Newcastle University

Scalable and Responsive Real Time Event Processing Using Cloud Computing

by

Visalakshmi Suresh

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Science, Agriculture and Engineering

School of Computing Science

May 2017

Newcastle University

ABSTRACT

FACULTY OF SCIENCE, AGRICULTURE AND ENGINEERING
SCHOOL OF COMPUTING SCIENCE

Doctor of Philosophy

by Visalakshmi Suresh

Cloud computing provides the potential for scalability and adaptability in a cost effective manner. However, when it comes to achieving scalability for real time applications response time cannot be high. Many applications require good performance and low response time, which need to be matched with the dynamic resource allocation. The real time processing requirements can also be characterized by unpredictable rates of incoming data streams and dynamic outbursts of data. This raises the issue of processing the data streams across multiple cloud computing nodes. This research analyzes possible methodologies to process the real time data in which applications can be structured as multiple event processing networks and be partitioned over the set of available cloud nodes. The approach is based on queuing theory principles to encompass the cloud computing. The transformation of the raw data into useful outputs occurs in various stages of processing networks which are distributed across the multiple computing nodes in a cloud. A set of valid options is created to understand the response time requirements for each application. Under a given valid set of conditions to meet the response time criteria, multiple instances of event processing networks are distributed in the cloud nodes. A generic methodology to scale-up and scale-down the event processing networks in accordance to the response time criteria is defined. The real time applications that support sophisticated decision support mechanisms need to comply with response time criteria consisting of interdependent data flow paradigms making it harder to improve the performance. Consideration is given for ways to reduce the latency,improve response time and throughput of the real time applications by distributing the event processing networks in multiple computing nodes.

Contents

Nomenclature v

Acknowledgements vi

1 Introduction 1

2 Background 9

2.1 Cloud Computing Overview . . . 10
2.2 Event Driven Research . . . 12
2.2.1 Principles of Event Processing . . . 13
2.2.2 Simple Event Processing . . . 13
2.2.3 Complex Event Processing (CEP) . . . 14
2.2.4 Intelligent Event Processing (IEP) . . . 14
2.2.5 Volume of events: . . . 15
2.2.6 Volume of event processing: . . . 16
2.2.7 Number of event producers: . . . 16
2.2.8 Number of event consumers . . . 17
2.2.9 Context partitioning: . . . 18
2.2.10 Context state: . . . 19
2.3 Bin Packing . . . 20
2.4 Event Driven Systems . . . 21
2.4.1 Aurora and Borealis . . . 21
2.4.2 Cayuga . . . 23
2.4.3 Streams . . . 25
2.4.4 NiagraCQ . . . 26
2.4.5 TelegraphCQ . . . 26
2.4.6 DISPEL . . . 27
2.4.7 Others . . . 27
2.4.8 Summary . . . 28
2.5 Parallelisation of Event Processing . . . 28
2.5.1 Summary . . . 32
2.6 Scalability in Cloud Computing . . . 33
2.6.1 Resource Utilisation . . . 33
2.6.2 Summary . . . 37

3 Real time streaming application **39**
3.1 Introduction . . . 39
3.2 Ambient Kitchen Overview . . . 40
3.3 Benchmarking Application . . . 41
3.4 Autonomous Event Processing Network (EPN) Design . . . 42
3.4.1 Event Producers . . . 44
3.4.2 Processing Elements: . . . 44
3.4.3 Event Consumers . . . 47
3.4.4 Dashboards . . . 47
3.4.5 Activity Recognition Summary: . . . 47
3.5 Contributions . . . 54

4 Architecture **57**
4.1 Introduction . . . 57
4.2 Architecture . . . 58
4.3 System Description . . . 64
4.4 Problem Specification . . . 65
4.5 Event Processing Architecture . . . 66
4.5.1 Configuration Scheduler . . . 66
4.5.2 Inter-EPN Parallelism . . . 68
4.6 Evaluation . . . 69

5 Algorithms for Responsiveness **71**
5.1 Introduction . . . 71
5.2 Analytical Estimation of EPN Response Times and CPU Usage . . . 71
5.2.1 Modelling and Calibrating a Single EPN . . . 72
5.2.2 Modelling Multiple EPNs in a Single Host . . . 73
5.3 Optimal Configuration Selection . . . 75
5.4 InterEPN Parallelism . . . 77

6 Evaluation 1: Response Time Estimation **80**
6.1 Introduction . . . 80
6.1.1 Response Time Measurement . . . 81
6.1.2 Calibration and prediction of EPN Response Time . . . 87
6.2 Experiment Setup . . . 90
6.2.1 Validation of Response Time Prediction by Measurement . . . 93
6.3 Conclusion . . . 94

7 Evaluation 2: Reconfiguration **95**
7.1 Introduction . . . 95
7.2 Design of Configuration Scheduler . . . 96
7.2.1 Grouping of EPN: . . . 97
7.2.2 Optimum response time: . . . 97
7.2.3 Balanced resource utilization and response time: . . . 98

7.2.4 Optimum resource utilization: 98
7.3 Implementation of Configuration Scheduler 98
7.3.1 Resource Scheduler: 101
7.3.2 EPN Tracker: 102
7.3.3 Host configurator: 102
7.3.3.1 Hiring and registration of computing nodes: 103
7.3.3.2 EPN instances: 103
7.3.3.3 Local autonomy: 103
7.3.3.4 Parameters for the computing node: 104
7.3.4 Elastic Management (Shrink/Scale): 104
7.4 Evaluation of Configuration Scheduler and Reconfiguration Algorithm . 105
7.5 Validation of Multiple EPN Reconfiguration 109
7.6 Conclusion 111

8 Evaluation 3: Inter-EPN Parallelization 112
8.1 Introduction 112
8.2 Parallelization of EPN 114
8.3 Implementation of Inter-EPN Parallelization 117
8.4 Use case analysis 122
8.4.1 Consecutive Events: 122
8.4.2 Time Limited Search 127
8.5 Evaluation 131
8.5.1 Inter-EPN Parallelization of Consecutive Events: 131
8.5.2 Inter-EPN Parallelization of Time Limited Search 135
8.5.3 Inter-EPN Parallelization Experiments Outcome 137
8.5.4 Observations 139
8.6 Conclusion 140

9 Challenges and Future Work 141
9.1 Introduction 141
9.2 Virtualisation 141
9.2.1 Scalability 142
9.2.1.1 Cost of cloud computing utilization 142
9.2.1.2 Policy driven scheduling 143
9.3 Resource Utilisation 144
9.4 High Availability 145
9.5 Conclusion 145

Bibliography 147

Nomenclature

RT_i	The response time
T_i	The target response time
AR_i	The total arrival rate
W_i	The estimated response time
U	Total load utilization of the computing nodes
λ	Poisson Arrival Rate
δ_i^m	Relative deviation of RT_i and m indicates that δ_i^m is measurable
$M_{2,i}$	second moment of the processing times
DAG	Directed Acyclic Graphs
EPN	Event Processing Networks
ρ_i	Processing Load imposed by EPN_i on the host node
b_i	Average processing time

Acknowledgements

Firstly, I would like to express my heartfelt gratitude to the Dr Paul Ezhilchelvan for spending his valuble time in various research discussions to shape this work. For the past four years, he has contributed immensely in shaping the research objectives, streamlined my focus towards a particular aspect of the research challenge and been patient in providing feedback on my various versions of the thesis. I am extremely thankful for this collaboration process of research, which has provided me an insight into the incremental stages of research progression, technical writing and publications.

Secondly, I would like to thank Professor Paul Watson for guiding me to register for this PhD whilst providing me employment in various EPSRC funded research projects such as MESSAGE, SwitchEV, SiDE and BeSiDE. For the past seven years, these projects exposed me to various research challenges and let me identify an area of interest in real time event driven architecture. Paul's vision and directions helped to keep my focus and got me out of many deviations from the work related commitments.

I am extremely thankful to both my supervisors, without whom the thought of undertaking doctoral research would have been impossible.

Above all, my thanks to my husband and kids for giving me the time and space to pursue my research.

Dedicated to my parents

Chapter 1

Introduction

Cloud computing attempts to realize the vision of utility computing, through the provisioning of virtualized hardware, software platforms and software applications as services over the Internet. It comes with many options such as on-premises private clouds, off-premises public cloud, hybrid cloud and the mix-and-match approach of multi-cloud options. The crowding landscape of the cloud computing offerings only exacerbates the new offerings, which constantly enter the market place. The real time information management using cloud computing attempts to utilise the cloud and event processing to process complex real time workloads. In a nutshell, this research is an extensible framework providing ability to process real time streaming events to Infrastructure As A Service(IaaS) cloud systems, predict the application performance based on the arrival rate of streaming events and subsequently perform the analysis on multiple instances of virtual machines hired from the cloud. To implement the real time information management in cloud computing, few key challenges in real time application performance and scalability must be recognised. Listed are the application performance related challenges thesis focused in this thesis.

