (possibly) non-uniform scale about an arbitrary point, a rotation about an arbitrary point and axis, and a translation. The *center* field specifies a translation offset from the local coordinate system's origin, (0,0,0). The *rotation* field specifies a rotation of the coordinate system. The *scale* field specifies a non-uniform scale of the coordinate system - *scale* values must be >= 0.0. The *scaleOrientation* specifies a rotation of the coordinate system before the scale (to specify scales in arbitrary orientations). The *scaleOrientation* applies only to the scale operation. The *translation* field specifies a translation to the coordinate system.

VRML97 defines more than 50 built-in node types; further types can be defined through the PROTO extension mechanism (see section 6.3.3). Important for the static description of a world are *geometry nodes* and *grouping nodes*:

Geometry nodes define visible artefacts in a scene. Examples are the `Box`, `Cone`, `Cylinder`, `Sphere`, `ElevationGrid`, `IndexedFaceSet`, `Extrusion` and

`Text` nodes. In order to become visible, geometry nodes must be embedded in a `Shape` node that describes the geometry's visual appearance, such as texture, material or reflectivity. Figure 6-6 shows a simple VRML file and the visual result when viewed with a VRML browser. A material with default field values is applied to the surface, which renders as uniform matt grey colour.

Grouping nodes do not render visible results; however, they describe transformations for a set of nested children nodes within their scope. Every grouping node has a `MFNode` children field that contains all associated children nodes. For example, the `Transform` node defines a coordinate system for its children nodes while the `Anchor` node attaches a hyperlink to its children. Figure 6-7 shows a more complex VRML scene graph. Various `Transform` nodes are used to position and scope nested geometry nodes.

Figure 6-6 A simple VRML file.

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material { }
  }
  geometry Cylinder {
    height 2.0
    radius 1.5
  }
}
```

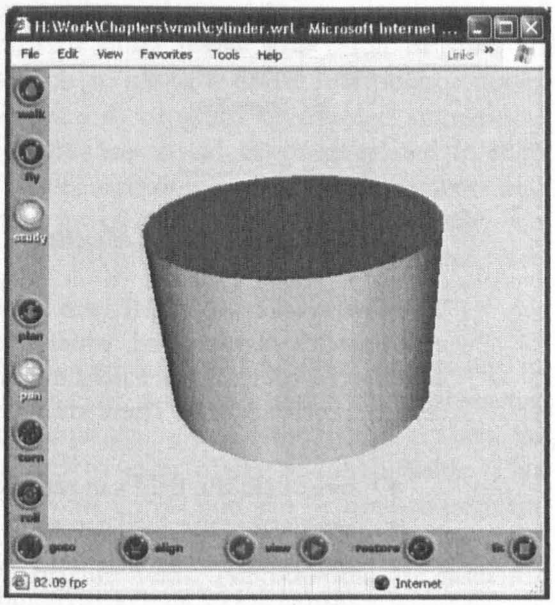
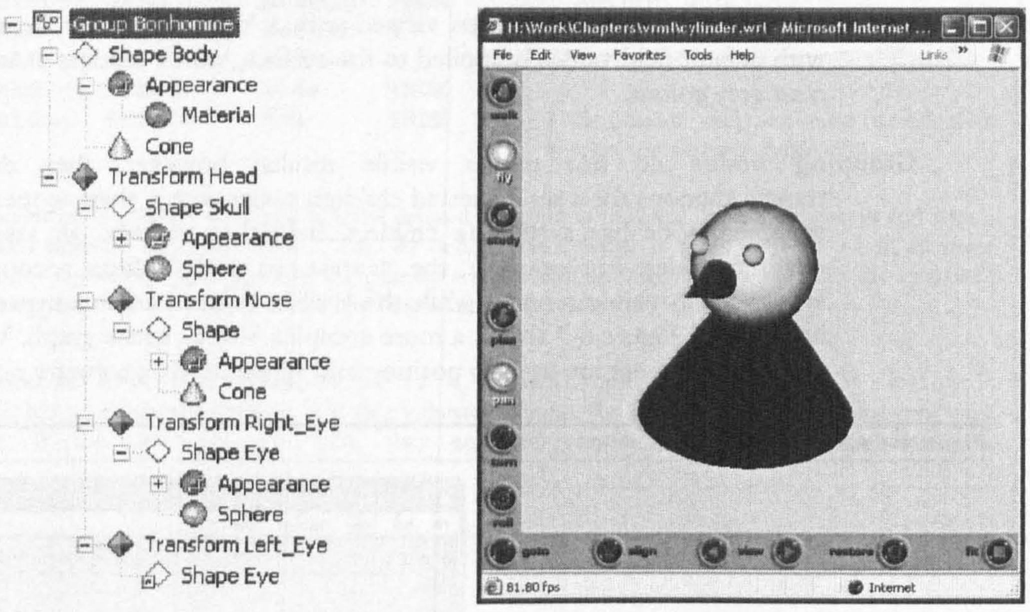

H:\Work\Chapters\vrml\cylinder.wrl - Microsoft Internet ...
File Edit View Favorites Tools Help
Links
walk
fly
study
plan
pan
turn
roll
goto
align
view
restore
fit
82.09 fps
Internet

Figure 6-7 Several Shape nodes and Transform nodes.
The head consists of three spheres and one cone. Field names are not shown in the scene graph.



The VRML Execution Model

The first version of VRML was limited to purely static worlds. With VRML97 an event model was designed, which allows VRML worlds to send and receive events along given routes. Every field has an attribute that controls whether the node can send or receive events. There are four types of attributes:

- **`eventIn`**: the field can receive events; event consumer.
- **`eventOut`**: the field can send events; event producer.
- **`exposedField`**: the field can receive and send events; combines the properties of an `eventIn` and `eventOut` field.
- **`field`**: the field can neither receive nor send events. Its value is constant.

An `eventOut` field of one node may be *routed* to an `eventIn` field of another node with matching type. A route propagates any value-change of the `eventOut` field in the form of an event to the connected `eventIn` field. This affects the state of the receiving node. In addition, event reception may trigger off some internal processing at the receiver. The nature of such activity is dependent on the node's type. The total of all `eventIn-eventOut` routes in a world is called the *routing graph*.

The routing graph mediates one-way event notification between the nodes in a scene, as governed by the VRML execution model. An event is produced whenever an `eventOut` field's value changes. The event is a tuple, containing the `eventOut` field's new value and a timestamp. The timestamp indicates the event's generation time (see event cascades below).

The following VRML97 node types act as controllers for the dynamics of a VRML world:

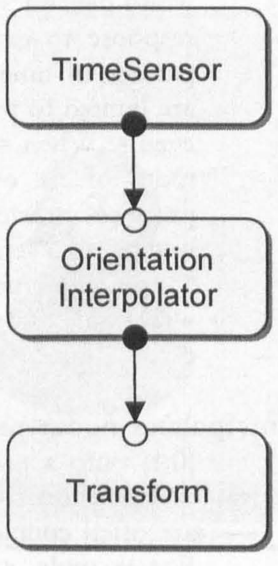
Sensor nodes take a unique role in the VRML execution model as drivers of all world dynamics. They have the ability to spontaneously generate events in response to external stimuli. When triggered by user interaction or the passage of time, Sensors act as initial event generators. Non-Sensor node are limited to producing secondary events in response to incoming Sensor events. When stimulated, a Sensor node dispatches an event on one or more of its `eventOut` fields. For example, the `TouchSensor` node produces an event whenever it is "clicked" by a user. Other types of sensor nodes are sensitive to mouse drag, user proximity, visibility, etc. `TimeSensor` nodes are triggered according to the passage of simulated world time. For example, a `TimeSensor` can be used to generate events in regular time intervals or *continuously*, i.e. with every rendering frame.

Interpolator nodes provide linear interpolation of a scalar value in the interval [0;1[onto a positional, rotational or other value. The result of such an interpolation can then be passed on to another node. Interpolator nodes are often coupled with `TimeSensors` to program key-framed animations. For example, a `PositionInterpolator` could be programmed to map scalar values produced by the TimeSensor onto a set of line segments and therefore describe an object's movement over time. Similarly, an `OrientationInterpolator` can progressively rotate an object (see complete example in Figure 6-8).

Script nodes. Interpolator nodes perform only fine-grained event processing, in the form of linear interpolation. `Script` nodes are much more powerful, since they allow arbitrary, programmatic event processing: A Java or JavaScript method can be attached to each of its `eventIn` fields. This method is executed whenever an event arrives and can be used to perform complex world update logic. The results of such processing can be made available on the Script node's `eventOut` fields. The potential use cases for Script nodes are virtually unlimited. For example, a `Script` may drive a physics-based simulation, contain some artificial intelligence algorithm, sample external instruments, or perform database queries, e.g., to access data for knowledge visualisation [114]. The disadvantage to the `Script` nodes' flexibility is that they can incur high processing costs and therefore slow down rendering frame rates.

Figure 6-8 Simple key-framed animation in VRML.
This rotates the Transform node and all its children geometry every 20 seconds.

```
DEF Geometry Transform {
     rotation  0 1 0  0
     children [ ..... ]
   }

   DEF Sensor TimeSensor {
     cycleInterval         20
     .......
   }

DEF Interpolator OrientationInterpolator
{
   key              [ 0 0.5 1 ]
   keyValue         [ 0 1 0  0,
                      0 1 0  3.1416,
                      0 1 0  6.2832]
   }

   ROUTE Sensor.fraction_changed
     TO Interpolator.set_fraction
   ROUTE Interpolator.value_changed
     TO Geometry.rotation
```

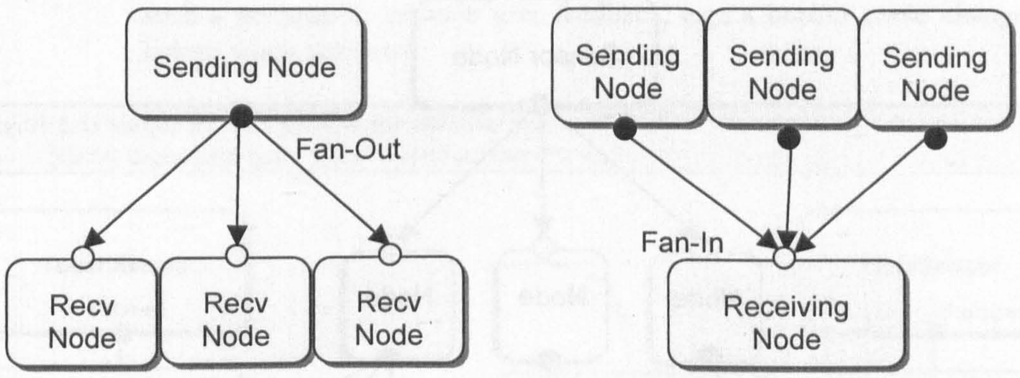


Since the scene graph structure is described via `MFNode` or `SFNode` fields, a dynamic modification of the scene graph is possible. For example, the set of children nodes for a `Transform` node is exposed via the node's `children` field (see Figure 6-5). `addChildren` and `removeChildren` are "convenience" `eventIns` provided to successively add and remove children nodes.

Event Cascades

The routing graph provides the communications backbone for a VRML world. Events flow along the edges of the routing graph, from `eventOut` fields to `eventIn` fields. If a single `eventOut` is connected to multiple `eventIns`, this is called a *fan-out*. As a consequence, events produced by the `eventOut` field are replicated to all connected receivers: the receiving `eventIns` will take on identical values when the `eventOut` fires. Similarly, an `eventIn` may be the destination for more than one incoming route. Such an `eventIn` is the destination of a *fan-in* (see Figure 6-9).

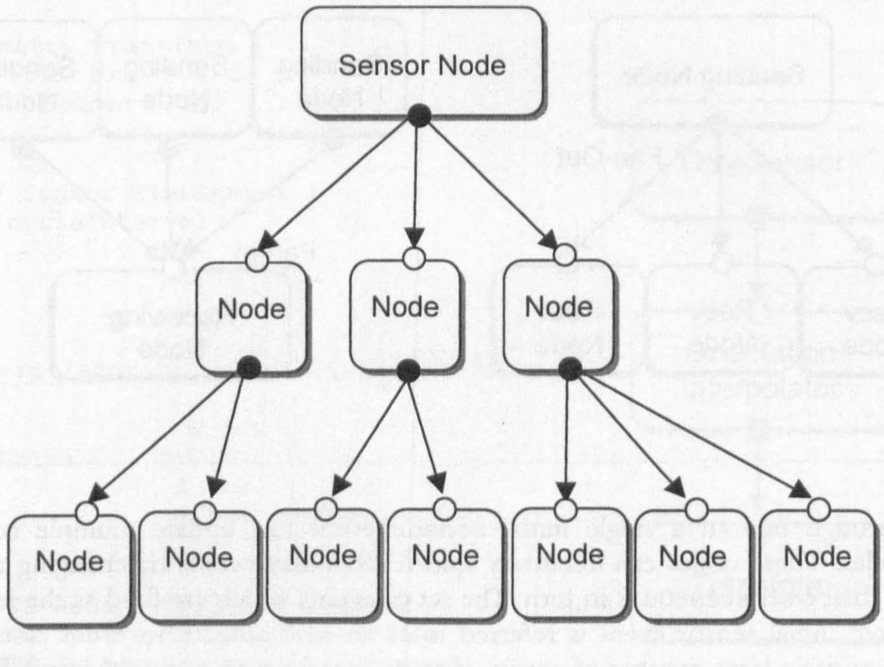
Figure 6-9 Fan-out and fan-in routing configurations.



Through fan-out a single initial `Sensor` event can update multiple connected nodes. These nodes can iteratively spawn secondary events by changing the value of their own `eventOuts` in turn. The set of events which are fired as the result of a given initial sensor event is referred to as an *event cascade.* An event cascade may comprise a large number of events, if many fan-out routes are traversed. All events in an event cascade are associated with the initial `Sensor` event.

Event cascades with many fan-outs and complex processing within nodes can induce significant activity: a large subset of the routing graph's edges may fire and trigger event processing throughout the scene graph. The processing burden on the VRML browser can be significant, especially if performance-intensive Script nodes are involved. In the conventional execution model a world designer must consequently take care to put a ceiling on the complexity and number of Script nodes in a world in order not to overload the VRML browser.

Figure 6-10 An example event cascade



Discrete and Continuous Events

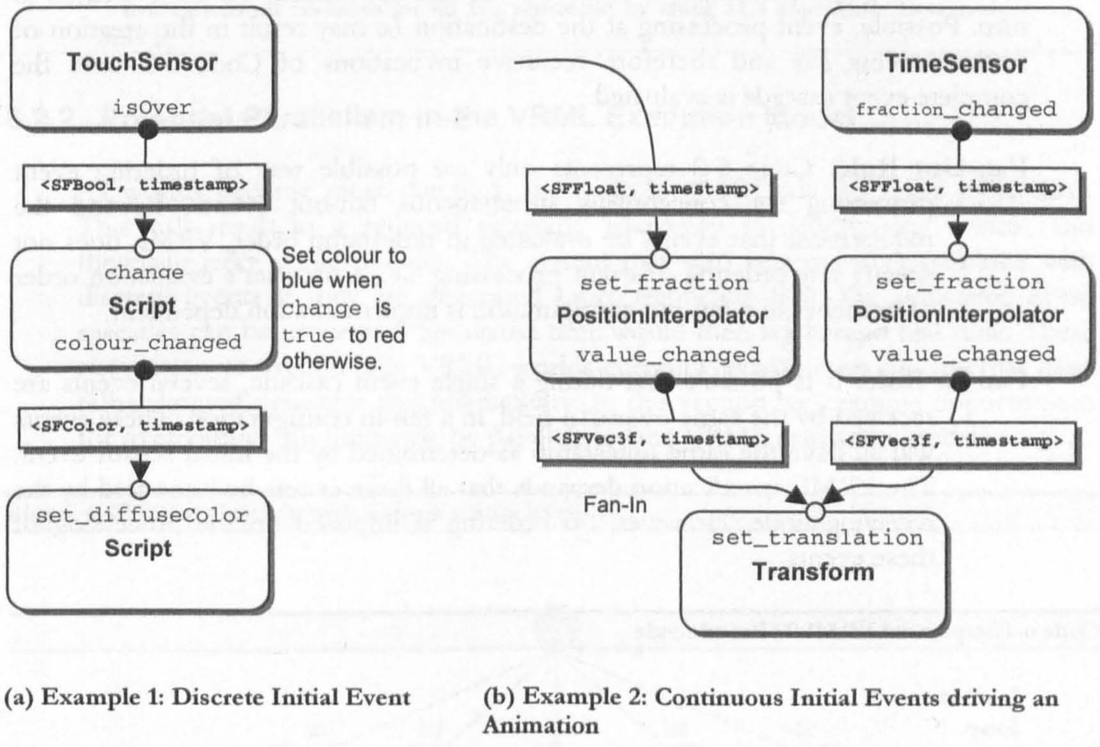
In the VRML execution model all events in an event cascade are considered to occur simultaneously. Therefore they carry the same timestamp as the initial Sensor event that triggered the event cascade. This timestamp is used to prevent infinite loops in the routing graph: a *loop-breaking rule* prevents cycles by limiting each `eventOut` field to a single firing per timestamp. i.e., an event can fire at most once per event cascade.

Most events produced during world execution are *discrete*: they happen at well-defined world times, e.g. as determined by the time of user interaction. However, `TimeSensor` nodes also have the capability to model continuous changes over time: A browser generates sampling events on the `fraction_changed` and `time` `eventOut` fields of `TimeSensors`. The sampling frequency is implementation dependent, but typically samples would be produced once per frame, e.g., once for every rendering of the user's view on the world.

The VRML specification also requires that continuous changes be up-to-date during the processing of discrete events. i.e., continuous changes that are occurring at the discrete event's timestamp shall behave as if they generate events at that same timestamp" ([36], §4.11.3.).

Example 1. Figure 6-11(a) depicts a simple event cascade. The `TouchSensor`'s `isOver eventOut` sends `<true, touchTime>` when the user moves the pointing device over the associated geometry and `<false, retractTime>` upon retraction. These events are routed to a `Script` node, which performs author-defined event processing, in this example resulting in colour value being sent to a `Material` node. A world author might employ such a scenario to provide user feedback, e.g., a button could change its colour when activated.

Figure 6-11 Simple Event Cascades for different Sensor Events
(circles depict field types: filled↔eventOut, empty↔eventIn).



Example 2. The `TimeSensor` in Figure 6-11(b) produces continuous events containing a number in the range [0; 1[on its `fraction_changed` field with the passage of time. These continuous events are passed to a `PositionInterpolator` that animates the `translation` vector of a `Transform` node. This is an example of a linear key-framed animation; continuous events are typically sampled once per rendering timeframe. A fan-in situation can arise in this example for the `Transform` node, if both `PositionInterpolators` send events with identical timestamp.

Sequential Implementation

Code 6-1 shows the pseudo-code algorithm of a typical VRML97 browser. If no discrete events are scheduled, continuous events are sampled as quickly as possible. The sampling frequency is influenced by hardware capabilities.[34] This event evaluation is alternated with frame rendering of the new geometric layout.

[34] Some VRML browser implementations allow the definition of a target frame rate in order to conserve CPU power.

Scheduled discrete events force the evaluation of all continuous events at that same time (see up-to-date requirement above). If any discrete events have not yet been evaluated, no rendering takes place.

Code 6-2 shows the evaluation of the event cascade for each initial Sensor event C_i or D_i (mapped to E). The loop breaking rule prohibits cyclic loops by limiting each `eventOut` to at most one event per timestamp. Otherwise, R_O contains all edges of the routing graph pointing out of E. R_Os fan-out destinations In_i are evaluated in turn. Possibly, event processing at the destination In_i may result in the creation of further events E_{Oij} and therefore recursive invocations of Code 6-2 until the complete event cascade is evaluated.

Fan-Out Rule. Code 6-2 represents only *one* possible way of ordering event processing for conceptually simultaneous fan-out events. Beyond the requirement that events be evaluated in timestamp order, VRML does not specify any ordering of event processing. i.e., a browser's evaluation order of branches in a fan-out configuration is implementation dependent.

Fan-In Rule. It is possible that during a single event cascade, several events are received by the same `eventIn` field, in a fan-in configuration. These events will all have the same timestamp as determined by the initial sensor event. The VRML specification demands that all these events be honoured by the receiving node. However, no ordering is imposed on the processing of these events.

Code 6-1 Sequential VRML97 Pseudocode