1. Prediction of response time requirements for the application.

2. Placement of the applications across multiple instances in the cloud.

3. Splitting of application workload across the instances hired in the cloud.

Event processing is a computing performed on continuous real time streaming events rather than finite stored data sets. Each event stream has a fixed schema with known finite set of attributes. The instance of an event stream at any point of time is a

bag of tuples which may occur in a fixed or varying interleaved sequence generated by a single or multiple data sources. In the event processing, the incoming stream or events are not bounded by any particular limits, where the answer to the query at any point of time is a function of the input event streams received over a period of time. Continuous queries are used to evaluate the event streams. The query evaluation environment has an access for local memory and CPU to process, store and disseminate the events arriving in to the system. Each query registered within the system is computed in a bounded memory and CPU defined by the characteristics of the hardware. The arrival rate of the events and the underlying hardware determines the response time of the queries. When the arrival rates of the events are higher, insufficient or static hardware allocation leads to a decrease in the response time. In this research, algorithms are formulated to predict response time for the event processing under varying arrival rate of events by hiring or releasing virtual machines using cloud computing. Firstly, algorithms are developed to predict the response time of the real time, continuous streaming events using queuing theory. Secondly, algorithms are developed to meet the optimum placement of the applications by addressing additional memory requirements of real time continuous streaming events through hiring and releasing optimum number of virtual machines using cloud computing. Thirdly, when the stated response time cannot be met within a single virtual machine, algorithms for distributed event processing are developed to split, process and merge the events, maintaining the correctness of the results. The research executes the development, implementation and the testing of the algorithms for the response centric scalability of real time event processing by hiring virtual machines from cloud computing.

Event processing approach is ideally suited in scenarios intended to deliver situational awareness and responsiveness where high volume of streaming events represent accumulating, incremental state change. The complexity of the event processing are measured based on the degree of changes within the event flow, number of event sources, number of consumers, state and context management. Event volumes and timeliness are two factors that most typically influence the selection of the event processing approach. In this research the event processing is encapsulated within Event Processing Network (EPN). EPNs are designed as an acyclic graph. The processing elements within the EPN make up the node of the graph. Few examples of the processing elements are continuous queries, event producers, event consumers, event processing agents and channels. The EPN abstracts the operations in the events that include processing, creating, transforming or discarding the events based on the specification detailed by the application consuming the events. Besides event processing, most workflow languages support the processing of event constructs such

as sequence, partitioning sliding window, iteration, splits (AND or OR) and joins . Few continuous data flow models, Floe [Kumbhare et al. (2013)] implements adaptive resource allocation to effectively use elastic cloud resources to meet varying arrival rates. Floe implemented pattern matching criteria on iterative constructs such as for-loops. However, the interpretation and support of pattern matching constructs on complex real time event processing requirements need further extension on the workflow languages [Van Der Aalst et al. (2003)]. The workflow architectures must integrate meta-models such as workflow languages such as BPMN [Zimmermann and Doehring (2012)] and rule definitions in a minimal-invasive way with little changes to the original meta-models [Dhring et al. (2010)]. To accomplish pattern recognition on streaming events, standard rules and protocols for ECA (event, condition and action) based on continuous queries for real time streaming events need to be implemented. There is no clear holistic definition for a loosely coupled architecture to achieve the tasks accomplished by a complex event processing system (CEP) [Luckham (2001)]. For example, workflow management system can be extended step-by-step with rule engine or an exclusive CEP engine can be embedded within the workflow modules. The advancements in proprietary languages addressing complex event processing (CEP) and continuous query language are a motivating factor to develop the EPNs using CEP engines [EsperTech][Microsoft][Oracle].

The responsiveness and the complexity in identifying the patterns in streaming events are used as main criteria to develop EPNs. EPNs are designed to support low latency and high throughput thereby improving the responsiveness of the application built on the continuous streaming events. In this work, complex event processing is used as an enabling technology where multiple real time stream of events are analysed to identify patterns (aggregation, correlation or filtering). Real time stream of events are continuous in nature, possibly very rapid and time varying. Developing event processing component involves many novel and challenging problems, since the queries tend to be continuous (long-running) rather than one-time. Distribution of event streams and arrival characteristics may be unpredictable and system conditions (eg.,available memory,CPU,etc.,) fluctuate over time. When the streaming events are processed in a specific server or a desktop, the query evaluation environment has access to a limited or fixed CPU/local memory for storing or processing the events. However, the memory and CPU used to process of events influence the responsiveness or response time of the application in terms of latency and throughput. The initial goal is to characterise the worst-case memory or CPU requirements under fluctuating arrival of events and add additional memory or processing capabilities through hiring of virtual machines from cloud computing. However the research progressed in developing

generic algorithms to characterise and predict the CPU requirement for the EPNs through queuing theory. Following assumptions were initially made to progress the research

1. Queuing delays or the data transfer time between the EPNs are dependent on the networks or the environment for deployment such as GPU, HPCs, private cloud, public cloud, etc., To improve the accuracy in the prediction of the EPN response time, both the queuing delays and the data transfer time between the EPNs were not considered.

2. A deterministic CPU allocation is considered to model the response time prediction

3. The hiring time of a virtual machine is not considered

4. The cost or economic factors to hire the virtual machines are not considered. For example reserved instance could be cheaper than on-spot instances. These factors were not considered in the model.

5. The events received by the EPN from the source are not modified during the transit.

6. When the single EPN does not meet the target response time (contention between EPNs), operators are shared between the EPNs based on a split, process and merge paridgm to maximise resource utilisation.

As an initial step, characterising memory requirements for queries over continuous streams of relational tuples were developed through a series of examples from a research project called ambient kitchen [Olivier et al. (2009)] in Newcastle University. The ambient kitchen is a laboratory replication of the real kitchen embedded with RFID tags, readers, object mounted accelerometers, under-floor pressure sensing using combination of wired and wireless networks. Range of research activities were undertaken in the lab to support people with dementia, situated services associated with food planning, food preparation and cooking. The events emerge from the sensor infrastructure within the kitchen utensils such as knives, pots, lids, frying pan, spatula, sieve, peeler, ladle, whisk and copping board. Continuous stream of events from pervasive sensors arriving from ambient kitchen are processed using EPNs deployed in virtual machines hired using private and public cloud computing services.

Given the nature of the events arriving from the ambient kitchen, the arrival rate of the events is observed to be arbitrary. The design of the response time prediction

for EPNs are dependent on the events, that cannot be processed immediately, whose processing or service time of events is interrupted or has to wait in one or more queues. A scheduling policy is required to control the order in which the waiting events are selected for service. These are the 'queuing systems' and their study is the called 'queuing theorem'. The queuing theorem consists of three parts. The first part describes the nature of the arrival process, the second part describes the distribution of the service time and the third describes the number of servers or virtual machines required to process the events. Letters M, D and G are used as the abbreviations for 'Memoryless', 'Deterministic' or 'General' respectively. For example, in a $M/D/5$ system, the events are memoryless (i.e., poisson), processing time is deterministic or constant and there are 5 servers or virtual machines involved in the system to process the events. In $M/G/1$, the processing or service of events is memoryless (i.e, exponentially distributed) with generic distribution and a single server or virtual machine is used to process the events. The models examined and utilised in this research are $M/G/1$. For analytical tractability, poisson arrival rate is used to predict the response time of the EPN. If the arrival rates of events deviate from the poisson distribution, the results will be of closer-approximation. When there is a poisson arrival, most of the assumption fit in the $M/G/1$ queuing theory model and the accuracy of the response time prediction in EPNs are analysed. Degree of the approximation will be verified using experiments in chapter .

The parameters which influence the average response time of the EPN of the $M/G/1$ system are the arrival rate , the average service time, b, and the second order moment,M2 (or the squared coefficient of variation, c2), of the service time. When the offered load approaches 1, both the average delays and the queue length grow without bound. In a heavily loaded virtual machine or server, a small increase in traffic can have a huge impact in the response time of the EPNs. The shape of the processing time distribution function affects the performance only through the second moment. For fixed values of λ and b, the best performance is achieved when $c2 = 0$, i.e. when there is no variability in the service or the processing time of the EPNs. All EPNs are calibrated before deployment and a deterministic service or processing time is known prior to the placement of EPNs in the virtual machine.

When the virtual machine receiving the $M/G/1$ event queue goes through alternating periods of being idle and busy, the load in the servers or virtual machines processing the EPNs undergo an idle or busy state. An idle period starts with the departure of events leaving an empty queue and ends with the arrival of a new set of events. Due to the memoryless property of the exponential distribution, the average length

of an idle period is $1/\lambda$. The busy period in the virtual machine starts with the arrival of the events, which finds an empty queue and ends with the departure of the next set events leaving the empty queue. Due to the dynamic nature of the event arrival, few virtual machines are likely to be underutilised and few tend to be over utilised. An intelligent placement of EPNs is essential to hire minimum number of virtual machines. Bin packing algorithms are used to place the EPNs in minimum number of virtual machines whilst maintaining the optimum response time. Virtual machines are hired from the cloud computing services based on the event arrival rate. The scalability of the algorithms is tested using the streaming events arising from the ambient kitchen. Initially one virtual machine is hired and the EPNs are deployed within the machine. As the event flow increases, the response time of the EPNs decreases and hence further count of virtual machines are estimated using the algorithm in section . Multiple EPNs can be hosted on a single virtual machine. Based on the arrival rate and the response time of the EPNs, the count of EPNs hosted on the virtual machine is varied dynamically. Lightly loaded virtual machines are called acceptors where additional EPNs can be hosted within the VMs without compromising on the response time. Heavily loaded virtual machines are called donors where the EPNs need to be moved in to another VM in order to improve the response time. The list of the acceptors or donors are maintained within the system and fed to the EPN reconfiguration and the placement algorithms. Frequent EPN movement between the virtual machines or computing nodes are likely to generate oscillatory behaviour. The dynamic hiring of new virtual machines from the cloud computing took an approximate of 1 to 5 minutes from few well known commercial cloud service providers such as Amazon EC2. Keeping the upper limit of virtual machine hiring time i.e., five minutes, the movement of EPNs were triggered by the reconfiguration algorithm. Fault tolerance or check pointing of VMs are reserved for the future work.

The EPNs are tested for scalability and responsiveness. The EPNs are built using expressive open source event processing language called Esper, which allows continuous queries, aggregation and joins delivering high speed processing and identification of most meaningful events, analysing the impacts on the application and undertake subsequent actions in the real time. The events emerging from the application used in this research, i.e., ambient kitchen are processed by the EPNs regardless of whether incoming events are real-time or historical in nature. The pre-processed patterns of various activities such as cutting, chopping, coring, slicing, stirring, etc., in the ambient kitchen are stored in the database. The real time activity recognition in the kitchen is accomplished by correlating the incoming real time events and pre-defined patterns of activities. Instances of Esper event processing engines are prebuilt and

stored as reserved instance in the cloud computing. The prebuilt instances of the virtual machines are scaled in multiple machines based on the arrival rate and the response time requirements. Streaming events emerge from sensors embedded in various utensils, equipments and utilities. The characteristics of each device or event source are pre-defined using a schema. Event windows are created by the EPNs to manage the fine-grained rules on event processing. The EPNs have a set of rules on how long to retain set of relevant events or under what conditions event need to be discarded or stored in the database. The event windows operate on the level of individual queries and sub queries as illustrated in section.. The EPNs developed in this research are focused targeted towards one particular application requirement called activity recognition. Ambient kitchen is a very broad area of research. In this thesis, the focus is made on building the EPNs based on one application called activity recognition targeted towards demonstrating the scalability of EPN placement in multiple virtual machines hired from cloud computing services with an aim to achieve the specific responsive time. The hiring of virtual machines from cloud computing has various cost implications based on the policy of hire such as on-demand instances, reserved instance, spot instances etc., The algorithms developed in this research mainly focus on the responsiveness of EPNs and to identify the appropriate virtual machine to place the EPNs without compromising the response time. The optimum cost computation policies to hire the virtual machines are reserved for future work.

To summarize, the characteristics of the streaming event and its processing requirements are analysed to address the response time prediction of the event processing applications. An architecture is developed along with the functional components required for the event processing to meet the scalability of the system using utility computing. After a broad literature review to provide a bird's eye view of the research and development efforts in the areas of event processing, cloud computing and scalability, this work focuses on the response centric scalability of real time event processing systems. Techniques and approaches required for guaranteeing response time (latency), placement of EPNs, parallelisation of EPNs are presented in detail. This thesis is structured on the topics of

1. Event processing network design
2. Modeling and prediction of the EPN response time using queuing theory
3. Dynamic reconfiguration and placement of event processing networks using cloud computing (Public and Private cloud)
4. Parallelization of event processing network using Inter-EPN processing

The thesis describes the scalable, response centric design and implementation of event processing systems using exemplars of data emerging from the ambient kitchen. Hardware resource from public and private cloud computing is used to demonstrate the elastic and scalable architecture for real time event processing.

Journal Publication from this research :

Scalable and Responsive Event Processing in Cloud, Suresh V, Ezhilchelvan P, Watson P, Philos Trans A Math Phys Eng Science, 2012, December 10, 371(1983):20120095.doi: 10.1098/rsta.2012.0095. Print 2013 Jan 28

Chapter 2

Background

Real time applications involve streaming data characterized by high-volume event streams, which require timely sense and respond infrastructure. Applications such as web clients, or volatile markets such as finance or betting, have an emerging interest in continuous monitoring. Commercial social media applications and online gaming are pioneers in creating analytical services for the real time analysis. Emerging applications in RFID, pervasive computing and others adopt large scale real time event tracking and monitoring. The wide deployment of real time monitoring systems in surveillance, health care, environmental monitoring, instrumented vehicles and others require filtering, correlation and aggregation to aid decision support systems. The number of data sources and the rates at which they generate events can vary widely, driven purely by the external processes. The common patterns and architecture driving these platforms iterate multiple features based on the arrival of data.

The data emerging from streaming events motivates building processes which respond to the continually changing inputs.Celerity, connectedness and complexity of interacting event sources result in the increasing use of event-driven applications. The need for the systems to analyze and react to the continuously varying incoming data streams leads to the emergence of the event-driven architecture moving away from the traditional database oriented approach. This chapter starts with a review of the literature on event processing research, looking in more detail at the continuous queries, distribution and scheduling of complex event processing, along with the scalability aspects of cloud computing.

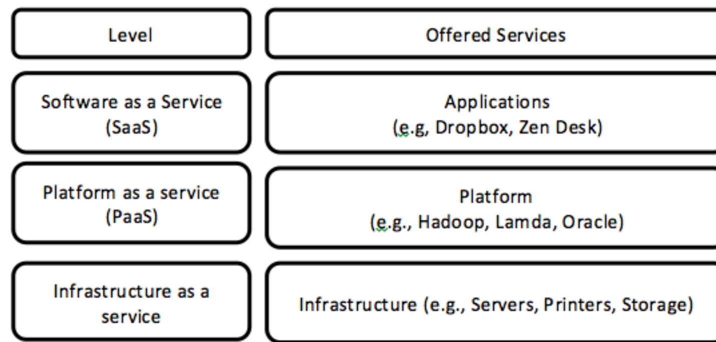


Figure 2.1: Cloud Abstraction Layers

2.1 Cloud Computing Overview

The cloud computing encapsulates several layers of computing provisioning. It includes the hardware resources located at the data-centres of cloud providers, the operating system software on top of that hardware, and lastly the applications that are delivered as services over the Internet. Cloud providers purchase hardware in large quantities, which is significantly more economical [Greenberg et al. (2008)]. They can then amortize the cost of owning and operating a large capacity infrastructure by time-multiplexing their resources between many clients [Armbrust et al. (2010)]. Moreover, large scale systems require deep automation, which results in cost reductions due to need for smaller operational staff. In a well-run enterprise, a typical ratio of administrators to servers is 1:100, while in a cloud data-center the ratio is at least 1:1000 [Greenberg et al. (2008)]. Operating at this scale allows cloud providers to offer services to clients with a lower cost than what an in-house computing facility would achieve.

Clouds provide services at three different levels of abstraction: At the highest abstraction level, Software as a Service (SaaS) delivers specialized software to the consumers over the Internet. SaaS typically involves a usage-based pricing scheme, in which the cost increases in relation to the number of users and the used application features.

Platform as a Service (PaaS) is located at a lower abstraction level. PaaS pertains to the provisioning of an integrated environment which can be used for the development, testing and deployment of applications. The PaaS users are not occupied with deploying and managing the underlying hardware and software.

At the other end of the abstraction level spectrum is Infrastructure as a Service (IaaS). IaaS refers to the on-demand provisioning of virtualized resources, i.e., computation,

storage and networking. Users lease VM instances, which encapsulate a provider-specified amount of resources (providers typically offer several types of VM instances), and can run a user-specified operating system enriched with required applications and libraries. Clients fully configure and control their instances as root via ssh.

Generic guidance from Cloud Service providers is available to select the ideal instance type, sizing and storage. However, the individual applications need to predict their run time resource requirements based on the varying incoming workloads. Understanding the real-time application demand, delivering consistent response time on each of the hired instance and predicting the response time to provision the resources are few of the key challenges associated in the real-time information management across the cloud. Underestimating the application requirements will lead to performance issues and over-provisioning will lead to the cost related issues. Once the instances are selected from the Cloud computing providers, performance prediction is required to ensure the availability of sufficient resources to the applications.

Many cloud computing providers deliver well architected framework ensuring the expectation on the instance and application performance expectations. Listed below are few features delivered by commercial off the shelf products in the cloud computing.

Cloud Watch: Monitors the CPU, memory, network, load, storage and related parameters specific to the hired instances. Periodic Review: Many cloud providers come with monitoring dashboards on real time application performance and response time.

Alarm-based notifications: Automated alerts from the monitored system metrics when they are outside the safe bounds.