```
lasttime ← 0;
loop
   now ← Browser.getWorldTime();
   if any discrete sensor eventOuts $S_i$ scheduled with $lasttime < t_{E_i} \leq now$, e.g.,
   asynchronous user input, or finished TimeSensor cycle then
      $t_D$ ← time of most imminent $S_i$;
      $D \leftarrow \{D_j | t_{D_j} = t_D\}$;
      C ← sample of all continuous eventOuts at time $t_D$;
      evaluate event cascade for each $C_i \in C$;              /*algorithm 2*/
      evaluate event cascades for each $D_j \in D$;             /*algorithm 2*/
      lasttime = $t_D$;
   else
      C ← continuous events sampled from all active and enabled TimeSensors at
      time now;
      evaluate event cascades for each $C_i \in C$;             /*algorithm 2*/
      lasttime = now;
      rendering of the new geometric world layout;
   end if
end loop
```

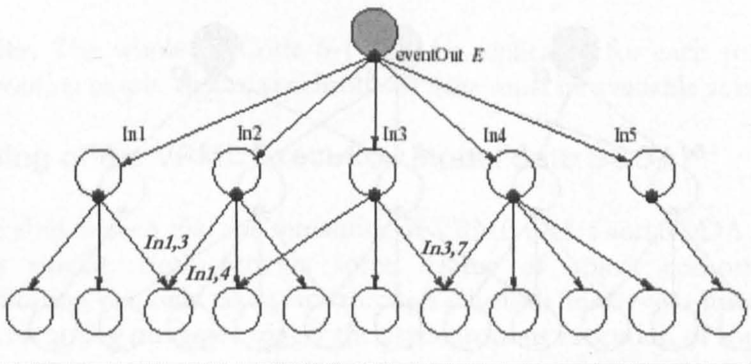
Code 6-2 Event Cascade Evaluation for a sensor Event E

```
if eventOut $E$ has already 'fired' for time $t_E$ then
    stop; $\boxed{\text{loop breaking rule}}$
else
    $R' \leftarrow \{(Out, In_i) \subset R | Out = E\}$
    process all $In_i$, potentially generating a set of new events $E'_{ij}$ for each $In_i$;
    evaluate event cascades for all $E'_{ij}$ produced by using this algorithm recursively;
end if
```

6.2.2 Potential Parallelism in the VRML Execution Model

As worlds become more complex, the main loop of Code 6-1 takes more time. This will result in a reduced sampling frequency for continuous events, and therefore jerky scene updates. The system may also become over-saturated with discrete events if they are generated more frequently than the associated event cascades can be processed. Simulated time would then lag behind real time. These circumstances explain why VRML worlds currently available on the Internet have rather limited dynamics and interactivity. In this section we examine opportunities for overcoming this limitation by parallelising the VRML execution model.

Figure 6-12 Event Cascade with a single initial Event E.



Parallelism within a Single Event Cascade

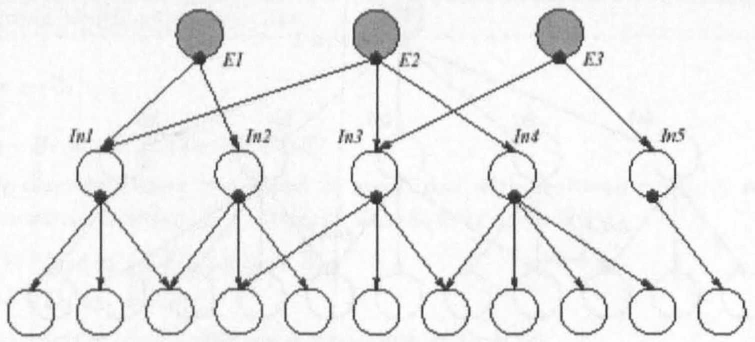
The VRML97 specification does not impose an execution order for the branches of a fan-out configuration (Fan-Out Rule). In Code 6-2, if a single initial sensor event E has a fan-out configuration, all `eventIn` fields *Ini* linked to it can be processed in parallel (see Figure 6-12). Recursive fan-out configurations in an event cascade can lead to a high degree of potential parallelism. The grain size is only determined by the complexity of event processing in the participating nodes. Parallelism can be exploited without affecting VRML97 semantics: Nodes can only communicate via event notification; otherwise they are isolated from each other. Therefore no undesirable interference can occur between two execution paths in an event cascade.

A restriction of parallelism is necessary when the event cascade contains fan-in configurations. During fan-in, two events of the same event cascade (therefore with identical timestamps) arrive at a single `eventIn` (for example $In_{1,3}$, $In_{1,4}$, and $In_{3,7}$ in Figure 6-12). The Fan-In Rule specifies that event processing must occur in some sequential order. In a parallel implementation, some form of synchronisation is therefore necessary. For example, incoming events could be queued in a buffer for sequential processing. The semantics of VRML nodes are therefore very similar to SODA active objects.

Parallelism between Event Cascades

Several event cascades that occur at identical times can be executed in parallel. For example, continuous events might be produced for a set of sensor nodes. Any interference between the branches of these initial events (e.g., In_1, In_3 in Figure 6-13) is governed by the VRML event model: Several initial sensor events that are scheduled with the same timestamp are treated by VRML as if they were members of the same event cascade. Fan-ins of events with the same timestamp are handled according to the Fan-In Rule (e.g., all events must be processed, but the ordering of event processing is implementation-dependent). VRML's loop breaking rule is applied to prevent multiple writes to a single `eventOut` field. All events D_j and C_i scheduled in the main loop of Algorithm 1 can therefore be evaluated in parallel[35].

Figure 6-13 Several Event Cascades with Initial Events E_i, that all have the same timestamp t_E.



Routing Graph Partition

One would expect most large and complex worlds to be built from relatively simple, localised and autonomous behavioural units (e.g., a conversation amongst some people somewhere, a physical simulation somewhere else). Such units would be modelled by largely independent event cascades. If we assume a fixed routing graph, then the corresponding routing graph could be statically split into a set of disjoint *partitions*. Within each partition, event cascades with different timestamps can run in parallel without interference. Within a partition, non-simultaneous event cascades have to be serialized in order of timestamps.

[35] i.e., by spawning several instances of Code 6-2 for each event.

It is possible that a user simultaneously views geometry nodes that are in disjoint partitions of the routing graph. Out-of-order processing of partitions could lead to noticeable visual artefacts: a user would see an incorrect event ordering. However, this is unimportant for almost all worlds, unless non-simultaneous event cascades are more than a few milliseconds apart. Causally related behaviour will always be visualised in the correct order as this must be in the same routing graph partition.

Further parallelisation within a routing graph partition is more intricate. There is a risk that the VRML specification is violated when event cascades with different timestamps share a fan-in node. It could happen that an event with later timestamp is received first by the fan-in node. This is in conflict with the requirement that a node processes events in order of their timestamp. However, if it can be established that some routes within the partition do not fire for a given event sequence, the partition can be safely subdivided. Such subdivision relies on a dynamic evaluation of the firing of routes which is likely to be expensive. This feature is not implemented in the current server version.

Further Parallelism

Beyond parallel event cascade evaluation, further opportunities for parallelism exist:

Evaluation of `Sensor` nodes can be done in parallel if their required sensor information is available (e.g. current time, user location, etc.). `Sensor` nodes may then register discrete events with a Scheduler.

Scheduler. The whole of Code 6-1 may be replicated for each partition of the routing graph. Again, synchronised time must be available at each location.

6.2.3 Mapping of the VRML Execution Model onto SODA

We have already seen that the semantics of VRML nodes and SODA active objects are very similar. Both provide some degree of object encapsulation, since communication can only take place through a well-defined event-interface. In both systems, incoming messages trigger the asynchronous execution of some activity.

We applied a mapping between VRML and SODA as follows: VRML nodes are directly represented by active objects. Those nodes may then perform parallel event generation or processing, which is the mainstay for parallel event cascade evaluation. Asynchronous VRML event passing is mapped onto active object method calls. Figure 6-14 shows how all elements of the VRML execution model map onto a valid equivalent in SODA.

SODA Funnels are used to implement subcalls in event cascades. This is useful so that a Sensor can be informed about the termination of an event cascade. For example, the `TimeSensor` implementation might create a new continuous event as soon as processing for the previous one has completed. Code 6-3 shows how the `PositionInterpolator` deals with the `processEvent` method. After receiving a (`set-`)`fraction` event, a new position value is computed and made available on the value field which is then propagated along all attached routes via a call to route. route takes the funnel as an argument and uses this to collect the results of a set of future subcalls (see Code 6-4), one for each outgoing route. The funnel is then activated in Code 6-3.

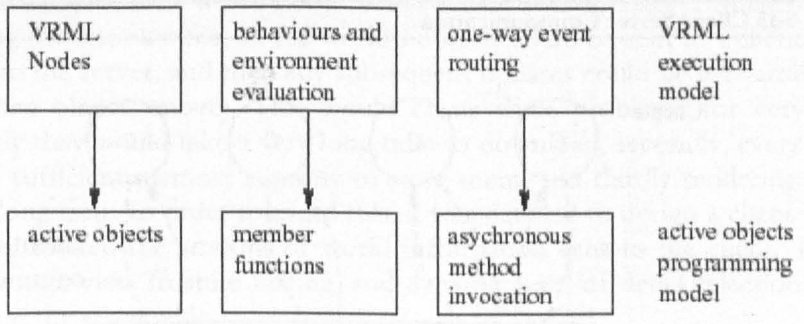
Code 6-3 Excerpt of the PositionInterpolator class

```
active class PositionInterpolator {
  SFVec3f value = new SFVec3f(Field.EVENT_OUT, 0.0f, 0.0f, 0.0f);
  ...
  public Future processEvent(Event e) {

    Future fut = null;

    if (e.getName().equals("fraction")) {
      if (setIndexFract(((ConstSFFloat) e.getValue()).value)) {
        // compute the new position values given the fraction
        int v0Base = iL * 3;
        int v1Base = (iL + 1) * 3;
        v1[0] = keyValue.mvalues[v0Base];
        v1[1] = keyValue.mvalues[v0Base+1];
        v1[2] = keyValue.mvalues[v0Base+2];
        v2[0] = keyValue.mvalues[v1Base];
        v2[1] = keyValue.mvalues[v1Base+1];
        v2[2] = keyValue.mvalues[v1Base+2];
        x = (v1[0]*af) + (v2[0]*f);
        y = (v1[1]*af) + (v2[1]*f);
        z = (v1[2]*af) + (v2[2]*f);

        value.setValue(new ConstSFVec3f(x, y, z), e.getTimeStamp());
        fut = new Future();
        Funnel fun = new Funnel(fut);
        value.route(fun);
        fun.activate();
        return fut;
      }
    } else if (e.getName().equals("keyValue")) {
      ...
    } else if (e.getName().equals("key")) {
      ...
    }
  }
```

Code 6-4 Excerpt of the Field class