Trigger-based actions: Alarms cause automated actions to remediate or escalate issues.

Google Cloud Platform, AWS, Microsoft Azure are few of the prominent cloud services offering the performance best practices in terms of monitoring metrics, setting thresholds and responding to the alerts. Public clouds promise many such performance related tools however they do not guarantee or predict an application performance when subjected to the varying workloads from real time streaming events.

In Infrastructure as a Service (IaaS) the virtual machines are classified as reserved or on-demand instances. For a predictable workload, the reserved instances are appropriate. For an unpredictable and varying workload, the on-demand instances having flexible hourly payments without any long-term commitments are more suitable. In this work, reserved and on-demand instances are provisioned to meet the varying

workloads for compute intensive applications to building scalable and effective framework to process the incoming workloads. The figure ?? below represents the overview of IaaS, PaaS and SaaS

2.2 Event Driven Research

Traditional database management systems (DBMS) generally act as repositories to store the data. The architecture assumes a pull-based model for the access of information and analysis. The data is stored in the system and intelligence is extracted by submitting the requests or queries. DBMS acts as a passive repository storing a large collection of data elements and humans initiate queries and transactions on this repository known as a human-active, DBMS-passive (HADP) model. Monitoring continuous event streams involves continuous execution of queries rather than human initiated transaction. This is called a DBMS-active, human-passive (DAHP) model. The emerging applications with real time event streams require complex information processing and performance requirements to model the structural and behavioral aspects of an application. Consider a situation in a roadside traffic monitoring system, which requires identification of the traffic congestion or hotspot occurring in a particular section of the road network. In some circumstances (e.g. peak hour, sporting event) the flow of data from sensing devices such as GPS navigation systems, mobile phones, loop counters, traffic signals and related sources significantly increases. In a database management system, the monitoring functionality can be improved only through additional servers and processing capacity. However, the semantics of the processing need to be distributed and replicated across heterogeneous data sources executed through different application programs. Given the complex semantics involved in the real time event based applications, the difficulty stems from ascertaining appropriate polling frequency to access the data from the database. If the frequency of the polling increases, the maintenance and cost of the DBMS increases. If the DBMS infrastructure cost is reduced, the response time for the information processing is compromised. Event based research progressed in the development of Complex Event Processing (CEP), which allow the easy capture of sequences of events involving complex ordering relationships. In CEP, the events are processed directly in-network, as streams flow from source to sinks. Event based systems focus on event notification, with a special attention to their ordering relationships to capture complex event patterns. Moreover, they target the communication aspects involved in the event processing, such as ability to adapt to different network topologies as well

as to various processing devices, together with the ability of processing information directly in-network.

2.2.1 Principles of Event Processing

The basic building block in the architecture of real time streaming data is events. Events are described as any action that occurs in a given instance of time or contemplated as happening in the current time. This could be a landing of an airplane or state in a finite state machine. The event object represents an activity and each event records multiple attributes of the activity. In many systems the events are immutable. Creating a new event object rather than altering the events leads to the transformation of events. Deletion of an event in the architecture would entail the system making meaningful decisions. The software artifact to process the events is called Event Processing Agent (EPA) [Etzion and Niblett (2010)]. EPA receives a single event or stream or collection of events based on the agent type and capabilities. The agent type would indicate whether the EPA acts as an event source or event sink or both. Channels are used to connect the event processing agents. A set of event processing agents and channels form the Event processing Networks (EPN). The EPA and channels can be dynamic. The dynamic behavior is controlled by the patterns occurring in EPN. EPN can be designed as an acyclic graph or feedback loops. The figure illustrates various components in the EPN. The processing elements in the EPN are event producers, event consumers, agents and channels. The processing elements make up the node of the graph. The output elements are referred to as edges in the EPN. The EPN is an abstraction of the operations on events, which includes reading, creating, transforming or discarding events based on the specification detailed in an application.

2.2.2 Simple Event Processing

Simple event is defined in the atomic level where processing of the events is based on the occurrence of a single event[Luckham and Schulte (2008)]. It is viewed as summarising, representing or denoting a set of other events.Filtering and routing are the two types of processing undertaken on these events. Filtering decides whether the event progresses to the next stage within the EPN. In real world scenarios, event processing is used to detect or report situations that initiate actions automated or triggered by human interventions. The relationship between events is categorized as

deterministic or approximate, based on the mapping between a situation in the real world and its representation in the event processing system.

2.2.3 Complex Event Processing (CEP)

CEP creates actionable, situational knowledge from events arising from distributed data sources, databases, application in real time or near real time. This extracts higher-level knowledge from situational information abstracted from processing low-level event feeds [Etzion and Niblett (2010)]. In order to create a processing environment, the processing network should be capable of one-to-one, one-to-many, many-to-one and many-to-many communications.

2.2.4 Intelligent Event Processing (IEP)

The IEP enables complex event processing (CEP) and predictive analysis. Designing event processing networks for intelligent event processing utilizes data mining and machine learning techniques for discovering new insights and pattern detection. Common types of predictive models include decision trees, neural network, regressions, clustering and association models. In a few instances, an event processing network or event processing engine invokes predictive models such as a scoring system [Muthusamy et al. (2010)]. Intelligent event processing is commonly supported through logical proofs, which helps the transitions of the events.

The three types of event processing discussed in the previous sections lead to the main goal of timeliness in the deliverables leading to the emphasis on performance. The characteristics of events such as arrival rates, logical structure, system defined attributes, application defined attributes vary based on the event source and volume of events. The design of the event processing systems needs to deal with optimization and performance metrics relating to the characteristics of the event source and the incoming arrival rate of the events.

In an event based systems context, content and the inter-relations among events play a vital role in the optimisation and performance metrics. The interaction between the event source and sinks, distributed in wide geographic areas adds to the latency and compromises the responsiveness of the system. Varied number of users such as high-level applications, end users, mobile applications push the requirements on the expressiveness, scalability and flexibility of the event processing systems. The general

model on Data Stream Management Systems popularly known as DSMS [Arasu et al. (2004)] makes architectural choices with a set of standing queries focusing on the data flow and data transformations. [Li et al. (2007)] allow the capture of sequences in events involving complex ordering.

This research considers aspects of the effective event processing including the complex relationship among the data with a focus on response centric event processing using resources from cloud computing. To pursue the goal, the quality of service is analyzed in terms of scalability and responsiveness. Scalability is defined as the ability of the system to adapt fluctuations in the intensity of use, volume or demand. Several dimensions of scalability relevant to the event producers, event consumers, contexts, complexity and environmental conditions are listed below:

2.2.5 Volume of events:

The large volume of events and bursty inputs with temporary overloading are increasingly common in streaming events. Traditional architecture uses exclusive ownership of hardware or physical resources that enable the optimization of event flow for the specific hard disk by issuing sequential read and writes. However to enable event processing, the capital expenditure on procurement of resources limits the capability to handle the high volume of the events. The advancements in the virtualized environments such as cloud computing increase the agility of handling the high volume streaming data through elastic, on-demand storage and processing capabilities. The hardware procurement cost for short term use to handle the bursts of event flow is considerably reduced with the use of cloud computing.

The natural solution to handle the high volume of events is to avoid the overheads associated with resource contention. During the overload of the events, systems were designed to shed load in order to maintain low latency results [Tatbul et al. (2003)]. Fraction of the tuples are dropped in a random fashion or based on the importance of the content. Overloading or increase in volume of the events is unforeseen and immediate attention is vital such as adapting the system capacity by adding new computing resources or distributing the computation to multiple nodes [Cherniack et al. (2003a)]. Application specific quality of service information was used to make load-shedding decisions. For example, in a banking scenario, one event does not identify a fraudulent login. A pattern of events over a period of time is important to detect fraudulent actions. The load-shedding scenario has the risk of deleting the event history. This research achieves scalability of event processing without load

shedding under varying incoming arrival rates. Event processing is implemented using the hiring of cloud computing resources.

2.2.6 Volume of event processing:

Event processing networks use the event-condition-action (ECA) paradigm from the situational point of view to detect the complex or composite events. This research analyzes the scalability of event processing through the development of inter-EPN parallelisation described in Chapter 7 to address the issues listed below:

1. Consistency in the outcome of the processed events while scaling the event processing networks.
2. Models to synthesize the aggregation of results from the multiple event processing networks.
3. Scaling of the event processing networks to detect temporal, spatial, correlated or complex events.

2.2.7 Number of event producers:

The quantity of events from various event sources or producers consistently increases over time. For example, data emerging from RFID or pervasive sensing exceed several millions of events per day. Continuous events need to be processed iteratively against the incoming events to produce continuous results. The event based systems process the events and produce long running results and may also access stored relations across the heterogeneous event producers. The event-based systems use continuous query language (CQL) which consists of relational operators such as select, project, join and aggregation operators. The operators process the incoming events as they arrive and cannot assume the events streams to be finite. For an operator or a query to generate continuous output on the events generated from multiple producers, a notion called *window* is commonly used to define a finite portion of the events. The database management systems use blocking operators where output from one operator is sent to the next operator. The event based systems use non-blocking operator imposed by a *window*. A *window* can be defined as a snapshot of finite portion of events from a single event producer or multiple event producers at any instance of time. *Window* can be defined as time based (5 minutes or 5 seconds) or tuple based (count of event

tuples) according to the computation. *Windows* define the start and end boundaries to process the events. The advancement of processing from one window to another window can be defined using overlapping or disjoint boundaries of time or tuple count.

Windowing based computation through unblocking operators forms the basis for handling and scaling events from multiple event producers. In continuous event processing, the count of event producers cannot be restricted or defined during a system initiation. Processing of events across continuous events is conceptualized as the data flow diagram moving way from pipelined or iteration-based pull approaches used in the database management systems. The abstract semantics that converts event streams to relation, relation to streams and relations to relations have been defined in literature [Arasu et al. (2004)]. With the increase in the volume of event producers, many distributed event processing architectures have emerged, such as placement of replicated tasks [Lakshmanan et al. (2010)].

2.2.8 Number of event consumers

Event based systems are decoupled in time, space and synchronization to support scalability [Eugster et al.]. In a general design, systems assume the existence of mutual agreement between the producers and the consumers of the event semantics, as they add explicit dependencies between participants. The tightly coupled producers and consumers need exact matching of subscriptions to address the semantic heterogeneity in the events. Hasan et al. (2012) proposed an approximate semantic event matching paradigm focusing on event enrichment. Filtering or matching of events against the large number of subscriptions and arrival rates requires complex computations. The event based systems use the unblocking operator approach where event consumers can subscribe from multiple processed event queues or heterogeneous event producers. The schematics of the event subscription is conceptualised in data flow diagrams defined by the high level event driven architecture. The decoupling of the event subscription provides scalability options for event producers and consumers. In the spatial decoupling the interacting consumers need not know each other. The event producers are unware of the count of event consumers or the interaction between the event consumers. In the time based decoupling, the event consumers need not actively participate in the same time instance to consume the events. There are instances which can be defined to notify the consumers of a delay. During synchronisation decoupling, the event producers and consumers can choose to get asynchronously notified through

a call back mechanism. Decoupling the event producers and the consumers provides a huge level of scalability options to the entire architecture.

2.2.9 Context partitioning:

The event based systems support windowing techniques and use sampling or approximation to cope with the increase in event arrival. In a few proposals [Etzion and Niblett (2010)], distributed architecture is adopted in the event processing with scalability as their main concern. Distributed architecture through partitioning of the incoming events is viewed as a solution for scalability. This adopts scalability of event processing networks connected in an overlay with specialized routing and forwarding functions. Further classification of the partitioning on the events for event processing are categorized based on the centralized, hierarchical, acyclic and peer-to-peer systems [Eugster et al. (2003)]. Forwarding schemes adopted by brokers and the techniques of processing event patterns among the distributed brokers outline the methodologies involved in the event processing.

Under partitioning of the events, the time taken to process the events is considered as the total number of steps required to execute all the partitions. Events can be partitioned or split across multiple EPNs or the copies of same EPN executed in parallel, where one or more EPNs can be executed in each node. Consider the situated prompting illustrated in Figure. 3.6, which involve processing of events related to tasks A,B,C,D,E,F,G,H,I. Events related to each task stream from multiple sensors processed by EPNs are located in multiple nodes. For example tasks B,C,H and I are related to the baking dish, movement of ingredients such as sugar, butter and pear. The events arising from these sources are assigned to the nodes of distributed EPNs in a particular way, where the system should conclude activity I in Figure 3.6. In general it is impractical to run the system and count the number of event partitions or all possible interactions between EPNs to process the events arising from devices and equipmens in the kitchen. It is important to establish generic properties for event processing based on the context of the situated prompting. This research discusses an automated method to split and process the events irrespective of the arrival of events from multiple data sources.

As a design aspect, the partitioning of the event streams needs to be carefully analyzed to maintain the consistency of features extracted from the events. The selection and consumption of events from the various windows of time or context must be optimally performed to customize output and to increase the efficiency of the system. The size

of the output can be multiplicative with respect to the input. Expressing negation or the non-occurrence of the event, such as a dementia patient not answering a phone within a specified time, can be difficult during the context partitions. Messaging or publish subscribe systems improve filtering or stateless computing. However, the event processing systems include processing of both the occurrence and the non-occurrences of the events with the temporal or spatial constraints. The scalability in volume of context partitions needs careful design of the performance guarantee and output consistency. In this research, algorithms have been designed to maintain the semantic integrity and correctness of the resultant output through the Inter-EPN parallelisation.

2.2.10 Context state:

The scalability of the event based system associated with a long-running context partition requires accumulation of a large number of events. For example, a query running over a 48-hour period would acquire a large quantity of events to retain its internal state over the period of time. StreamMine [Heinze et al.] architecture describe state machine based operators to encapsulate reusable complex event processing operators with a set of transition rules.

1. Remove outdated events from the window
2. Add new event to the aggregation
3. Propagate the new events to the next phase.

All steps are integrated in to the single state machine by encapsulating transition between corresponding actions from events to trigger the removal of the outdated event. The parallelization is acheived by instantiating several identical state machines on different parts of the event streams. StreamMine processes the event stream as event batches, allowing users to define the size of the batch, trading off throughput for the latency with a compromise in the performance.

The execution of the distributed EPNs in multiple computing nodes can be modelled as a state transition system where the states correspond to the combined states of the EPNs and transitions correspond to the EPNs performing actions in parallel, where each EPN's action is defined in the CEP queries representing the context of the application.

2.3 Bin Packing

In bin packing, a set of items, capacity and a size function is given. The goal is to identify a feasible assignment minimizing the number of bins utilisation. A feasible assignment of N items is a partition PPn items, such that each Pk , the sum of sizes of the items in Pk does not exceed the capacity C. The objective is to decide whether all items can be packed using atmost N bins. The problem is known to be strongly NP-hard [Garey and Johnson (1979)]

Bin packing has been proposed as early as four decades ago [Kou and Markowsky (1977)][Gabay and Zaourar (2013)] and heuristics have been implemented in virtual machine consolidation. Challenges in resource allocation and memory allocation can be formulated as packing variable sized items into fixed size containers in order to minimise the total number of used containers. The bin packing problem is NP hard, even when restricted to one dimensional case and books such as[**?**] detail most of the theory literature on bin packing problems. To generalise, the systems that manage resources in a shared hosting environment benefit from good heuristics for bin packing. Focusing on virtual machine placements, several VM [**?**][Tang et al. (2007)][Lesh and Mitzenmacher (2006)][Pinheiro et al. (2003)] consolidation heuristics use bin packing as a base protocol to schedule resources.

Bin packing models are commonly used to address static resource allocation problems between a set of servers with known capacities and a set of applications or services with known amount of resources. In this research, EPNs are dynamically deployed in the virtual machines. The demands of the EPNs and the capacity of the virtual machines are likely to increase based on the arrival rate of the events. The growing trend in cloud computing comes with a challenge of utilising the compute power in an efficient manner. However, the utilisation power of the cloud computing is dependent on the management layer that schedules/assigns the EPNs in to the virtual machines. When assigning the EPNs, it is important to ensure that hosts are not overloaded while minimising the count of hired VMs. When multiple EPNs are placed in the same host, their load across each host is an additive measure, then the problem of statically assigning the EPNs to the VMs is a similar to the bin packing.

In this research each application/EPN need to be assigned to virtual machines located in multiple servers. Each server provides resources such as cpu, ram or disk, in order to run various application and processes. The optimal assignment of each EPN on multiple virtual machines is a NP hard challenge. A set of EPN needs to be assigned or reassigned to a set of virtual machines or servers. For example given n virtual

machines with a static capacity, feasible assignment of applications or processes need to be made. Local search based heuristics were one of the successful option[arts and lenstra]. Greedy randomised adaptive search procedure(GRASP) was enlarged by running several local searches through iterative process starting from diversified initial solution [feo and resende 1989]. When applying GRASP heuristics to machine assignment problem, bin packing arises as a sub problem for generating initial solutions.

Given the set of virtual machines with constrained resources and set of EPNs with dynamically changing arrival rates of events, the algorithm need to identify the count of virtual machines along with the optimal placement for the EPNs within the hired virtual machines. This is called as dynamic application placement [9]. The placement is an NP hard problem. This work will present an online approximation algorithm to solve this.

2.4 Event Driven Systems

Event driven systems [Luckham (2001)] emerged to process the flow of data, which are seen as an observable events. In event based applications, the data or events are pushed into the system that is composed of queries and a response is obtained. The event based model inverts the traditional data management assuming the users to be active and the data management system to be passive [Cherniack et al. (2003b)]. Event-driven architecture not only needs the current values of the events, but requires previous values and patterns from historic time series to trigger alerts in the essential conditions. Event-driven architecture has the characteristic of unpredictable load, where policy driven retainment of input data is very critical for operation. Intelligent resource management and graceful degradation strategy in event-driven architecture is highly essential when the incoming load is high. Much research such as NiagraCQ [Chen et al. (2000a)], Aurora [Babcock et al. (2002)], Stream [Arasu et al. (2004)] and Tribeca [Sullivan and Heybey (1998a)] addresses the development of event-driven architecture.