```
public abstract class Field {
  ...

  public void route(Funnel fun) {
    // only route eventOuts and exposedFields.
    if ((fieldType & Field.EVENT_OUT) == 0) return;

    // no routes attached to this eventOut?
    if(routes.size() == 0) return;

    // if this eventOut has already been routed with the same
    // timestamp and the VRML97 loop breaking rule shall be applied,
    //then ignore this repeated firing.
    if (APPLY_LOOP_BREAK_RULE) {
      if (lastUpdate <= lastFireTime) return;
    }

    //currently routed value is the latest event on this field now.
    lastFireTime = lastUpdate;

    // send the event to all routes attached to this eventOut.
    Iterator i = routes.iterator();
    while (i.hasNext()) {
      Route connect = (Route) i.next();
      Event ev = new Event(connect.destEvent, lastUpdate,
                           (ConstField) this.toConst());
      Future f = connect.destNode.processEvent(ev);
      f.setFunnel(fun);
    }
  }
}
```

Figure 6-14 Mapping of the VRML execution model onto Active Objects


VRML Nodes
behaviours and environment evaluation
one-way event routing
VRML execution model
active objects
member functions
asychronous method invocation
active objects programming model

The scheduler described by Algorithm 1 could be implemented as an additional active object. It could then be used to register continuous and discrete events and fire the corresponding event cascades. This would allow the spawning of event cascades in timestamp order with parallel execution of simultaneous event cascades. However, one problem of such a scheduler object is that it represents a central bottleneck and impacts on scalability.

For this reason, our implementation does not use a central scheduler. Instead, server events are immediately evaluated as they are triggered. This is an optimistic implementation that renders correct results in respect to the VRML execution model under two preconditions:

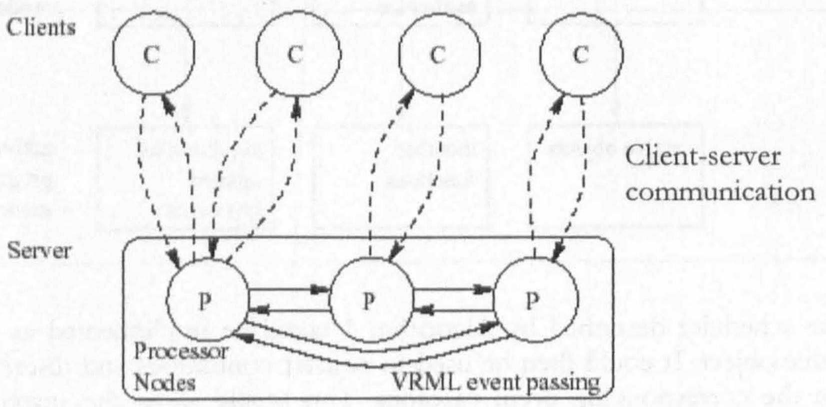
- A world's routing graph must be split into a set of partitions so that fan-in cannot occur for events with non-identical timestamps. This is the responsibility of the world designer. The world designer must ensure that fan-in routings do either not occur at all or only occur for simultaneous events. However, this requirement can be given up if it is not important that VRML semantics are exactly followed.
- The system clocks of individual hosts in the cluster must be synchronised. This is important, because TimeSensor events are based on the local system time.

Note that the first requirement is only necessary if the VRML specification is to be strictly observed. This is not always necessary. Our implementation can therefore handle a more relaxed interpretation by ignoring out-of-order events at any node. e.g., events that do not follow a monotonously increasing order of timestamp at their arrival are not processed.

6.3 Client-Server Architecture

So far we have described how the server implements a powerful, scalable execution engine, which maintains evolving world state and evaluates world dynamics on a distributed memory architecture. This removes from the clients the need to compute world changes, so reducing their required processing power. We will now focus on how this server can support multiple, heterogeneous clients. In particular, the server exploits information filtering techniques, reducing a client's bandwidth, storage and processing requirements. Clients with limited resources can therefore participate in highly complex, virtual worlds.

Figure 6-15 Client-Server Communication



6.3.1 Requirements and Approach

The following is a list of minimum requirements which we demand of the client-server communication scheme:

Near Real-time Requirement. A state change to the server-side scene graph should be replicated as fast as possible to the AOI of all affected clients. This means that clients with overlapping AOI perceive near state equality. It is also impossible to provide real-time interaction, exact equality of dynamic shared data and scalability, simultaneously. The handshaking required to insure exact equality between more than a few shared copies adds latency that makes real-time interaction impossible. To achieve fast interaction, one must allow temporary disagreements between clients about the state of the shared data. Fortunately, if such disagreements are below the frame refresh interval, they will be unnoticeable for a user.

Scalability Requirement. Hundreds of simultaneous users and hundreds of thousands of objects should be supported. The main technique to fulfil this requirement is information filtering by view frustum culling on the server side, which is performed in parallel.

Low Bandwidth Requirement. Operation over low-bandwidth links (e.g., mobile devices on wireless networks) is desirable. This can be achieved through dynamic level of detail switching and adaptation of the frequency in which AOI replication occurs.

Information must be sent from the server to the client so that a user can view the world. For example, the server could render each client's view of the world and so only send clients a stream of frames, represented as bitmaps. This limits the work of the client to reading frames from the network and displaying them. However, it would place a large load on the server and the network. Further, many clients have specialized graphics hardware and can render 3-D scenes efficiently. It was therefore decided that each client should render its own view of the world.

This approach requires the world's objects to be sent from the server to each client for rendering. At one extreme, all the world's objects could be sent to a client when it connects to the server, and then any subsequent updates could be forwarded (for example when objects move). This would create three problems for very large worlds: firstly they would take a very long time to download; secondly, every client would need sufficient memory capacity to store them, and thirdly rendering could take a very long time. In order to avoid this, it was decided to design a client server interface that limited the amount of world information sent to the client. This is achieved through view frustum culling, and dynamic level of detail selection, both performed on the server side.

6.3.2 Server-side culling

When a client initially connects to the server, one host is chosen to act as a proxy for it, handling all subsequent client-server communications. A load-balancing scheme can be used to spread the set of client proxies evenly across the parallel server's processors, to promote scalability.[36] Once connected, the client negotiates its desired LOD quality level as well as its maximum viewing distance. Further on, the client reports to the proxy any changes to its position and orientation. Based on this information, the server updates the position and orientation of the avatar that represents the user, so that other online users can see those movements. The client proxy also computes the four planes describing the client's view frustum. This is then used to determine which objects in the world are visible to the user, as only these objects are sent to the client.

Two options were considered for selecting objects. In the first, the world is represented as an unstructured set of objects. Therefore, every time a client connects, or moves in the world, all the worlds' objects must be compared with the client's view frustum to determine which need to be sent to the client for rendering. While this can be done in parallel, with each machine comparing the frustum with the objects held in its own memory, for a large world, this requires a very large amount of processing, and this would limit the scalability of the system. Therefore, an alternative was designed.

In this scheme, the VRML world is structured as a set of complex objects. It makes use of VRML's "PROTO" statement that enables the encapsulation of a partial scene graph, with a well-defined interface of fields and events [35][36]. In addition, this allows a world programmer to move responsibility for repetitive and low-granularity behaviours to the client side.

6.3.3 PROTO encapsulation

The client-server interface revolves around PROTOs as atomic shared units. In our system, PROTOs contain a set of mandatory fields that are accessed both by the client and server (see Code 6-5 for an example PROTO node definition). Whereas the client deals with the contents of the PROTO, the server sees it as a "black box". The mandatory fields are: the node's position, orientation and scale; the size and position of its bounding box; and, references (in the form of URLs) to files that contain the encapsulated scene graphs at different Levels of Detail (LOD). As is described in detail below, the server determines the appropriate level of detail for every PROTO and only the corresponding LOD file is downloaded to the client.

It is important that a client's copy of a PROTO is synchronized with the server's version of the same PROTO. Therefore, if the server-side state of the PROTO is updated, the client is notified of the change by the server. Vice versa, a notification message is sent to the server if the user interacts with the PROTO, leading to an `eventOut` being generated on one of the exposed fields. All behaviour encapsulated in the PROTO is evaluated on the client side, which is more efficient in terms of network utilization for e.g., simple animations. Complex and non-repetitive behaviour however should be defined outside of PROTOs, so that evaluation takes place on the server. This gives a world designer control over the location of behaviour evaluation.

[36] Load-balancing is not provided in the current implementation. It is the clients' responsibility to balance their connection requests.

Code 6-5 An example PROTO node definition

```
PROTO House002 [

   #mandatory fields for shared PROTOS
   field          SFVec3f     bboxCenter   2.5 2.5 2.5
   field          SFVec3f     bboxSize     5   5   5
   field          MFFloat     levels       [100,200,300]
   field          MFString    urls [
     "http:////www.cs//vrml//House002_LOD0.wrl",
     "http:////www.cs//vrml//House002_LOD1.wrl",
     "http:////www.cs//vrml//House002_LOD2.wrl",
   ]

   exposedField SFVec3f    position     0 0 0
   exposedField SFRotation orientation  0 1 0 .5
   exposedField SFVec3f    scale        0 0 0