2.4.1 Aurora and Borealis

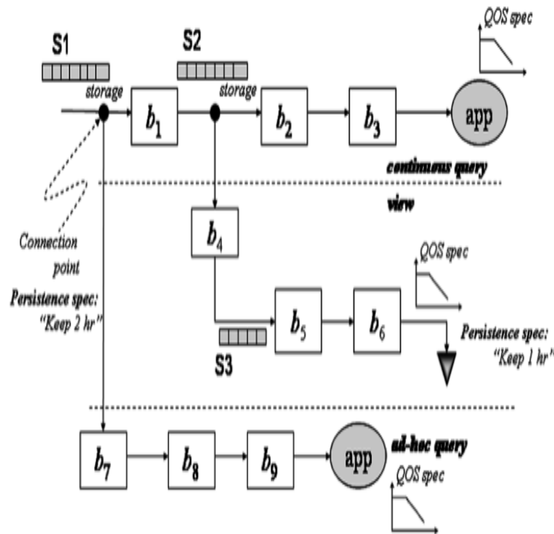
Aurora is an event processing system which implements data flow using a box and arrow paradigm. The data flow graphs enable a loop-free directed graph of event

processing operations. The graph is represented as three blocks, namely real time processing, views and ad hoc query processing represented as three individual layers. The topmost path represents the continuous query. The continuous query (CQ), once registered, issues results in response to the incoming data streams. Without an upstream node, a connection point as shown in Figure 2.4.1 represents unprocessed data streams stored for specific hours. The connection point records tuples processed by each stage. Once the inputs data stream has worked its way through all the reachable points, the data item is drained from the network. Persistence specification indicates exactly how long the items are kept in the data store.

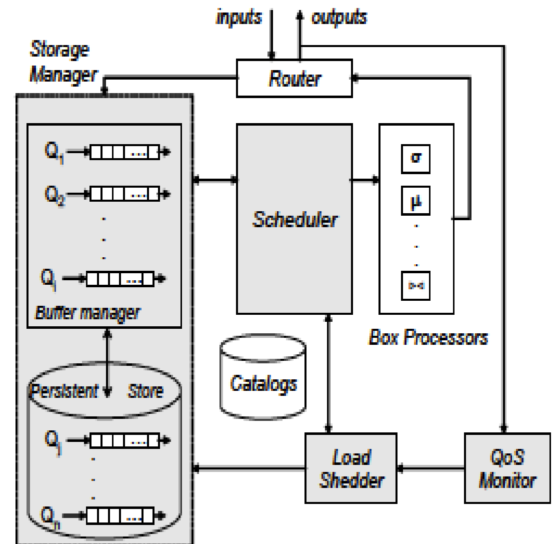
Aurora uses strategies such as combining boxes, inserting projections and reordering boxes to optimize the data flow network. The high data rate and amount of computation time determine the complexity of the optimization. During execution, the run time statistics are gathered to calculate the average cost of the execution and selectivity for each box. The traditional way of combining all boxes to a massive query and applying conventional query optimizations is not suitable for continuous event streams. The Aurora optimizer selects a portion of the network for optimization and corresponding sub networks are optimized. In order to reduce the overall processing cost, Aurora optimizations involve the user-supplied query structure. The approaches invoked are: (*i*) Allow boxes to queue many tuples without processing, leading to long tuple trains (*2*) Allow interbox non-linearity by combining boxes in to a bigger box where possible (*3*) Recording boxes (if commutative) based on the selectivity s, an input tuple i, generates $s \times i$ output tuples. This minimises the I\O operations and minimises the box calls made per tuple.

Adhoc Query optimizations: An operator sequence is attached to a connection point, which is a repository of historical information of some redefined durations. History information is stored in a B-tree form. If the connection point is a filter, when indexed, lookup is done on a B tree. If the connection is a join, the choice between indexed look up and merged sort is implemented based on the cheapest join. The optimizer converts the historic sub network in to an optimized one which is then executed. When the historic optimization is finished, Aurora switches the implementation to standard push-based data structures. The runtime architecture of Aurora consists of storage manager, scheduler, QoS monitor and load shedder. Load shedding [Carney et al. (2003)] is treated as an optimisation problem. Whenever the load is greater than system capacity, choices are made to drop the events for maximising the results. Both random and semantic drops are used for the load shedding.

Aurora Data Flow Diagram



Aurora Architecture



The Aurora storage manager (ASM) manages the tuples flowing in the data processing network in terms of queue management and maintains additional tuple storage required at connection points. In queue management, the windowed operation requires historic collection of tuples equal to the size of the window. If the network is saturated, the ASM manager evicts the lowest-priority main memory resident block known as 'tuple trashing'. Under the conditions of event distribution, this situation could change and different window sizes would make the catalog based remote management harder.

Medusa [Balazinska et al. (2004)] is an infrastructure built to support the federation operation of Aurora. Borealis is designed as a distributed data stream management system to support the scalablity aspect of Aurora. Each processor maintains its own persistent storage, load shedder, scheduler and optimiser.

2.4.2 Cayuga

Cayuga [Demers et al. (2007)] is an expressive language focused on the development of a query processor for streaming events with a specialized garbage collector. The query language operates based on the query algebra. The Cayuga automaton is implemented by an event language represented as a non-deterministic finite state automaton (NFA). The automaton is acyclic where states are assigned to a fixed relational schema as well as input stream. Three types of edges – namely forward, filter and rebind – are used to

define the event flow. Forward edges are those whose destination nodes are different from source nodes. The filter corresponds to the negation operator. Automaton instances need to store the attributes and values of the events contributing to the state transition. These attributes and values are called bindings. To avoid the overwriting of the events on the rebind edge, an attribute rename function is implemented. General algebraic expressions are handled by feeding the output events from one automaton in to another, forming a complex event-processing network.

The Cayuga data model allow events with simultaneous arrival time. During an epoch t, all events with detection time t are processed without instantiating a new automaton instance during the same epoch. At the end of each epoch, the state's pending instance is merged with the state's surviving instance. This allows recursive queries to construct an automaton that can loop forever in an epoch, generating unbounded event processing for events with the same detection time.

The Cayuga architecture comprises of event receivers (ER), priority queue (PQ) and client notifier threads (CN). The event receiver thread deserializes the incoming events, assigns temporal signatures and inserts the events in to the priority queue. The engine dequeues the events in the order of arrival, performing state transitions. For each automaton instance reaching a final state, an appropriate CN is notified. Client notification as well as insertion of new events in to the priority queue support recursive subscription. Cayuga uses the garbage collector (GC) to facilitate high performance and small memory footprint.

Scalable, stateful publish/subscribe system based on the non-deterministic finite state (NFA) was developed by Demers et al. (2006) using multiple query optimization techniques. Features are added to a publish subscribe system through query semantics defined using formal language operators. Evaluation of automaton instances and edge predicates for each incoming event is achieved using the indexing of the predicates. The static part of the predicates is handled using the Active Node Index (AN-Index) where the automaton instances of the static parts of the filter predicates are satisfied. The dynamic predicates increase with the event arrival and number of automaton instances and general indexing is not used. The total number of automaton instances can be very large at a given instance of time; however, the number of instances affected by the event is significantly lower which is referred to as Active Instance Index (AI-index). Identification of the instances affected by incoming events is complemented with the information on the forward and rebind (FR) edges of the traversing instance. The FR-index evaluates the static part of the FR predicate to make a decision on

whether to drop or advance the instance. The architecture of the Cayuga index illustrates the event stream subscription, notification and indexes. The implementation of the automata in a distributed setting is not investigated.

2.4.3 Streams

Stream: Stanford data stream management system supports large classes of declarative continuous queries over the streaming data. The prototype targets high performance through sharing state and computation across query plans. Query plans are composed of operators, queues and synopses. Constraints on the streaming data such as ordering, clustering and referential integrity are used to reduce the resource usage. Continuous monitoring and the re-optimization subsystem handle the fluctuating incoming arrival rates. The semantics of the system is based on two data types called streams and relations defined using an ordered, discrete *time domain* called Γ.

1. A *stream* S is defined as an unbounded bag of events, where s is a tuple and $\tau \subset \Gamma$ represents the timestamp of logical arrival time of tuple s.
2. A *relation* R is defined as $R(\tau)$ an instantaneous relation of the time-varying bag of tuples.

The abstract semantics uses three operator classes, namely *relation-to relation, stream-to-relation* and *relation-to-stream*. A declarative language, CQL (continuous query language) instantiates the operators of the abstract semantics. Stream-to-relation operators utilize the sliding window [2] concept based on tuples, time or partition of the streams. Relation-to-stream operators are classified as Istream (insert stream), Rstream (relation stream) and Dstream (delete stream) operators. Istream is applied to a relation R whenever tuple s is inserted into $R(\tau) - R(\tau - 1)$ at time τ. Dstream is applied to a relation R whenever tuple s is deleted from $R(\tau - 1) - T(\tau)$. Rstream is applied to a relation R where every current tuple is streamed at the current time instance. Each operator reads from an input queue (producing plan operator O_p) and writes to an output queue (consuming plan operator O_c. The operators Rstream, Istream and Dstream form the basis of building the EPNs to window and group the events for the continuous event processing.

Load shedding for the aggregate non-join queries are used in Babcock et al. (2004), where load shedding operators are part of the query plan parameterised using the sampling rate. To compensate the lost tuples due to the load shedders, aggregate values

calculated by the system are scaled appropriately to produce unbiased query results. Issues addressed as part of the project include distributed data streams [Olston et al. (2003)], adaptive query caching [Babu et al. (2005)] and resource management [Motwani et al. (2003)].

2.4.4 NiagraCQ

NiagraCQ [Chen et al. (2000b)] addresses the scalability of the continuous queries through grouping of the queries of common computation to reduce the I/O cost. The memory of the computing node forms the limitation of the grouping. Unneccesary query invocations are reduced by defining of the selection predicates to avoid large operations of grouping. Group optimisation approaches use dynamic re-grouping, query-split scheme, change and timer based queries. The incremental evaluation of queries uses push and pull based models for detecting heterogeneous data source changes and memory caching.

2.4.5 TelegraphCQ

TelegraphCQ [Chandrasekaran et al. (2003)] is focused on meeting challenges arising from high volume, varying event streams based on adaptive relational query processing. The Eddy [Avnur and Hellerstein (2000)] and Psoup[Chandrasekaran and Franklin (2002)] systems were extended to develop TelegraphCQ.

An *eddy* is a stream execution mechanism which is built on the continuous reordering of the operators in a query plan. Each input tuple in to the eddy maintains its history separately from the execution history using the *done* bitmaps and *ready* bitmap. An eddy routes a tuple to the succeeding operators based on the *done* bitmaps maintained in the execution history. Once the tuple satifies the predicates of an operator, appropriate updates are made on the bitmaps and tuples are returned to the eddy. Eddy continues iteration until the tuple has visited all operators. *Psoup* builds adaptive query processing techniques by combining ad-hoc, continuous queries by treating data and queries symmetrically. The new queries are applied to the old data and new data are applied to the old queries.

2.4.6 DISPEL

DISPEL [Griffis et al. (2013)] is a scripting language with a parser to generate data flow graphs in the form of executable workflow built on the OGSADAI [Mukherjee and Watson] architecture. The language expresses a directed graph where processing elements represent the computational nodes and the flow of data between processing elements indicates the connections. In the lower level, DIPSEL handles a registry validation and workflow optimizations.

The DISPEL registry maintains specifications for each processing element containing the description of workflow element, description of input/output streams, iterative behavior of the event processing between various stages, propagation rules from input/output and properties of the enactment engine to optimize workflow execution. The enactment engine buffers data units flowing through communication objects within a workflow, responsible for optimizing the mapping of processing elements to computing resources to minimize communication cost such as serialization, compression of data for long-haul movement and buffering techniques.

DISPEL uses three type of systems to validate abstraction, compilation and enactment. The language type system statically validates operations during compile time. The structural type system describes the format and low-level inter-operation values across processing elements. The domain type system describes application-domain related interpretation of the data. A streaming execution engine is developed where data is processed as streams through processing elements which are connected together into workflows. The workflows are managed by distributed gateways which optimize the workflow execution according to data locality and availability. The ADMIRE [Tran et al. (2011)] platform is based on the DISPEL scripting language.

2.4.7 Others

The SASE [Gyllstrom et al. (2007)] language uses both data-flow graphs and a non-deterministic finite state automation for simple and complex event detection. It receives sequences of events fed in to the query plan consisting of five operators connected in a pipeline. The *negation* operator expresses the non-occurrences of the event, *sequence scan* operator detects the positive events, *selection* filters detect sequences of events on predicates, *transformation* concatenates all attributes of the events in the detected sequence. The pipeline concept enables distribution of the processing stages in to a separate machine due to the stateless nature of the operators except

for *sequence scan*. Parallelisation of the *sequence scan* operator needs to see all the incoming events and maintain the states across each partition. Moving predicates and windows in the query plan introduces complexity and makes the computation more centralised.

Gigascope [Zhu et al. (2002), Cranor et al. (2003)] is a lightweight system specialised for the network monitoring. The queries submitted in Gigascope are analysed and optimised using a set of C and C++ language modules, compiled and linked to the run time system.

OpenCQ [Liu et al. (1999)] is a distributed event-driven query system using incremental materialised views to support continuous query processing.

Tribeca [Sullivan and Heybey (1998b), Sullivan (1996)] is based on a set of stream-to-stream operators using demultiplexing and multiplexing of the incoming event streams. The feature in the language to convert the single stream output with no notion of the relation makes it more expressive.

2.4.8 Summary

To summarise, this section surveys related work on the event driven architecture and event stream processing, focusing in the language design, scripting and querying methodologies. Systems such as NiagraCQ, Cayuga, Tribeca, Aurora focus on the specification of data flow architecture, continuous query operators and language centric features of the event based systems. In this research, these systems provide the baseline of the event driven architecture to build the event processing architecture for the ambient kitchen.

2.5 Parallelisation of Event Processing

Wu et al. (2012) derive the shared state and synchronisation overhead of operators based on parallelisation. The suitability and the optimal degree of parallelisation are assessed in terms of the bottleneck operator. An example of a KNN operator, the most commonly used in intelligent transport related applications, is used to illustrate the parallelisation. The count of the operators is estimated during the system initialisation along with the exclusive and non-exclusive reads or writes to the shared state. Configuration of the operator is represented in terms of the operator throughput. The

throughput is defined as $\frac{1}{T+\alpha_r\phi_r+\alpha_w\phi_w}$. T is the time taken by the non-parallelised operator to access memory and cost of the memory access is assumed to be negligible. The α_r and α_w denote the non-exclusive reads and writes on the shared operator. α_w and α_r denote the exclusive reads and writes on the non-shared operator.

The average time C is calculated to process the tuples for a specific application scenario. Wait time w is calculated based on the arrival rate. In order to determine throughput, an exclusive lock is required for the operator parallelisation. The operator in the waiting queue beyond a particular time is assumed to acquire lock and continue to process for a particular time unit t_s. Whenever a new lock request arrives from the incoming events, the operator continues the processing for another $C-w+t_s$ time units. The operator does not accept any new request from an incoming event source until the lock is released. The framework implementation uses System S and CAPSULE to determine the count of operators without a shared state. Multithreaded implementation is used to demonstrate the parallelisation of the shared state.

Access latency between the operators adds to the overall latency of the query. However, Wu et al. (2012) assumes the access latency between the operators to be an independent value along with the frequency of operator access. Overall latency prediction becomes challenging in this approach as the latency involved in the locking of the operators and inter-operator latency are unknown.

Cugola and Margara (2012) use parallel hardware to speed up the complex event processing. The algorithms to process the rules for parallelisation are built using operators such as sequences, aggregates, etc. The complex event-processing engine uses multi-core CPU for the simplest rules and the co-processor of the GPU (Graphics Processing Unit) is allocated exclusively for the implementation of the complex rules. CUDA, a parallel computing platform and programming model invented by NVIDIA, is used in this research. CUDA expresses parallelism by feeding hierarchically organised multithreaded memory blocks. Each memory block is decomposed in to finer pieces, processed co-operatively in parallel by all the threads within a block. Each thread has a private local memory for automatic variables and shared memory visible to all threads in the same block known as global memory. Thread blocks synchronise between each other through *barriers*. The programming model separates the device from the CPU in separate memory spaces. The CUDA architecture is built around multi-threaded Streaming Multiprocessors, where a host CPU invokes a kernel, the blocks executing the kernel are distributed in streaming multiprocessors. Multiple blocks may execute concurrently on the same streaming multiprocessor. As the blocks terminate, a new block is launched on the streaming multiprocessor. The *Automata*

based Incremental Processing algorithm is used to implement the complex event processing engine. Each sequence of the incoming events is mapped to a state and the transition between two states is denoted with *content and timing constraints*. Run time adaptation of the parallelisation is the key factor to reduce the queuing delay of the incoming streaming events. The parallelisation algorithms utilise multithreading and are limited to hardware with high processing specifications.