   #optional shared eventIn/eventOut fields, specific
   #to the described object
   exposedField SFBool     doorOpen     FALSE
] {...}
```

Structuring the world in terms of complex nodes defined by PROTO descriptions is the key to an efficient client-server interface. Performing culling is reduced to checking if the bounding box of each PROTO is visible to the client. For a typical world, this reduces the number of comparisons by a factor of 10 to 1000 when compared to culling every individual node that makes up a scene.

When a client connects to the server, the set of PROTOs that are visible to a client are determined, and the server sends the contents of all mandatory and optional fields. The server knows the distance of each object from the client's viewpoint, and so can determine the appropriate LOD level for a PROTO, which it communicates to the client. LOD levels for a PROTO can change dynamically as a result of the client navigating in the world. On the basis of URLs the client can perform efficient and easy to implement caching of LOD files, as is now explained.

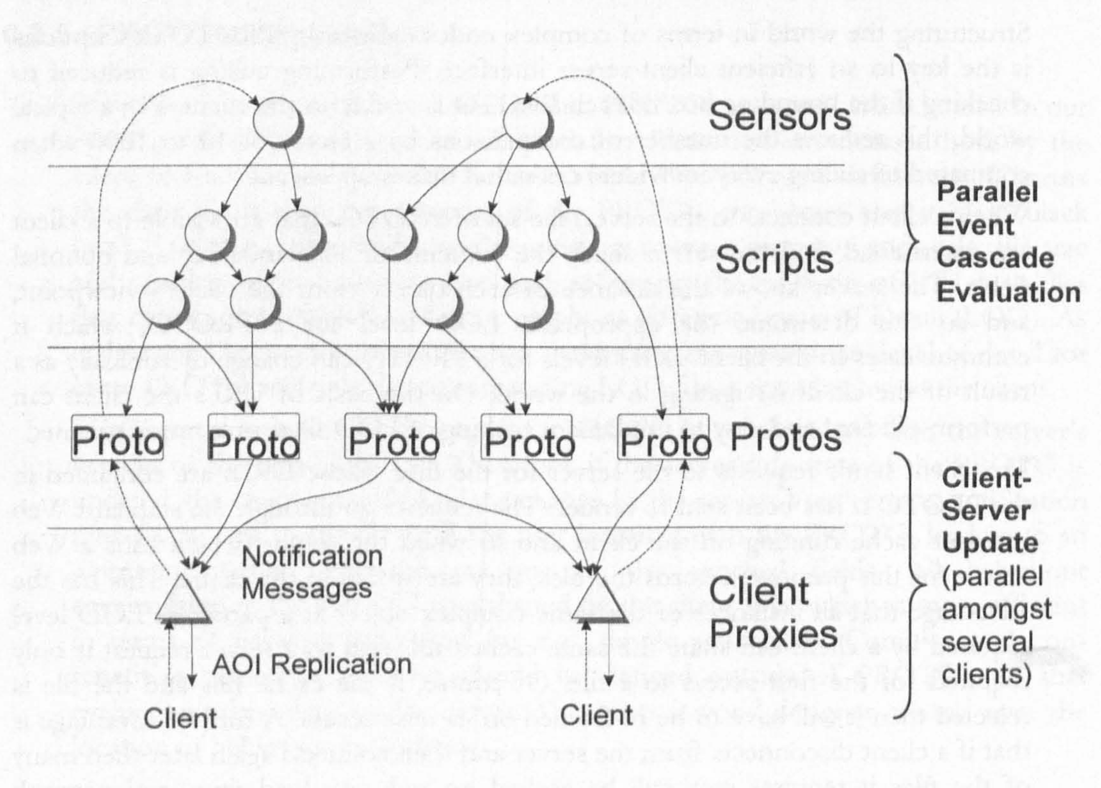
The client sends requests to the server for the files whose URLs are contained in the PROTO it has been sent to render. The requests go through the standard Web browser cache running on the client and so when the server (which runs a Web server for this purpose) returns the files, they are stored in the cache. This has the advantage that all instances of the same complex object at a particular LOD level required by a client can share the same cached file, and so a server request is only required for the first access to a file. Of course, if the cache fills and the file is rejected then it will have to be re-fetched on its next access. A further advantage is that if a client disconnects from the server and then connects again later then many of the files it requires may still be cached, so reducing load time, and network bandwidth. By using the client's Web cache to store the LOD files, the benefits of cache management were obtained without any extra development being required. It is important to note that the client handles all PROTO downloads and additions to the client scene asynchronously and without blocking the user's navigation. PROTOs are streamed on demand into the client scene.

The client proxy on the server keeps a record of all the PROTOs that it has sent to the client. If the state of the complex object is changed on the server, as the result of an event, then the changes are propagated to the client. Similarly, if a user interacts with an object on the client, then all resulting events are routed to the server so that they can cause the necessary change in the world. Consequently, all other users will observe a change to the world made by one user.

After the initial loading of objects into the client, further objects are transmitted if the client moves, so that new objects are visible, or if moving objects come into the client's field of view. Similarly, if objects move out of visibility then the server informs the client so that the client can remove PROTOs from the local scene graph.

One disadvantage with this approach is that the world has to be represented in a particular, structured format in the VRML file. This means that existing large-scale VRML scenes need to be manually reworked to represent the scene as a collection of PROTO nodes at the server. For each of these components, bounding boxes must be created and, for optimum performance, a set of LODs should be provided. Tools exist to automatically create LODs from a high-polygon-count VRML model, e.g., LODESTAR [151].

Figure 6-16 Server Architecture



6.3.4 Update Accumulation Algorithm

Our protocol achieves low bandwidth operation by using reliable communication of object update messages. This allows the use of compact differential messages and eliminates the need for keep-alive messages, as e.g., in DIS. For each client, the server maintains a separate FIFO queue of update messages that the client can retrieve at its own pace. The consistency requirement for every client is relaxed somehow: it is sufficient to have "eventual" consistency, i.e., once all pending messages have been received by all clients. We do not currently provide a priority scheme for different update messages. However, this would be feasible, e.g., based on an object's distance from the viewer or a client's particular interest in a world.

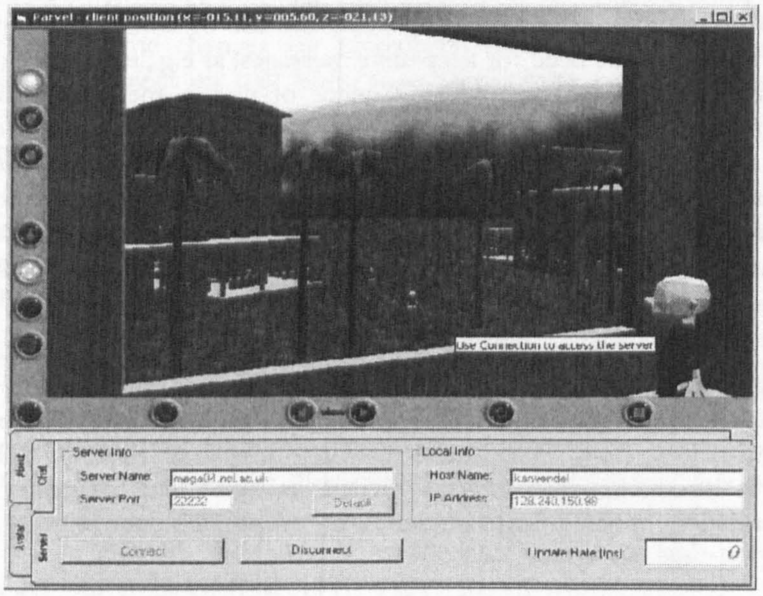
The client proxy is responsible for generating update requests to all PROTOs in a scene. These requests contain information about the client's AOI. Every PROTO compares whether its bounding box intersects with the new AOI and if so, returns recent field updates. To optimise this algorithm, every PROTO field on the server has a timestamp that indicates when its value last changed. Based on this information, updated field values are only propagated once to every client.

All updates are queued at the client proxy for batched delivery to client (flow-control). While field updates are pending delivery to the client they might still be overwritten by newer values as they become available. As mentioned above, instead of having a FIFO queuing scheme, it would be more appropriate to support different delivery priority. These could be negotiated based on what aspect of the world the client is interested in (e.g., a given client might be interested in air traffic, but not in vehicle traffic, etc.).

6.4 Client-side implementation

The client-side implementation of this system is based around a VRML97 browser component controlled and monitored by a lightweight application layer. Three client versions exist: a Java applet, using the VRML browser as plug-in on a web page, a Java stand-alone application and a Visual Basic implementation. The Visual Basic client uses the ParallelGraphics VRML browser that acts as a Windows COM component. Figure 6-17 shows a screenshot of the browser interface.

Figure 6-17 Screenshot of the browser interface
Other online users are represented through their Avatars.



Farvel - client position (x=-015.11, y=005.60, z=-021.13)

Server Info
Server Name: mega04.nel.ac.uk
Server Port: 22232

Local Info
Host Name: karwendel
IP Address: 128.240.150.98

Update Rate (ips): 0

6.4.1 EAI

The External Authoring Interface (EAI) was developed as an extension to VRML [111]. It provides an interface for programmatic manipulation of a VRML world to an external application. For example, it is possible to modify the values of `eventOut` or `eventIn` fields or to set up event listeners that receive callback notification when a field value changes. A Java binding of EAI exists in the `vrml.external` package, which comes with many VRML browsers.

The capabilities of the EAI are useful for the implementation of our VRML client. Since EAI allows manipulation of the scene graph, it is possible to dynamically add and remove VRML PROTOs to the client's AOI. In addition, event listeners are used to receive callback notification for user interaction with sensor nodes in the AOI. This mechanism is used to inform the server about client interaction.

6.5 Performance Results

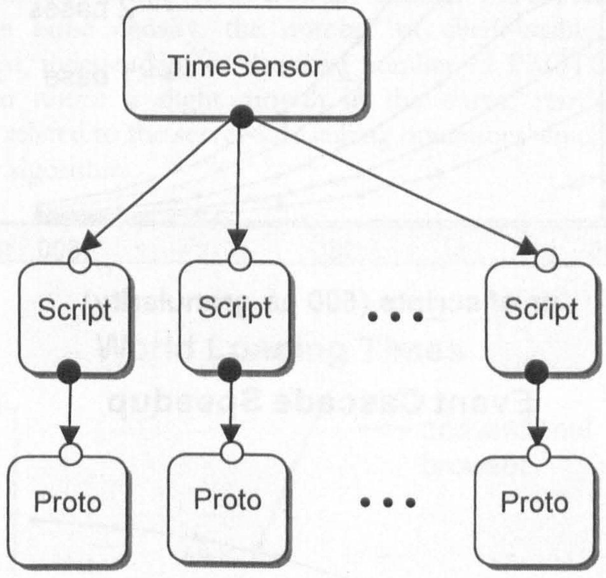
The performance of our implementation was measured separately for parallel event cascade evaluation and client-server communication (see Figure 6-16).

6.5.1 Parallel Event Cascades

To measure the performance of the server-side event cascade evaluation mechanism, we use a test world with a simple routing graph that is automatically generated by a script. This world consists of a variable number of `Script` – PROTO node pairs. The `Script` nodes each have a granularity of 500μs (on the

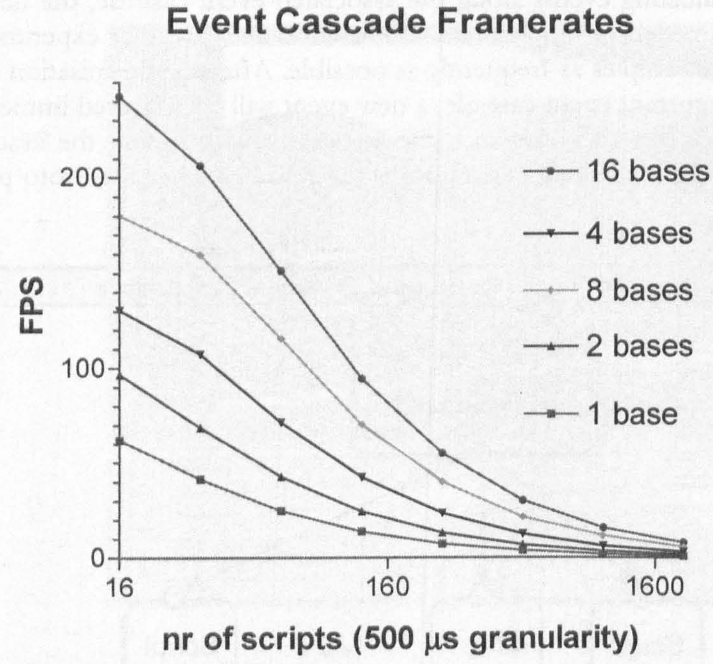
Mega platform). They are all connected to the `fraction_changed` eventOut of a single `TimeSensor` as shown in Figure 6-18. To measure the maximum attainable update rate for evaluating events along the associated event cascade, the detached method of the TimeSensor implementation is modified for this experiment to produce continuous samples as frequently as possible. After synchronisation on the termination of the current event cascade, a new event will be triggered immediately on the TimeSensor. We then measure the frequency with which the associated event cascades can be evaluated in relation to the number of Script-Proto pairs in the routing graph.

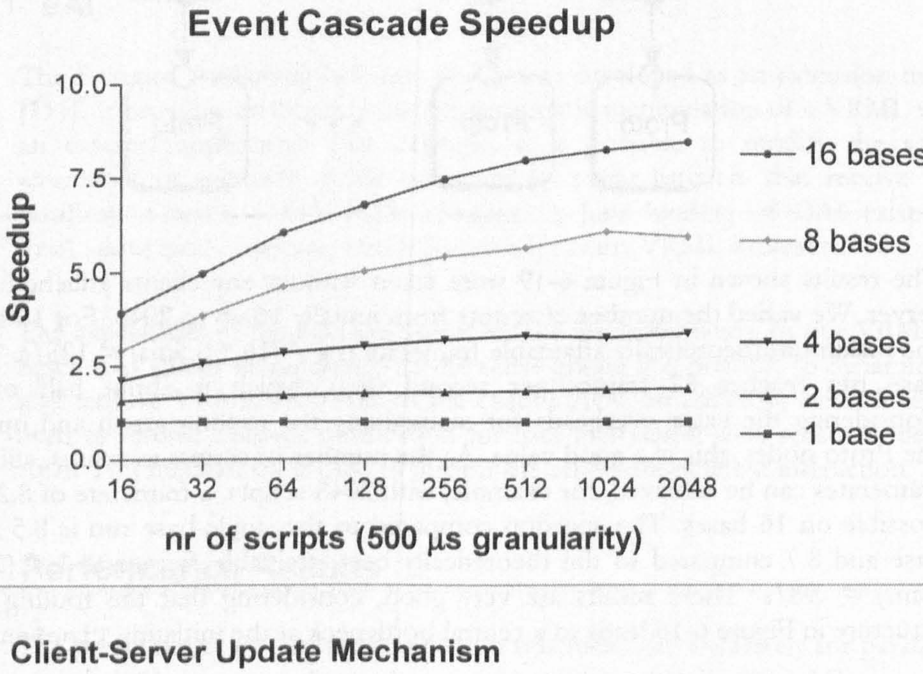
Figure 6-18 Routing Graph for Testing the Event Cascade Evaluation Performance



The results shown in Figure 6-19 were taken without any clients attached to the server. We varied the number of scripts from initially 16 up to 2048. For 16 scripts, the maximum theoretically attainable framerate is 1 / (16 * 0.5ms) = 125/s. The 1-base run reaches 62 frames per second (fps), which is about half of this. Considering the extra overheads for maintaining the routing graph and updating the Proto nodes, this is a good value. As the number of scripts increases, still good framerates can be achieved; for example, with 2048 scripts, a framerate of 8.2 fps is possible on 16 bases. The speedup compared to the single-base run is 8.5 in this case and 8.7 compared to the theoretically best attainable framerate 1 / (2048 * .5ms) = .98/s. These results are very good, considering that the routing graph structure in Figure 6-18 leads to a central bottleneck at the initiating `TimeSensor`.

Figure 6-19 Parallel Event Cascades





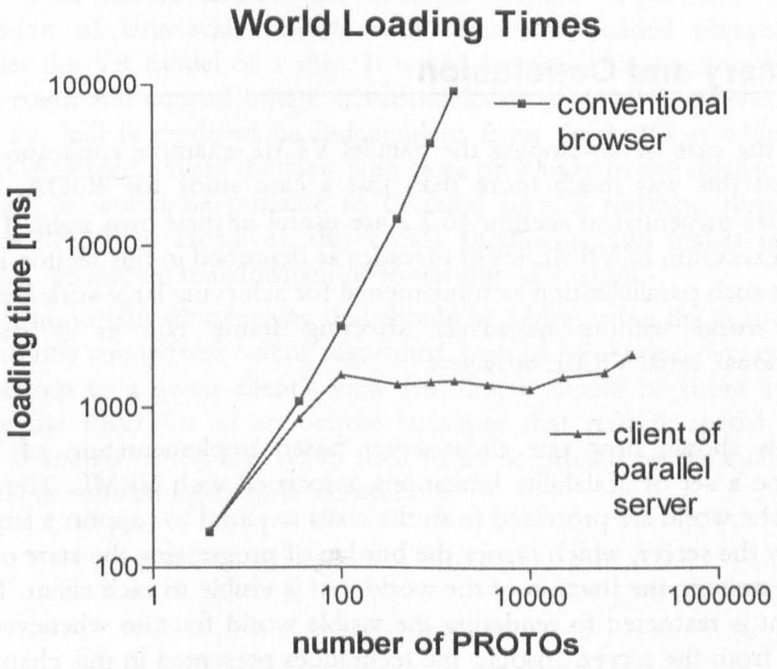
6.5.2 Client-Server Update Mechanism

A second experiment is concerned with the performance of the server-client update mechanism. As client-machine, a dual-processor PIII-500Mhz with hardware-accelerated graphics card was used. Again, a script-generated world was used, this time with a parameterised number of PROTO nodes. Each PROTO embedded a scene graph description of 23kByte (uncompressed, without textures). With a growing number of PROTO nodes, also the total world size was increased to distribute them at the same density. No routing graph was defined in order to only measure the performance of the client-server interface.

Figure 6-20 shows the loading times in relation to world size, as a comparison between a conventional VRML browser and a client on the parallel server. As the world grows, these increase dramatically for a conventional browser. For more than 1000 PROTOS, the total scene size reaches more than 20 MByte. In this situation the conventional browser crashes after a delay of several minutes.

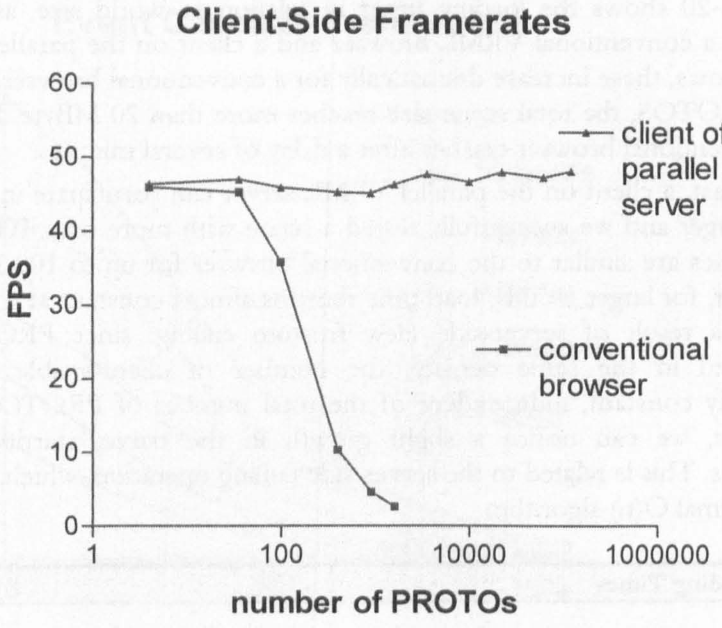
In contrast, a client on the parallel VRML server can participate in scenes that are much larger and we successfully tested a scene with more than 100000 PROTOs. Load times are similar to the conventional browser for up to 100 PROTO nodes. However, for larger worlds, load time remains almost constant at roughly 1400 ms. This is a result of server-side view frustum culling: since PROTO nodes are positioned in the same density, the number of client-visible nodes remains essentially constant, independent of the total number of PROTOs in the world. However, we can notice a slight growth in the curve, starting from 10.000 PROTOs. This is related to the server-side culling operation, which currently uses a non-optimal O(n) algorithm.

Figure 6-20 Loading Times



View frustum culling is also beneficial, once the client starts to navigate through a world. Since much less geometry is loaded into the navigation is much smoother and can deliver high framerates of 45 fps. Figure 6-21 shows the comparison with a conventional browser presenting the same VRML world; for worlds larger than 100 PROTOs, these become non-interactive in the case of the conventional browser.

Figure 6-21 Client Frame Rate

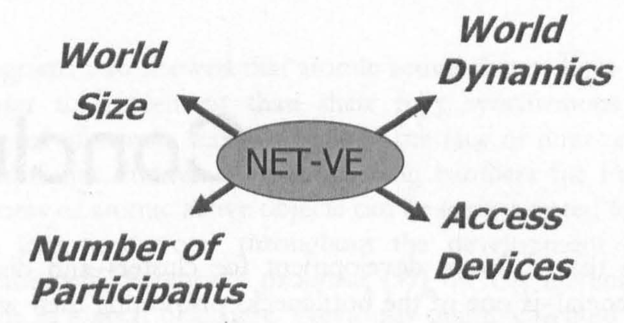


6.6 Summary and Conclusion

During the case of developing the parallel VRML example application it became clear that this was much more than just a case study for SODA. In fact, the techniques presented in section §6.2.2 are useful in their own right. The scalable, parallel execution of VRML event cascades as described in this section is novel. We consider such parallelisation as fundamental for achieving large-scale behaviour in a VRML world without negatively affecting frame rate as is the case for conventional, serial VRML browsers.

We have shown how our client-server based implementation of VRML can overcome a set of scalability limitations associated with VRML. The clients that browse the world are protected from the costs required to support a large, complex world by the server, which carries the burden of progressing the state of the world, and determining the fraction of the world that is visible to each client. The work of the client is restricted to rendering the visible world fraction whenever it receives updates from the server. Insofar, the techniques presented in this chapter can be a stepping stone towards future, standards-based networked virtual environments (NET-VE) that are scalable along four different axes: the world size, the number of simultaneous users, the complexity of world behaviour/simulation, and the capability of access devices.

Figure 6-22 Dimensions of Scalability



Experience in building the VRML server has shown the power of the SODA active object model for parallel, object-based software design. Extraction of parallelism was facilitated at the level of the event execution model, in the information filtering and traffic shaping of client proxies and in the evaluation of sensor nodes. The results demonstrate that real performance gains can be achieved.

Future work should address the issue of "nested" PROTOS. Currently, the description of large-scale objects with smaller embedded objects is difficult. Consider the VR model of a ship. It would be useful to describe the ship's hull, engine room and control bridge at various levels of detail. However, the LOD at which the hull is rendered be independent from the LOD at which the engine room is displayed, unless the view is in close proximity to the smaller components. Currently, it would be possible to describe such a situation through a set of separate PROTOs. However, this breaks modularity and makes it awkward to apply e.g., geometric transformations to the ship as a whole.

Another important shortcoming that should be addressed in the near-term future is the currently suboptimal culling algorithm. Instead of checking every PROTO for membership to a given client's view frustum, it would be more appropriate to arrange the PROTOs in an octtree structure that reflects spatial arrangement. Whole branches of the tree could then easily be pruned by the culling algorithm, resulting in reduced algorithmic complexity.

Chapter 7

Conclusion

The suspicion that software development for clusters–and distributed memory machines in general–is one of the bottlenecks preventing their widespread use has been confirmed in the literature survey. The current level of software support was found to make it difficult to harness the power of a cluster for novice users or large-scale, real-world applications. In most systems, the burden of managing details of the parallel execution, such as allocation of data and activities, synchronisation and inter-machine communication rests with the developer. With growing program complexity, such low-level issues become increasingly difficult to manage and can easily become a main task in the design and implementation of a parallel program. As a result, the potential of clusters is not yet fully realised.

The aim of this thesis was to examine the viability of combining COOP and implicit parallelism address this problem. Especially the integration of both paradigms into the active object concept appeared very promising. We experimented with many existing active object systems, focussing on their usability in the context of large-scale real world applications. Since they were all inadequate at some point, we developed the SODA programming environment as a basis for experimentation and proof of concept. The SODA programming model implicitly hides details of decomposition, allocation, synchronisation and communication behind the object façade, while active object instances make parallelism explicit. A SODA program can expose large amounts of parallelism; the SODA runtime system is then responsible for limiting this parallelism and adapting it to platform characteristics based on thread-multiplexing and aggressive inlining of most active object method invocations. This *expose-and-then-reduce* approach was found well suited for irregular, dynamic programs where static compile-time analysis is of limited value.

The development of the SODA programming model went hand-in-hand with the design and implementation of test programs. Especially useful in this aspect was the case study in Chapter 7 capturing the requirements of a real-world large scale server application. The experience gained was invaluable for making design decisions in SODA. Many features not originally intended or though of were added as a result of this early testing and exposure to real-world requirements. For example, the need for detached methods became obvious when implementing Timer objects and TimeSensors in the VRML execution engine. The example programs were also an important test-bed for measuring the trade-offs between ease-of-use and runtime efficiency. Experience also showed that makes it easy to expose concurrency, but more difficult to programmatically restrict it (for example, see §3.8.2, Dining Philosophers). We see this as a desirable property for a parallel programming model.

Our test programs also showed that atomic active objects have usability advantages and are easier to implement than their fully synchronous counterparts. The deadlock hazard of atomic active objects in the face of direct or indirect recursion is removed through Funnels as non-blocking handlers for Future sub-calls. The reduced liveness of atomic active objects can be compensated for through detached methods. A long-time worry throughout the development of SODA was the inheritance anomaly. Ferenczi's proposal [59] on circumventing the inheritance anomaly came as a great help here. Previously unimplemented it showed the value of monitors in the context of atomic active objects and when interpreted as conditional critical regions.

We explored the associated trade-offs between ease-of-use and runtime efficiency. The performance results show SODA's practical value and the absence of fundamental performance limitations in its programming model. We also showed that SODA programs can execute efficiently across different distributed-memory architectures and achieve significant speedups. In the future, we wish to experiment with larger platforms, since performance results on up to 16 nodes have been promising.

Most future work should address current limitations of the runtime system. One main shortcoming is the lack of support for dynamic migration of active object. This restricts the currently supported application domain. Objects with highly varying processing and communication requirements would lead to load imbalance in the system. Work in this direction must also provide a mechanism to detect load imbalance in a decentralised and scalable manner and initiate compensating object migrations. This mechanism must also take into account the communication bandwidth between active objects in order to improve locality. A huge body of research work exists in the area of dynamic load balancing and it would be interesting to experiment with various proposed techniques in the context of SODA. A related issue is the inappropriate support for multiprocessor nodes. Currently, aggressive inlining does only exploit single processor nodes and should be assisted by heuristics to expose further machine-local parallelism.

Another issue the implementation should address is the instability of the underlying hardware platform. Currently, a SODA program aborts if any of the cluster nodes it is running on fails. With increasing size of the cluster the likelihood of such an incident increases rapidly. If a single node fails with probability p, then a n-node cluster fails with probability $1-(1-p)^n$. Possible approaches include the use of transactional active objects or replicated active objects as mentioned in §3.3.1.

In summary, our work has shown that the SODA programming model can combine ease-of-use with efficient execution on distributed memory machines. This confirmed our original thesis that the integration of COOP and implicit parallelism is viable and worth pursuing. With its novel features, SODA can bridge the gap between previous active object systems and all-implicit programming approaches. It is the author's hope that SODA will be used as the basis of further research and experimentation.

Appendix A

NET-VE Related Work

The new usage model for the presentation of VRML (see §6.1.2), draws heavily on the topic of networked virtual environments (NET-VEs), so we will discuss the relevant background literature in this area. A NET-VE can be defined as a distributed computing system, which allows several participants to interact and navigate in a virtual, simulated, three-dimensional space, or *world* [156]. Typically every participant is on a separate host or *client*, which is responsible for rendering the participant's view on the shared world. The overall aim of NET-VEs is to "*transform today's computer networks into navigable and populated 3D spaces that support collaborative work and social play*" [21]. The metaphor of spatial interaction is readily adoptable for humans and many new and exciting applications could be based on NET-VE systems, going beyond what is currently possible in the field of computer-supported cooperative work (CSCW). Example applications include education, large-scale visual simulation, entertainment, collaborative design [114], analysis and decision support, and human-computer interfaces.

At the low end of the price range, standard desktop workstations with 2D mice can be used for the graphical presentation and navigation. Advanced systems will provide specialised hardware devices to increase a participant's perceived realism. Examples are head-mounted displays (HMD), that offer stereoscopic viewing or Cave Automatic Virtual Environment (CAVE) hemicubes, which provide total immersion and possibly tracking of a participant's head or limb movements. With forthcoming developments in the area of wireless networking, even low-powered PDAs could be used to increase the reach of NET-VEs into the area of mobile, potentially location-based applications.[37]

Aural presentation can be just as important to the illusion of reality as is the visual presentation. As you approach sound sources, they get louder. However, in this work, we will not consider other media types than visual content.

A.1 Scalability

Improvement of scalability is a major thrust in the area of virtual reality research. We distinguish four dimensions of scalability:

[37] There are many potential applications for wirelessly connected PDAs, possibly fitted with GPS positioning systems. Examples include augmented reality (e.g., visualisation of cabling or piping behind a wall), tourist guides pointing out nearby restaurants or attractions, military applications and "dating" services.

World size. There is a natural desire to make NET-VEs large in spatial extent, in the number of entities populating this space and in the model quality of these entities. Ideally one would want to model every entity with high geometrical precision, for example using a large number of polygons and detailed textures, to improve the degree of realism.

Behaviour. For some applications it is sufficient to have purely static worlds. Examples include virtual architectural walkthroughs [67] and cityscape visualisation. However, we are interested in worlds which expose a certain degree of dynamicity and interactivity, e.g., the entities populating the world exhibit some kind of *behaviour*. To make such worlds realistic, we require a world update logic, which describes the world's behaviour over time. For example, some artificial intelligence algorithms or physics-based simulation might be used to drive entities populating a world (see §A.2). An example is a traffic light button which can be pressed by a participant and which in turn influences the traffic over a crossroad section. A large, realistic world will have lots of interaction and therefore, the world update logic may be a delimiting factor. However, it is important to note that behaviour will often be locally confined, e.g., a discussion between some pedestrians in one corner of the world, a traffic light somewhere else. This means that there is some form of *behavioural locality* common to many worlds.

Numbers of simultaneous participants. The number of simultaneously connected participants is another factor of interest. If this number is growing, latencies and inconsistencies are becoming more and more important.

Device heterogeneity. In a general-purpose system there might be a range of heterogeneous access devices with which users would want to access the system. These devices will have different capabilities in terms of processing powers and network bandwidth. A participant with a low-powered device will want to trade graphical complexity against increased frame rate and update frequency.

The scalability issue is further exacerbated through the real-time requirements of NET-VEs. The reduction of lag is important to increase a participant's comfort and perceived realism. e.g., changes to the shared world brought about by one participant should be visible to another participant with minimum delay. One key factor to guarantee scalability, implemented in virtually all NET-VEs to date is based on viewer locality: in a sufficiently large virtual world, a participant's perceptible space is limited due to occlusion and distance. Information about the world outside the perceptible volume is not of interest to any participant. On the basis of locality powerful information filtering schemes can be adopted (see §A.3 - Area of Interest).

A.2 Entities

A world is populated by *entities*, which are graphical representations of real-world objects. An entity is described in terms of its geometry and other visual characteristics, such as colours, textures, etc. The granularity of entities is usually established by the NET-VE designer. The state of an entity is described by attributes, such as its position and orientation, its colour, shape and size, etc. While most entities will be static, a subset of entities may exhibit a temporal behaviour. For example, a "car" entity might change its position and orientation, the colour of its indicators, the rotational degree of its tyres, the "open" state of a door, and so on. Entities might be atomic or hierarchically composed of sub-entities.

Avatars are a special type of dynamic entities. They provide participants with a graphical embodiment that conveys their identity, presence, location and activities to other participants. Any navigational movements of a participant are directly reflected by the associated avatar. Avatars may also convey further state information about a user, for example body movements or facial gestures, such as smiling or frowning.[38]

An entity's state will be maintained at some master location in the network. To reflect changes to a remotely kept entity, interested participants will typically create a local replica. The state of such replicas must reflect updates to the original entity with minimal lag. This is critical to provide a near real-time view to all participants and to keep information consistent across participants. One approach, for example, is to broadcast every entity's update information to every participant [33;118].

In addition to update messages, interested clients must also know about an entity's fixed state (e.g., geometrical description, size, shape, etc.) before they can render it. Since such information tends to be very complex, it should be transmitted infrequently. To avoid this problem altogether, some systems assume the fixed state of all entities to be available locally *a priori* to starting the client. However, this makes systems rather inflexible and makes it difficult to deploy extensions to a world (compare, for example, the universal media description). Another solution is to use area of interest schemes to only download partial views of the world to a given participant (see §A.3).

A.3 Area of Interest (AOI)

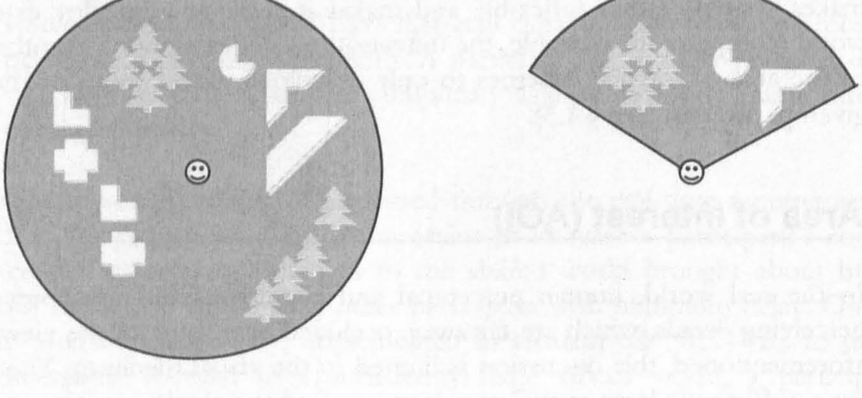
In the real world, human perceptual and cognitive limitations prevent us from perceiving details, which are far away, occluded or outside of the viewing angle (as aforementioned, this discussion is limited to the visual medium). The same is true for a sufficiently large virtual world; most of what a single participant can observer at any time is local in nature to the participant's location. This *locality of perception* is exploited in most NET-VE systems to filter data that is of no interest to a given participant. Every participant is ascribed an area of interest (AOI). This might be a sphere centred on the participant's viewpoint or a *view frustum*, a pyramid with apex at the viewpoint and the four side planes determined by the display edges. Entities

[38] Of course, the difficulty lies in how a participant can control such extended state of their avatar. It is common to see interfaces with buttons that allow the participant to select various predefined behaviours, e.g. "*walk*", "*jump*" or "*wave*".

and their updates are irrelevant to a client if they are outside of the area of interest. They can therefore be filtered out from the participant's view on the world. This exploitation of locality can be a powerful basis for schemes to reduce network traffic and participant-local graphical complexity and therefore allow scalability. Of course, the AOI must be adapted dynamically as a participant navigates the world.

As an example, consider a virtual model of an entire city, where hundreds of people are interacting and many simulations are active to compute dynamic world behaviour in response to user interaction and passage of time. In this situation, the world model contains a huge number of objects and because many activities are happening, there are lots of objects that are moving or otherwise changing their properties. If an individual user attempts to maintain a complete picture of the dynamically evolving world, he will have to receive and process a torrent of information about changes happening in the world. The client requirements would therefore grow linearly with the world size/number of connected participants. In the absence of interest management, such a system prohibits scalability, since demands would eventually overwhelm the resources available to any one participant. The main difficulty for AOI mechanism stems from the heterogeneity and the dynamics of the clients, not only in terms of bandwidth and processing power, but also in terms of data interest and virtual and physical locations. AOI management and information filtering is useful beyond the reduction of message bandwidth between participants. In particular, it can dramatically reduce the number of entities kept in participant local memory.

Figure 7-1 Distance based AOI vs. View Frustum AOI



A.4 Levels of Detail (LOD)

One could imagine a large cityscape being studied by a participant flying high above the terrain and therefore seeing the world from a bird's eye view. It is then possible that all entities in the world are within the AOI; however, such entities can only be perceived at a very low level of fidelity, due to the distance from the ground. In a NET-VE system simulating such a situation, it is therefore sufficient to send low-fidelity update information for such entities (Of course, a user could use binoculars to get a better view, but this would restrict the necessary update information to the much smaller viewing angle).

Entities that are far from the observer and cover a small area on the screen can be drawn with less detail without compromising significantly the appearance of the model (see Figure 7-2). Applying this technique to all objects in the world can dramatically reduce the rendering time of complex models and allows the creation of virtual worlds with a lot of detail while limiting the rendering costs to those details that are visible. Five criteria have been proposed to modulate an entity's level of detail (see Table 7-1) [145]. It seems likely that a combination of two or more of these is the best approach for the requirements of a general-purpose NET-VE system. One major issue of research is the automatic creation of various LOD levels for a given entity. Possible methods include geometry removal (of vertices and polygons), sampling (determine a simplified model, that fits a sample of the original) and adaptive subdivision (refine a crude model through subdivision where it varies from the original).

Figure 7-2 Three different levels of detail for a car object

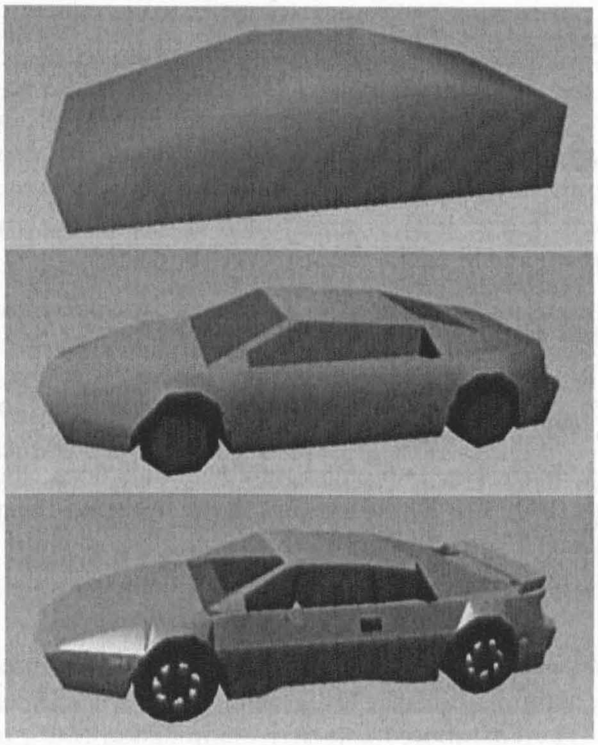


Table 7-1 Criteria for Dynamic LOD Selection [145]

Criterion	Description	Assumption
Distance	according to the distance from the viewer	All users have the same quality requirements. However, this might not be true for different display resolutions (e.g., PDA vs. graphics workstation).
Size	according to the pixel size on the display device	To compensate for the problems of the distance-based technique, chose the LOD according to the pixel size on the screen.
Eccentricity	according to the degree to which the entity exists in the periphery of the display	A user will focus his interest in the centre of the screen. Off-centre imprecisions can therefore be traded off against better quality in the centre of the view.
Velocity	differential velocity between viewer and entity	The detailed perception of quickly moving entities is limited (e.g., in fast flying sequences). They can therefore be rendered with less detail.
Fixed Frame Rate	Maintain a constant frame rate for the viewer	Frame rate is more important than accurate geometrical modelling.

A.5 Communication Structure

According to their communication structure, Net-VEs can be broadly classified into two categories: *client-server* systems vs. *peer-to-peer* systems. In peer-to-peer systems, communication occurs directly between participant's machines. In client-server systems, entity state is maintained at the server and the communication topology is restricted to single client-server connections.

Peer-to-Peer Broadcast

Peer-to-peer systems divide the world state and its update logic over all participant's hosts in order to achieve scalability. Successful examples in this category include military battlefield training simulations, such as SimNet [33;118] and NPSNet [109;192]. These systems were designed to support many dynamic entities, simulated or controlled by real users. Every entity is hosted on a single machine and represented by a single task processes.

Every machine participating in a SimNet simulation uses a *best-effort broadcast* protocol to communicate the state of locally kept entities to all other remote participants. The Distributed Interactive Simulation (DIS) protocol [33] defines a simple packet format for describing updates to an entity's state. The set of entities and possible states is fixed and known a-priori to all participants, which is suitable for the application domain. To minimise bandwidth requirements and the frequency of update messages, remote entities are simulated locally using *dead reckoning*. Dead reckoning extrapolates a remote entity's movement from the last known position and first (or higher order) derivation (e.g., speed vector, acceleration vector, etc.). When the sending entity deviates from the previous trajectory, it will broadcast this updated information. As a result, the trajectory will be gradually adjusted to match the parameters of the new movement equation. Dead reckoning works well for entities which perform regular movements. Erratic movements, however, are unlikely to benefit from dead reckoning.

Entity broadcast packets must then be examined at every host, even if the information is not intended for that participant's AOI. This information filtering at the application level can cause major performance penalties for that host, especially if the rendering module is already processor intensive. Moreover, the network can become flooded with unwanted traffic. These circumstances make a broadcast approach non-scalable.

In SimNet, distance-based AOI filtering is performed at the receiver. This endpoint-based filtering scheme, however, incurs unnecessary network load with messages that are irrelevant to most receivers. In addition, all receivers must expend processing power to determine whether a packet relates to an entity within their AOI. This endpoint-based filtering does not scale well to large numbers of participants [110].

Peer-to-Peer Multicast Based on Isomorphic Regions

In an effort to increase scalability, some systems use *region-based multicast* instead of broadcast. The world is divided into spatial regions and each region is associated with a separate multicast group to disseminate information about updates to that region. Participants interested in a region attend to the relevant multicast group. Therefore, a single host's load does not grow linearly in relation to the overall number of connected users, but only in proportion to the number of nearby users. This means that a multi-user NET-VE can become extremely large as long as the users are sufficiently spread out. Filtering is therefore taking place at the hardware level, since a participant only joins the multicast groups of regions it is interested in.

In NPSNET, for example, a world terrain is divided into fixed-size hexagonal cells, which are mapped onto separate multicast addresses. Cells are embedded in a global coordinate system; every participant is a-priori aware of the layout and the mapping onto multicast addresses. Every participant sends update packages to the multicast group of its current cell, while at the same time subscribing to many surrounding hexagonal cells to fully cover the AOI radius. A major problem is finding the right cell size. Larger cells mean that participants receive much irrelevant information; smaller cells mean that they need to subscribe to many more multicast groups. Also, since the cell size is fixed, this approach fails if entities are not distributed evenly over the world area, e.g., if "clumping" occurs within some cells.

For cell-based multicast, a participant must identify the cell it is located in and send updates to the associated multicast group. To receive data from other participants included in its area of interest, each participant has to join the multicast groups associated with the cells that intersect with its area of interest. The main difficulty with the cell-based approach is finding of the right cell size.

Peer-to-Peer Multicast Based on Variable-Sized Regions

Spline [20] is another example of a *cell-based multicast* peer-to-peer NET-VE. A world is comprised of a set of chunks called *locales*, with disjoint sets of entities and explicit lists of neighbouring locales. Every locale is owned by a single machine, but may be copied onto other interested clients. However, only the owner can change an entity's state in order to avoid writers/writers conflicts. Locales in SPLINE can be of any (fixed) size or shape. This allows the NET-VE designer to partition the world so as to try and avoid clumping of too many entities within the same locale. However, clumping can still occur, since a designer can never know in advance how many dynamic entities might enter a locale during execution time and the size of locales is fixed. The set of all locales available over a network provides the illusion of a large, continuous world. Interestingly, the "glueing" of locales does not need to follow Euclidian geometry (e.g., the interior of a building may be larger than its external dimensions; space might be warped, shrinked, etc.). This gives interesting opportunities to link locales it might also confuse users navigating through such a world. No machine maintains a complete copy of the world. A participant's AOI is based on predetermined visibility constraints of the participant-current locale. e.g., locales can be used to omit the rendering of the interiors of buildings when they are not visible from the outside. While every locale is hosted by a participant's machine and replicated on the machines of other, interested participants.

Another issue is that activities cannot span several locales. e.g., a button in one locale cannot control a bulb in another locale. Moreover, locales cannot be nested (e.g., with the top locale providing a low-detail overview of a city from far away, containing several levels of finer-grained locales).

Participation as a content provider in this system is very simple. One just designs a new locale and "stitches" it to some already existing locales on the Internet. Due to the non-Euclidian properties of Spline-space, it is easy to create hubs which link many different locales from a space (portal). Spline has recently been commercialised by Mitsubishi in the Schmoozer software (http://www.schmoozer.net).

One problem with region-based multicast communication is finding the right size for regions. In SPLINE and NPSNET, if cells/locales are too small, a client would have to subscribe to too many multicast groups, and if the cell/locale were too large, a client would have to listen to other clients it did not care about. This makes it difficult for a designer or implementer to find the "right" size for a region. Scalability is impeded if too many users gather in the same region, e.g., a football stadium.

Centralised Server with Multiple Clients

While peer-to-peer systems allow low-latency communications between participants, they frequently rely on network multicast, which is not yet supported on an Internet-wide scale.

In client-server systems, all communication is relayed through a central server. This has the advantage of providing better state equality, since the server can impose a global ordering on all events occurring in the world. The disadvantage is that communication is slower than in peer-to-peer systems, where participants are directly connected. However, this method allows a server to do processing and filtering of messages before propagating them to other participants; a central server can better control the quality and type of information sent to a given client and therefore adapt better to heterogeneous clients. Moreover, broadcast or multicast as required by peer-to-peer systems is not yet deployed on an Internet-scale. In addition, servers can determine if a client has made an illegal move (e.g., into a wall) or it can decide to limit sending the user's message to a subset of other users.

The scalability of a client-server system is limited by the load that the server can handle, as every update message is unicast through the server. Therefore, a centralised server can act as a bottleneck that prohibits scalability. We will focus on the client-server model and provide a parallel server in order to guarantee scalability.

Intermediate Servers

Systems can also use a hybrid scheme, where intermediate servers interact in peer-to-peer fashion, with clients connecting to a geographically nearby server. This helps to make the server less of a bottleneck. The intermediate servers can also detect, which user is near to which other one and dynamically route messages accordingly and therefore scale better to large numbers of users.

A.6 Multi-user VRML

A number of VRML-based multi-user NET-VEs exist. Typically these are implemented as client-server systems. VNET [149;182]for example, makes use of the EAI interface on the client-side to collect information about a user's movements and update the avatar state of other simultaneous participants. Unlike the system described in this paper, the VNET server acts merely as a communications relay for such updates to avatars. Commercially available implementations of VRML multi-user systems extend this scheme with authentication and simple, server-controlled entities, so called robots (e.g. blaxxun, ParallelGraphics). These systems are neither scalable to large-scale worlds, nor to complex behaviours as both scene graph and behaviour evaluation are actively replicated on all clients. i.e., the complete scene graph is downloaded to all clients and the server is responsible for the replication of avatar movements across all clients.

ActiveWorlds (http://www.activeworlds.com) is also based on the idea of a central server to broadcast avatar movements. In addition, the ActiveWorlds server performs distance based culling: clients are only informed about entities and other client avatars that are in their respective proximity. Therefore, ActiveWorlds is scalable in terms of world size and number of participants. Participants can upload VRML files as new entities to the server and thereby actively "build" the world. Besides entities addition and avatar movement, dynamics support is limited to simple animations. Unlike the servers described in this paper, the ActiveWorlds server does not perform any behaviour evaluation.

Chenney et al. [44] examine dynamics culling for invisible parts of a scene. While this approach is useful for physics-based simulations, which can easily be extrapolated over time, this method cannot be used for general, unpredictable behavior evaluation.

In the domain of computer games, BSP traversal with pre-computed visibility sets and portal rendering are frequently used to greatly reduce rendering time for large worlds. These techniques work well for mainly static worlds with movement restricted to simple animations and human or computer controlled actors. In multiplayer network games, such as Quake or Doom, the dataset describing the world and avatars is available to the client a-priori; no dynamic download takes place. The server (usually one of the player's machines) performs replication of client state (e.g., position, weapons usage) across all participants. This scheme restricts maximum world size to the clients' storage capacity. Levels are therefore rather small and uniform

Bibliography

1. Communications of the ACM. Vol. 44, 2001.
2. Achauer, Bruno. The DOWL Distributed Object-Oriented Language. Communications of the ACM. 1993 Sep; 36(9):48-55.
3. Agha, Gul. Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press; 1986.
4. Agha, Gul and Hewitt, Carl. Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming. Shriver and Wegner, editors: MIT Press; 1987; pp. 49-74.
5. Agha, Gul; Kim, W., and Panwar, R. Actor Languages for Specification of Parallel Computations. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1994.
6. Agha, Gul; Wegner, Peter, and Yonezawa, Akinori. Research Directions in Concurrent Object-Oriented Programming. MIT Press; 1993.
7. Allen, Randy and Kennedy, Ken. Automatic Translation of Fortran Programs to Vector Form. ACM Transactions on Programming Languages and Systems. 1987; 9(4):491-542.
8. America, P. H. M and Rutten, J. J. M. M. A Parallel Object-Oriented Language: Design and Semantic Foundation. de Bakker, J. W, editor. Languages for Parallel Architectures - Design, Semantics, Implementation Models. 1989; pp. 1-49.
9. America, Pierre. Inheritance and Subtyping in a Parallel Object-Oriented Language. ECOOP '87 Conference Proceedings: Springer Verlag; 1987pp. 234-242.
10. America, Pierre. POOL-T: A Parallel Object-Oriented Language. Yonezawa, Akinori and Tokoro, Mario. Object-Oriented Concurrent Programming. MIT Press; 1987; pp. 199-220.
11. America, Pierre and van der Linden, Frank. A Parallel Object-Oriented Language with Inheritance and Subtyping. OOPSLA/ECOOP '90 Proceedings, ACM Sigplan Notices. 1990; 25(10):161-168.
12. Anderson, Thomas; Culler, David, and Patterson, David. A Case for Networks of Workstations: NOW. IEEE Micro. 1995; (February).

13. Annot, J. K. and den Haan, P. A. M. POOL and DOOM: The Object Oriented Approach. Treleaven, P. C., editor. Parallel Computers, Object-Oriented, Functional, Logic. John Wiley & Sons; 1990; pp. 47-80.

14. Aridor, Yariv; Cohan, Shimon, and Yehudai, Amiram. SYMPAL: A Software Environment for Implicit Concurrent Object-Oriented Programming. Object Oriented Systems. 1997; 4:53-81.

15. Aridor, Yariv; Factor, Michael, and Teperman, Avi. cJVM: A Single System Image of a JVM on a Cluster. 1999 International Conference on Parallel Processing ; Wakamatsu, Japan. 1999.

16. Atkinson, Colin. Object-Oriented Reuse, Concurrency and Distribution. ACM Press; 1991.

17. Backus, John. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Communications of the ACM. 1978; 21(8):613-641.

18. Baker, M.; Carpenter, B.; Fox, G.; Ko, S. H., and Li, X. mpiJava: a Java MPI Interface. First UK Workshop, Java for High Performance Network Computing (within EUROPAR '98); 1998.

19. Baker, Mark and Buyya, Rajkumar. Cluster Computing: The Commodity Supercomputing. Software Practice and Experience. 1999 May; 29(6).

20. Barrus, John W.; Waters, Richard C., and Anderson, David B. Locales: Supporting Large Multiuser Virtual Environments. IEEE Computer Graphics and Applications. 1996 Nov; 16(6):50-67.

21. Benford, Steve; Greenhalgh, Chris; Rodden, Tom, and Pycock, James. Collaborative Virtual Environments. 2001 Jul; 44, (7): 79-86.

22. Beust, Cedric (beust@ilog.fr). Re: Synchronous vs. Asynchronous. E-mail to: Mailing List for Discussion of JavaSoft's Remote Method Invocation (RMI-Users@javasoft.com). 1997 May 12.

23. Bik, Art and Gannon, Dennis. javab - A Prototype Bytecode Parallelization Tool. Technical Report. Computer Science Department, Indiana University; 1998.

24. Bik, Art. javar - A Prototype Java Restructuring Compiler. Technical Report. Computer Science Department, Indiana University; 1998; 487.

25. Birrell, A. and Nelson, P. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems. 1984 Feb; 2(1).

26. Bjarne, Stroustrup. The C++ Programming Language. 3rd. ed. Addison-Wesley; 1997.

27. Bloom, T. Evaluating Sychronisation Mechanisms. Proceedings of the Seventh Symposium on OS Principles; 1979: pp 23-32.

28. Bräunl, Thomas. Parallel Programming - An Introduction. Prentice Hall; 1993.

29. Briot, Jean-Pierre; Guerraoui, Rachid, and Lohr, Klaus-Peter. Concurrency and Distribution in Object-Oriented Programming. ACM Computing Surveys. 1998; 30(3).

30. Buyya, Rajkumar. High Performance Cluster Computing. Prentice Hall; vol. 1, 1999.

31. Buyya, Rajkumar. High Performance Cluster Computing. Prentice Hall; vol. 2, 1999.

32. Cabri, Giacomo; Leonardi, Letizia, and Zambonelli, Franco. Weak and Strong Mobility in Mobile Agent Applications. Proceedings of the 2nd International Conference and Exhibition on The Practical Application of Java (PAJAVA 2000); Manchester, UK.

33. Calvin, J. et al. The SIMNET Virtual World Architecture. Proceedings of the IEEE Virtual Reality Annual International Symposium; Seattle, WA. IEEE Computer Press; 1993: 450-455.

34. Cann, D. C. Retire Fortran? Communications of the ACM. 1992; 35(8):81-89.

35. Carey, Rikk and Bell, Gavin. The Annotated VRML 2.0 Reference Manual. Addison-Wesley; 1997.

36. Carey, Rikk; Bell, Gavin, and Marrin, Chris. ISO/IEC 14772-1: 1997 Virtual Reality Modelling Language (VRML97). 1997.

37. Caromel, Denis. Towards a Method of Object-Oriented Concurrent Programming. Communications of the ACM. 1993; 36(9):90-102.

38. Caromel, Denis; Belloncle, Fabrice, and Roudier, Yves. The C++// System. Wilson, G. and Lu, P., editors. Parallel Programming Using C++. MIT Press; 1996; pp. 257-296.

39. Caromel, Denis and Vayssiere, Julien. Towards Seamless Computing and Metacomputing in Java.Concurrency: Practice and Experience; 1998; 10(11-13); pp. 1043-1061.

40. Carpenter, Bryan; Chang, Yuh-Jye; Fox, Geoffrey; Leskiw, Donald, and Li, Xiaming. Emperiments with 'HP Java'. Concurrency, Practice and Experience. 1997; 9(6):633-648.

41. Carriero, N. and Gelernter, D. How to Write Parallel Programs. A First Course. MIT Press; 1990.

42. Carriero, Nicholas and Gelernter, David. A Computational Model of Everything. CACM. 2001 Nov; 44, (11): 77-81.

43. Carter, John B. Design of the Muning distributed shared memory system. Journal of Parallel and Distributed Computing. 1995 Sep; 29(2):219-227.

44. Chenney, Stephen; Ichnowski, Jeffrey, and Forsyth, David. Dynamics Modelling and Culling. IEEE Computer Graphics and Applications. 1999; 19(2):79-87.

45. Chien, Andrew and Karamcheti, Vijay. Concert - Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. Conference on Supercomputing '93; 1993pp. 598-607.

46. Chin, Roger and Chanson, Samuel. Distributed, Object-Based Programming Systems. ACM Computing Surveys. 1991; 23(1):91-124.

47. Cugola, Gianpaolo; Ghezzi, Carlo; Picco, Gian Pietro, and Vigna, Giovanni. Analyzing Mobile Code Languages. Vitek, Jan and Tschudin, Christian, editors. Mobile Object Systems: Towards the Programmable Internet. Springer Verlag; 1997; pp. 93-110.

48. Dahl, Ole-Johan and Nygaard, Kristen. SIMULA - An ALGOL-based simulation language. CACM. 1966 Sep; 9(9):671-678.

49. de Bruin, Hans. BCOOPL: Basic concurrent object-oriented programming language. Software Practice and Experience. 2000; 30:849-894.

50. Decouchant, D.; Krakowiak, S.; Meysembourg, M.; Riveill, M., and Rousset de Pina, X. A Synchronzation Mechanism for Typed Objects in a Distributed System. ACM SIGPLAN Notices. 1989 Apr; 24(4):105-107.

51. Detmold, Henry and Oudshoorn, Michael J. [Technical Report]. Responsibilities: Linguistic Support for Safe and Flexible Remote Communication. Department of Computer Science, University of Adelaide; 1994 Nov; 94-12.

52. Dincer, K. Jmpi and a Performance Instrumentation Analysis and Visualisation Tool for Jmpi. First UK Workshop on Java for High Performance Network Computing; Southhampton. 1998.

53. Dongarra, Jack; Otto, Steve; Snir, Marc, and Walker, David. An Introduction to the MPI Standard [Technical Report]. University of Tennessee; 1995; CS-95-274.

54. Eager, Derek; Lazowska, Edward, and Zahorjan, John. Adaptive Load Balancing in Homogeneous Distributed Systems. IEEE Transactions on Software Engineering. 1986; SE-12(5):662-675.

55. Ekanadham, K. A Perspective on Id. Szymanski, B., editor. Parallel Functional Languages and Compilers. ACM Press; 1991; pp. 197-254.

56. Ellis, C. A. and Gibbs, S. J. Active Objects: Realities and Possibilities. Kim, W. and Lochovsky, F. H., editors. Object-Oriented Concepts, Databases and Applications. ACM Press; 1989; pp. 561-572.

57. Excelsior, LLC. Excelsior JET [Web Page]. 2002; Accessed 2002. Available at: http://www.excelsior-usa.com/jet.html.

58. Falkner, Katrina E.; Coddington, Paul D., and Oudshoorn, Michael J. Implementing Asynchronous Remote Method Invocation in Java. Proc. Parallel and Real-Time Systems (PART'99); Melbourne. 1999.

59. Ferenczi, Szabolcs. Guarded Methods vs. Inheritance Anomaly - Inheritance Anomaly Solved by Nested Method Calls. ACM SIGPLAN Notices. 1995 Feb; 30(2):49-58.

60. Ferrari, Adam. JPVM: Network Parallel Computing in Java. ACM Workshop on Java for High-Performance Network Computing; Palo Alto. University of Virginia; 1997.

61. Flynn, M. J. and Rudd, K. W. Parallel Architectures. ACM Computing Surveys. 1996 Mar; 28(1):67-70.

62. Foster, Ian and Kesselman, Carl. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers; 1998 Jul.

63. Foster, Ian; Thiruvathukal, G. K., and Tuecke, S. Technologies for Ubiquitous Supercomputing: A Java Interface to the Nexus Communication System. Concurrency: Practice & Experience. 1997 Jun.

64. Fox, Geoffrey and Furmanski, Wojtek. Java for Parallel Computing and as a General Language for Scientific and Engineering Simulation and Modeling. Concurrency, Practice and Experience. 1997; 9(6):415-425.

65. Fox, Geoffrey C., Roy D. Williams, and Paul C. Messina. Parallel Computing Works! Morgan Kaufmann Publishers; 1994 Apr.

66. Free Software Foundation, Boston, USA. The GNU Compiler for the Javatm Programming Language [Web Page]. 2001; Accessed 2001. Available at: http://gcc.gnu.org/java/.

67. Funkhouser, Thomas A.; Sequin, Carlo H., and Teller, Seth J. Management of Large Amounts of Data in Interactive Building Walkthroughs. ACM SIGGRAPH (Special Issue on 1992 Symposium on Interactive 3D Graphics, Cambridge, MA). 1992; 25(2):11-20.

68. Geist, Al; Beguelin, Adam; Dongarra, Jack; Jiang, Weicheng; Manchek, Robert, and Sunderam, Vaidy. PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing. Cambridge, MA: MIT Press; 1994.

69. Gelernter, David. Current Research on Linda. Metayer; Banatre, J. P., and Le, D., editors. Research Directions in High-Level Parallel Programming Languages. Springer Verlag; 1991; pp. 74-76.

70. Getov, Vladimir; Flynn-Hummel, Susan, and Mintchev, Sava. High-Performance Parallel Programming in Java: Exploiting Native Libraries. ACM 1998 Workshop on Java for High-Performance Network Computing; 1998.

71. Global Grid Form. Grid Computing Info Centre [Web Page]. Available at: http://www.gridcomputing.com/.

72. Gosling, James; Joy, Bill, and Steele, Guy. Java Programming Language. Addison-Wesley; 1996.

73. Greenhalgh, Chris. Large Scale Collaborative Virtual Environments [PhD Thesis]: University of Nottingham; 1997 Oct.

74. Grimshaw, Andrew. Easy-to-Use Object-Oriented Parallel Processing with Mentat. IEEE Computer. 1993; (May):39-50.

75. Grimshaw, Andrew. Object-Oriented Parallel Processing with Mentat. 1998. available online as http://legion.virginia.edu/papers/is.ps.

76. Guessoum, Zahia and Briot, Jean-Pierre. From Active Objects to Autonomous Agents. IEEE Concurrency. 1999 Jul-1999 Sep 30; 7(3):68-76.

77. Halstead, Robert jr. Multilisp: A Language for Concurrent Symbolic Computation. ACM Transactions on Programming Languages and Systems. 1985; 7(4):501-538.

78. Herath, J.; Yuba, T., and Saito, N. Dataflow Computing. Parallel Algorithms and Architectures. 1987 May; 269:25-36.

79. Hewitt, Carl . Viewing Control Structures as Patterns of Passing Messages. Artificial Intelligence. 1977; 8(3):323-364.

80. Hilderink, Gerald; Broenik, Jan; Vervoort, Wiek, and Bakkers, Andre. Communicating Java Threads. WoTUG-20 conference; Enschede, The Netherlands. IOS Press; 1997: pp. 48-76. ISBN: 90 5199 336 6.

81. Hirano, Satoshi; Yasu, Yoshiji, and Igarashi, Hirotaka. Performance Evaluation of Popular Distributed Object Technologies for Java. ACM Java Grande Conference; Stanford University, Palo Alto, CA. 1998. ISBN: http://ring.etl.go.jp/openlab/horb.

82. Hoare, C. A. R. Communicating Sequential Processes. CACM. 1978 Aug; 21(8):666-677.

83. Hutchinson, Norman. Emerald: An Object-Based Language for Distributed Programming [PhD Thesis]: University of Washington; 1987.

84. Hutchinson, Norman; Raj, Rajendra; Black, Andrew; Leyv, Henry, and Jul, Eric. The Emerald Programming Language [Technical Report]. Dept. of Computer Science, University of British Columbia; 1991 Oct.

85. Hyde, Daniel C. Java and Different Flavors of Parallel Programming Models. Buyya, Rajkumar, editor. High Performance Cluster Computing. Prentice Hall; 1999; pp. 274-290.

86. Ishikawa, Yutaka and Tokoro, Mario. Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation. Yonezawa, Akinori and Tokoro, Mario. Object-Oriented Concurrent Programming. MIT Press; 1987; pp. 159-198.

87. Java Community Process. JSR 51: New I/O APIs for the Java Platform [Web Page]. 2002 May. Available at: http://jcp.org/jsr/detail/51.jsp.

88. Java Grande Forum. Available at: http://www.javagrande.org.

89. Jul, Eric; Levy, Henri; Hutchinson, Norman, and Black, Andrew. Fine-Grained Mobility in the Emerald System. ACM Transactions on Computer Systems. 1988; 6(1):109-133.

90. Kafura, Dennis; Washabaugh, Doug, and Nelson, Jeff. Garbage Collection of Actors. Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages and Applications : 126-134.

91. Kafura, Dennis G. and Lee, Keung Hae. Inheritance in Actor-Based Concurrent Object-Oriented Languages. ECOOP '89 Conference Proceedings: Cambridge University Press; 1989: 131-145.

92. Karaorman, M. and Bruno, J. Introducing Concurrency to a Sequential Language. Communications of the ACM. 1993 Sep; 36(9):103-115.

93. Katholieke Universiteit Leuven. Correlate home page [Web Page]. Available at: http://www.cs.kuleuven.ac.be/~distrinet/projects/CORRELATE/index.html.

94. Kazi, Iffat H.; Chen, Howard H.; Stanley, Berdenia, and Lilja, David J. Techniques for Obtaining High Performance in Java Programs. ACM Computing Surveys. 2000 Sep; 32(3):213-240.

95. Kielmann, Thilo; Hatcher, Philip; Bouge, Luc, and Bal, Henri E. Enabling Java for High-Performance Computing. CACM. 2001 Oct; 44(10):110-117.

96. Kim, Woo Young and Agha, Gul. Efficient Support of Location Transparency in Concurrent Object-Oriented Programming Languages. Supercomputing 1995; San Diego, CA USA. ACM; 1995: pp. 39-es.

97. Kiniry, J. and Zimmermann, D. A Hands-on Look at Java Mobile Agents. IEEE Internet Computing. 1997 Jul-1997 Aug 31; 1(4):21-33.

98. Krishnan, Sanjeev. Automatic Runtime Optimizations for Parallel Object-Oriented Programming: University of Illinois at Urbana-Champaign; 1996.

99. Lam, Monica [Technical Report]. An Efficient Shared Memory Layer for Distributed Memory Machines. Stanford University, CA; 1994; CSL-TR-94-627.

100. Le Fessant, Fabrice; Piumarta, Ian, and Shapiro, Marc. An Implementation of Complete, Asynchronous, Distributed Garbage Collection. Conference on Programming Language Design and Implementation (PLDI); Montreal, Canada. ACM SIGPLAN; 1998.

101. Lea, Doug. Concurrent Programming in Java - Design Principles and Patterns. second ed. Addison Wesley; 2000.

102. Lewis, Ted and El-Rewini, Hesham. Loop Scheduling and Parallelising Serial Programs. In Introduction to Parallel Computing. Englewood Cliffs, NJ: Prentice Hall; 1992; pp. 283-346.

103. Lieberman, Henry. Concurrent Object-Oriented Programming in Act 1. Yonezawa, Akinori and Tokoro, Mario. Object-Oriented Concurrent Programming. MIT Press; 1987; pp. 9-36.

104. Lindholm, Tim and Yellin, Frank. The Java Virtual Machine Specification. second ed. Addison-Wesley; 1999.

105. Liskov, B.; Herlihy, M., and Gilbert, L. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. Proceedings of the 13th ACM Symposium on Principles of Programming Languages; St. Petersburg, FL; 1986.

106. Liskov, B. and Shira, L. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation; 1988 Jun; pp. 260-267.

107. Loehr, Klaus Peter. Concurrency Annotations for Reusable Software. CACM. 36(9):81-89.

108. Maassen, Jason; Nieuwpoort, Rob van; Veldema, Ronald; Bal, Henri; Kielmann, Thilo; Jacobs, Ceriel, and Hofman, Rutger. Efficient Java RMI for Parallel Programming. Vrije Universiteit Amsterdam; 2000 Mar.

109. Macedonia, M. R.; Brutzman, D. P.; Zyda, M. J.; Pratt, D. R.; Barham, P. T.; Falby, J., and Locke, J. NPSNET: A multi-player 3D virtual environment over the Internet. Proceedings of the 1995 Symposium on Interactive 3D Graphics; Monterey, CA. ACM SIGGGRAPH.

110. Macedonia, Michael R. and Zyda, Michael J. A Taxonomy for Networked Virtual Environments. IEEE Multimedia. 1997 Jan-1997 Mar 31; 48-56.

111. Marrin, Chris. Proposal for a VRML 2.0 Informative Annex, External Authoring Interface Reference. 1997.

112. MasPar VAST-2 [User's Guide, MasPar System Documentation]. MasPar Computer Corporation. Version 1.2. 1992 FebDPN 9300-9035.

113. Matsuoka, Satoshi and Yonezawa, Akinori. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. Agha, Gul; Wegner, Peter, and Yonezawa, Akinori. Research Directions in Concurrent Object-Oriented Programming. MIT Press; 1993; pp. 107-150.

114. Maybury, Mark; D'Amore, Ray, and House, David. Expert Finding for Collaborative Virtual Environments . CACM. 2001 Dec; 44(12):55-56.

115. Meieran, Eugene. 21st Century Semiconductor Manufacturing Capabilities. Intel Technology Journal. 1998; Q4.

116. Meyer, Bertrand. Systematic Concurrent Object-Oriented Programming. Communications of the ACM. 1993; 36(9):56-80.

117. Meyer, Bertrand. Object-Oriented Software Construction. second ed. Prentice Hall, Englewood Cliffs, NJ; 1997.

118. Miller, D. and Thorpe, J. A. SIMNET: The advent of simulator networking. Proceedings of the IEEE. 1995 Aug; 83(8):1114-1123.

119. Milner, Robin. Communication and Concurrency (International Series in Computer Science). Prentice Hall; 1989.

120. Milner, Robin. The Polyadic π-Calculus: A Tutorial. Hamer, F. L; Brauer, W., and Schwichtenberger, H., editors. Logic and Algebra of Specification. Springer-Verlag; 1993.

121. Mohr, Eric; Kranz, D. A., and Halstead, R. H. jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. IEEE Transactions on Parallel and Distributed Systems. 1991; 2(3):264-280.

122. Moreira, Jose. Closing The Performance Gap between Java and Fortran in Technical Computing. First UK Workshop on Java for High Performance Computing (within EUROPAR '98); Southampton, UK. 1998.

123. Moreira, José E.; Midkiff, Samuel P.; Gupta, Manish; Artigas, Pedro V.; Wu, Peng, and Almasi, George. The NINJA Project. Communications of the ACM. 2001 Oct; 44(10):102-109.

124. Morgan, Graham and Rischbeck, Thomas. Implementing Scalable Networked Virtual Environments Using Replicated VRML Servers. Workshop on Object-Oriented Reliable Distributed Systems (WOODS 2000); Nuremberg, Germany.

125. Mukherjee, Nandini. On the Effectiveness of Feedback-guided Parallelisation. Manchester: University of Manchester; 1999 Sep.

126. Murata, Kenichi; Horspool, R. Nigel; Manning, Eric G.; Yokote, Yasuhiko, and Tokoro, Mario. Unification of Active and Passive Objects in an Object-Oriented Operating System. Proceedings of 1995 Int. Workshop of Object Orientation in Operating Systems (IWOOOS'95).

127. Nagle, John. Congestion Control in IP/TCP Internetworks. Ford Aerospace and Communications Corporation; 1984 Jan; RFC896.

128. Nester, Christian; Philippsen, Michael, and Haumacher, Bernd. A More Efficient RMI for Java. Proceedings of the ACM 1999 Conference on Java Grande; Palo Alto, CA USA. ACM Press; 1999: pp. 152-159.

129. Nierstrasz, Oscar. A Tour of Hybrid
A Language for Programming with Active Objects. Mandrioli, D. and Meyer, B., editors. Advances in Object-Oriented Software Engineering. Prentice-Hall; 1992; pp. 167-182.

130. Nierstrasz, Oscar. Composing Active Objects - The Next 700 Concurrent Object-Oriented Languages. Agha, Gul; Wegner, Peter, and Yonezawa, Akinori, editors. Research Directions in Concurrent Object-Oriented Programming. MIT Press; 1993; pp. 151-171.

131. Nitzberg, Bill and Lo, Virginia. Distributed Shared Memory: A Survey of Issues and Algorithms. IEEE Computer. 1991 Aug; 52-60.

132. Open Systems Laboratory, Department of Computer Science, University of Illinois. The Actor Foundry home page [Web Page]. Available at: http://www-osl.cs.uiuc.edu/foundry.

133. Papathomas, Michael. Concurrency Issues in Object-Oriented Programming Languages. Tsichritzis, D. C., Editor. Object-Oriented Development. Geneva, CH: Université de Genéve, Centre Universitaire d'Informatique; 1989; pp. 207-245.

134. Papathomas, Michael and Nierstrasz, Oscar M. Supporting Software Reuse in Concurrent Object-Oriented Languages: Exploring the Language Design Space. Tsichritzis, D., editor. Object Composition. Geneva: Centre Universitaire d'Informatique; 1991 Jun; pp. 189-204.

135. Parastatidis, Savas. Run-time Support for Parallel Object-Oriented Computing [PhD Thesis]. Newcastle Upon Tyne, UK: University of Newcastle Upon Tyne; 2000 Jan.

136. Pfister, Gregory F. In Search of Clusters. second ed. Prentice Hall; 1995.

137. Philippsen, Michael. [Technical Report]. Imperative Concurrent Object-Oriented Languages. Berkeley, CA: International Computer Science Institute; 1995 Aug; TR-95-050.

138. Philippsen, Michael. [Technical Report]. Imperative Concurrent Object-Oriented Languages: An Annotated Bibliography. Berkeley, CA: International Computer Science Institute; 1995 Aug; TR-95-049.

139. Philippsen, Michael and Haumacher, Bernhard. More Efficient Object Serialization. Lecture Notes in Computer Science. 1999; 1586:718-.

140. Philippsen, Michael and Zenger, Matthias. JavaParty - Transparent Remote Objects in Java. Concurrency: Practice and Experience. 1997; 9(11):1225-1242.

141. Plainfosse, David and Shapiro, Marc. A Survey of Distributed Garbage Collection Techniques. Proc. Int. Workshop on Memory Management; Kinross, Scotland. 1995.

142. Pooley, R. J. An Introduction to Programming in SIMULA. Blackwell Scientific Publications; 1987.

143. Pramanick, Ira. MPI and PVM Programming. Buyya, Rajkumar, editor. High Performance Cluster Computing. Prentice Hall; 1999; pp. 48-86.

144. Raje, Rajeev R.; Williams, Joseph I., and Boyles, Michael. An Asynchronous Remote Method Invocation (ARMI) Mechanism for Java. Concurrency: Practice and Experience. 1997; 9(11):1207-1211.

145. Reddy, Martin. Perceptually Modulated Level of Detail for Virtual Environments [PhD Dissertation]: University of Edinburgh; 1997.

146. Rinard, Martin C.; Scales, Daniel J., and Lam, Monica S. Jade: A High-Level, Machine Independent Language for Parallel Programming. IEEE Computer. 1993; 26(6):28-38.

147. Rischbeck, Thomas and Watson, Paul. A Parallel VRML97 Server Based on Active Objects. Palma, Jose M. L. M.; Dongarra, Jack, and Hernandez, Vincente. VECPAR'2000 Selected Papers and Invited Talks from the 4th International Conference on Vector and Parallel Processing. FEUP, Porto, Portugal: Springer Verlag; 2001; pp. 47-60.

148. Rischbeck Thomas and Watson Paul. A Scalable, Multi-user VRML Server. IEEE VR2002 Conference; Orlando, FL. 2002.

149. Robinson, John L.; Stewart, John A., and Labbe, Isabelle. MVIP-Audio enabled multicast VNet . Proceedings of the Web3D-VRML 2000 fifth symposium on Virtual reality modeling language ; Monterey, CA, USA. 103-109.

150. Roulo, Mark. Accelerate your Java apps! Javaworld. 1998 Sep.

151. Schmalstieg, Dieter. LODESTAR - An Octree-Base Level of Detail Generator for VRML. SIGGRAPH Symposium on Virtual Reality Modeling Language (VRML'97); Monterey CA, USA. 1997.

152. Schmidt, Douglas; Stal, Michael; Rohnert, Hans, and Buschmann, Frank. Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects. John Wiley & Sons; 2000.

153. Schmidt, Douglas C. and Fayad, Mohamed E. Lessons Learned Building Reuseable OO Frameworks for Distributed Software. Communications of the ACM. 1997 Oct; 40(10):85-87.

154. Shah, Apu. JavaSpaces: An Object Sharing Framework That May Redefine Distributed Computing [Web Page]. 1997 Oct 21. Available at: http://www.developer.com/journal/techfocus/n_tech_javaspaces.html.

155. Silva, Luis Moura E and Buyya, Rajkumar. Parallel Programming Models and Paradigms. Buyya, Rajkumar, editor. High Performance Cluster Computing. Prentice Hall; 1999; pp. 4-27.

156. Singhal, Sandeep and Zyda, Michael. Networked Virtual Enviroments. ACM Press; 1999.

157. Skillicorn, David and Talia, Domenico. Models and Languages for Parallel Computation. ACM Computing Surveys. 1998; 30(2).

158. Sommerville, Ian. Software Engineering. sixth ed. Addison-Wesley; 2001.

159. Squyres, Jeffery M.; McCandless, Brian C., and Lumsdaine, Andrew. Object Oriented MPI: A Class Library for the Message Passing Interface. Parallel Object-Oriented Methods and Applications (POOMA '96); Santa Fe, NM.

160. Steinberg, Daniel. The Java Grande Forum Pushes Java Toward New Heights. Javaworld. 1999; (September).

161. Sterling, Thomas; Messina, Paul, and Smith, Paul. Enabling Technologies for Petaflops Computing. MIT Press; 1995.

162. Sun Microsystems. RMI [Web Page]. 1998.

163. Sun Microsystems, Inc. Java Native Interface Specification [Web Page]. 1997 May 16. Available at: http://java.sun.com/products//jdk/1.2/docs/guide/jni/spec/jniTOC.doc.html.

164. Sun Microsystems, Inc. Java Object Serialization Specification [Web Page]. 1998 Nov. Available at: ftp://ftp.javasoft.com/docs/jdk1.2/serial-spec-JDK1.2.pdf.

165. Sunderam, V. S. PVM: A Framework for Parallel Distributed Computing. Concurrency: Practice & Experience. 1990 Dec; 2(4):315-339.

166. Tada, Yusuke. Tara: Actor-Based Object-Oriented Language for Efficient Distributed Software Development: Kyoto University; 1998.

167. Taura, Kenjiro; Matsuoka, Satoshi, and Yonezawa, Akinori. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages on Stock Multicomputers. ACM SIGPLAN Notices. 28(7):218-228.

168. . Java Grande Forum Report: Making Java work for high-end ComputingThiruvathukal, George K. ; Breg, Fabian; Boisvert, Ronald; Darcy, Joseph; Fox, Geoffrey C.; Gannon, Dennis; Hassanzadeh, Siamak; Moreira, Jose; Philippsen, Michael; Pozo, Roldan, and Snir, Marc. Supercomputing '98: International Conference on High Performance Computing and Communications; Orlando, FL. panel handout.

169. Tomlinson, C. and Singh, V. Inheritance and Synchronization with Enabled Sets. Proceedings of the OOPSLA '89 Conference on Object-oriented Programming Systems, Languages and Applications: 103-112.

170. Tower Technology Corporation. TowerJ Java Virtual Machine [Web Page]. 2001; Accessed 2001. Available at: http://www.towerj.com.

171. Tripathi, Anand; van Oosten, James, and Miller, Robert. Object-Oriented Concurrent Programming Languages and Systems. JOOP. 1999; (November/December):22-29.

172. van Nieuwpoort, Rob; Maassen, Jason; Bal, Henri E.; Kielmann, Thilo, and Veldema, Ronald. Wide-Area Parallel Computing in Java. Proceedings of the ACM 1999 Conference on Java Grande; Palo Alto, CA USA. ACM PressPalo Alto, CA USA; 1999.

173. Van Oeyen, Johan; Bijnens, Stijn; Joosen, Wouter; Robben, Bert; Matthijs Frank, and Verbaeten, Pierre. A Flexible Object Support System as Runtime for Concurrent Object-Oriented Languages. Zimmermann, Chris, editor. Advances in Object-Oriented Metalevel and Reflective Architectures. CRC Press, Inc.; 1996; pp. 219-236.

174. Veldema, Robert; Nieuwport, Rob van; Maassen, Jason; Bal, Henri E., and Plaat, Aske. Efficient Remote Method Invocation. Vrije Universiteit Amsterdam, The Netherlands; 1998 Sep; IR-450. Technical Report.

175. Vinoski, Steve. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. IEEE Communications. 1997 Feb; 14(2).

176. Vinoski, Steve. New Features for CORBA 3.0. CACM. 1998; 41(10):44-52.

177. Watson, Paul and Parastatidis, Savas. [Technical Report]. NIP: A Parallel Object-Oriented Computational Model. University of Newcastle Upon Tyne; 1998 Nov; CS-TR-658.

178. Watson, Paul and Parastatidis, Savas. [Technial Report]. An Optimised Lazy Task Creation Technique for Iterative and Recursive Computations. University of Newcastle Upon Tyne; 1999.

179. Webgain. Java Compiler Compiler (JavaCC) - The Java Parser Generator [Web Page]. Available at: http://www.webgain.com/products/java_cc/.

180. Wegner, Peter. Dimensions of Object-Based Language Design. ACM SIGPLAN Notices. 1987 Dec; 22(12):168-182.

181. Welsh, Matt and Culler, David. Jaguar: Enabling Efficient Communication and I/O from Java. Concurrency: Practice and Experience. 1999 Dec.

182. White, Stephen. VNet website: http://www.csclub.uwaterloo.ca/u/sfwhite/vnet/ [Web Page]. Accessed 2000 Apr. Available at: http://www.csclub.uwaterloo.ca/u/sfwhite/vnet/ http://ariadne.iz.net/~jeffs/vnet/.

183. Wilkinson, Barry and Allen, Michael. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice-Hall, Inc.; 1999.

184. Yelick, Kathy; Semenzato, Luigi; Pike, Geoff, and Miyamoto, Carleton. Titanium: A High-Performance Java Dialect. ACM Java Grande Conference; Stanford University, Palo Alto, CA. 1998.

185. Yokote, Yashuhiko and Tokoro, Mario. Experience and Evolution of Concurrent Smalltalk. OOPSLA '87; Orlando, FL. ACM Press; 1987: pp. 406-415.

186. Yonezawa, Akinori. ABCL: An Object-Oriented Concurrent System -- Theory, Language, Programming, Implementation and Application. MIT Press; 1990.

187. Yonezawa, Akinori; Shibayama, Etsuya; Takada, Toshihiro, and Honda, Yasuaki. Modelling and Progamming in an Object-Oriented Concurrent Language: ABCL/1. Yonezawa, Akinori and Tokoro, Mario. Object-Oriented Concurrent Programming. MIT Press; 1987.

188. Yonezawa, Akinori and Tokoro, Mario. Object-Oriented Concurrent Programming. MIT Press; 1987.

189. Yu, Weimin and Cox, Alan. Java/DSM: A Platform for Heterogeneous Computing. Concurrency: Practice and Experience. 1997; 9(11):1213-1224.

190. Yuen, C. K. and Feng, M. D. Adding Objects to Parallel Languages. Software Concepts & Tools. 1995; 16:95-105.

191. Zenger, Matthias. Transparente Objektverteilung in Java: University of Karlsruhe, IKA; 1997 Feb.

192. Zyda, Michael J.; Pratt, David R.; Monahan, James G., and Wilson, Kalin P. NPSNET: constructing a 3D virtual world. Proceedings of the 1992 Symposium on Interactive 3D Graphics; 1992: 147-156.