The dynamic placement query operators in a distributed continuous query system are addressed in Zhou et al. (2006a). Static placement of the query operators fails to address the server load, data arrival rate and communication overhead in a distributed event processing scenario. This work proposes data stream processing by dynamically migrating query operators or query fragments from overloaded query operator nodes to lightly loaded nodes. The work formalizes and analyzes the operator placement problem in the context of distributed continuous query systems. Multiple continuous queries executed in distributed nodes without any adaptation to the changes environment parameters lead to a suboptimal performance. The Performance Ratio (PR) metric to measure the relative performance of each query and the full system objective is defined initially. The heuristics on building a cost model are defined around the following parameters:

1. Load balancing among all processing nodes
2. Restrict the number of nodes over which the operators are distributed
3. Minimize the communication cost under balanced load and restricted query operator placement.

The load balancing is implemented using a well known diffusive load balancing scheme [Osman and Ammar (2002)]. Redistribution of load under operator level is done by adapting movement of query fragments as the finest migration unit. The communication cost is kept to a minimum by using a data flow aware load selection strategy. The research problem is addressed by building a cost model to evaluate the data flow aware load selection through dynamic movement of query operators. Problem formulation describes a system consisting of geographically distributed data stream sources, set of distributed event processing nodes interconnected by a local network. The data transfer costs from data source to processing node are assumed to be higher than the data transfer cost between processing nodes, so each stream of data is routed to multiple processors through a delegation node. Users impose a set of continuous

queries and set of operators (filters, window joins, etc.) of a single query to be distributed across multiple processing nodes. There is a large overhead of several epochs of operator placement leading to rerun the algorithms for good placement of operator thereby maintaining the communication overhead as small as possible. The system predicts the likely arrival rates for deciding on the number of computing nodes requirement, and this prediction is based on the prevailing rates and their variation trends. Their algorithm also takes the cost of hiring extra nodes as a parameter which is not considered in this research.

StreamMapReduce [Brito et al. (2011)] combines event processing and MapReduce to deliver low-latency capabilities. The parallelization and scalability of MapReduce is a consequence of the fact that the mappers are stateless and reducers are independent processes that aggregate the results from mappers. In this work, the two types of reducers (windowed and stateful) are implemented as stateful operators divided based on the type of the state. *Windowed reducers* consider a state composed of a window of events. *Stateful reducers* consider arbitrary states customizable by the user. Movements of events in windows reducers are based on redirection of events to the new node or a complete state transfer. The stateful reducers utilize the parallelism by exploiting the multiple cores in the computing node using a thread pool. Reducers use the state to store partial value results. The state recovery is done through a combination of in-memory logging in the upstream node and checkpoints to enable fault tolerance. The maintenance and propagation of the state in complex event processing applications are not considered.

StreamCloud [Gulisano et al. (2010)] is implemented as middleware to explore strategies to parallelise data streams, address the bottlenecks and overheads to achieve scalability. The scalability is provided through intra-operator parallelism where individual sub queries are distributed to a large set of shared-nothing nodes. The compiler transforms the abstract query in to a parallel query that is automatically allocated to a given cluster of nodes. This uses Borealis, a distributed stream processing system, to parallelize the queries using regular stream operators. A full query is implemented in the sub cluster of nodes in which each node processes the fraction of the input. The load balancer for the stateful queries needs to be enriched with semantic awareness.

System S4 [Neumeyer et al. (2010)] is a distributed, scalable, fault tolerant stream computing pluggable platform developed for unbounded event streams. The application is primarily developed for large-scale data mining executed in large clusters with commodity hardware. The architecture comprises of symmetric nodes with no centralized service simplifying the cluster configuration changes. Throughput increases

linearly as additional nodes are added to the cluster. There is no pre-defined limit on the number of nodes. The basic building block of the platform is message queues, processors, serializer and check pointing system in the backend. Cluster management tasks are leveraged using ZooKeeper [Hunt et al. (2010)] through an efficient coordination service of distributed applications. S4 design comprises of map reduce and actors models. The core design is built on processing elements (PE) to exchange events, i.e. messages holding key value pairs. The PE consumes events corresponding to the value on which it is keyed. New elements are spawned with the occurrence of every new key. Processing nodes logically host the PE. S4 routes events to processing nodes based on hash functions of all known keyed attributes in the event. The event listener in the processing nodes passes the event to the Processing Element Container (PEC) and invokes the appropriate processing element. The exchange of events is accomplished through TCP sessions. A task periodically deletes events from within regular time intervals. Whenever a server fails, the stand-by-server is automatically activated to take over the tasks with check pointing and recovery to minimize the state loss. The state of the processes is stored in the local memory. Upon failure, the state needs to be regenerated using the input streams. Nodes will not be added or removed in a running cluster. The platform lacks the control of assigning multiple high computational tasks in to the same machine.

2.5.1 Summary

The distributed stream processing middleware's such as Borealis, System S and others adopt pipelined parallelism, where the run time schedulers determine the placement of the operator that may have application level bottleneck issues. Scheduling strategies for the continuous queries use load shedding mechanisms, which assume sufficient resources during the system initialization. The systems face shortage of resources during the execution of the registered queries. It may not be feasible to satisfy the QoS requirements of all the registered queries. A natural solution is to discard the events or tuples with the goal of minimizing errors in terms of load shedding. The load shedding process triggers few important issues such as where to discard the event tuples, how many tuples to discard, when to discard and when to stop discarding the tuples. The run time optimizations can be achieved using this process, however, the semantic integrity of the processed events along with the quality of processing is compromised. This research focuses on limiting the queuing of events through distribution of incoming events in multiple computing nodes to satisfy the tuple latency and to obtain the final query results based on the predefined QoS specifications.

2.6 Scalability in Cloud Computing

Cloud computing offer services such as platform, infrastructure or software abbreviated as IaaS, PaaS or SaaS to deliver execution environments such as operating systems, middleware or application software.

Virtualization in cloud computing is the key enabler to drive the ability of the hardware to scale according to demand. The cost effective on demand approach in the virtualization helps to reduce the hardware capital expenditure. The ability to scale the resources based on demand improves the operational efficiency and agility of the system. Under virtualization, a single piece of hardware is split with the capability of running multiple independent resources in terms of operating systems with distinct management capabilities allowing the utilization of the underlying platform resources. The provisioning of the resources is dynamically configured based on the configured thresholds by the service providers. Underlying hardware usage and the networking device state is dependent on the CPU utilization, application state and networking I/O. The optimization of resources is provisioned or released based on the acceptable limits of the underlying server. To offer the linear scalability to the applications, virtualisation provides the following tasks:

1. Resource utilization
2. Scalability
3. High Availability

2.6.1 Resource Utilisation

Under virtualization, the core objective is to provide performance isolation, scheduling priority, memory demand, network access and disc access for all the concurrently running resources spawned in a single piece of hardware between multiple tenants. The application running inside one virtual machine is expected to be independent of the co-located application running inside another virtual machine.

Commodity virtualization stacks such as Xen or HyperV let the applications run on the cloud with very little performance guarantee. The provisioning of the resources for multiple applications and customers leads to possible interference between users and performance problems. Koziolek (2010) defines the potential problems such as

cache restriction, load management, admission controls and thread priorities for the performance isolation. Krebs et al. (2012) introduce four approaches to enforce performance isolation focusing on QoS metrics using the admission control and the thread pool mechanisms. The round robin approach introduces specific queues for the each tenant. It provides good isolation, however it is not sufficient for overcommitted systems, as it cannot fully leverage the unused resources from some of the tenants. The blacklist method triggers a quota checker and monitors the violations in the allocated quota for each tenant. This method provides good isolation for the disruptive workloads to achieve different QoS for throughput. All the tenants use the unused resources equally. The separate thread pool method allocates limited size to the pools to isolate tenants from each other. The metrics are measured by triggering disruptive workload to various tenants occupying the specific hardware. This method provides good isolation and QoS differentiation is achievable by using different thread sizes. When the number of potential threads spanned by the tenants is higher than the optimal working point, the congestion is observed in the server.

Mei et al. analyzed idle instances management and the performance impact of co-locating applications in terms of throughput, performance and resource sharing (i.e. CPU, memory) effectiveness. The impact of running idle instances is studied to provide useful insights to cloud consumers for managing idle instances more effectively to scale the applications. This work provides the co-location impact of network centric I/O applications. A resource usage model is used to explore and quantify the performance gains and losses relative to the various configurations in the guest domains and applications. Measurement analysis reveals that applications should be arranged carefully to minimize unexpected performance degradation and maximize desired performance gains.

Koh et al. (2007) focus on the hidden contention for the physical resources caused by various virtual machines (VM) hosted on one physical host or a piece of hardware. Workloads characteristics are used in the mathematical models to predict performance interference. Clusters of applications are identified to generate metrics for series of system level workload characteristics such as average CPU utilisation, cache hits and misses, virtual machine switches, I/O blocks, disk read and disk write. If an application runs in a VM by itself it can use techniques to use a deterministic share of CPU. However, when the hypervisor switches to another VM, the new VM regains access to the CPU. This process is opaque in the physical representation of hardware, applications and guest OS are not aware of the changes leading to the difficulty in the optimisation techniques. Based on the performance interference, application

clustering is done based on the linear regression analysis of performance score prediction. The workload behaviour based on the streaming events leading to network I/O impacts is not included in the prediction of the resources.

Q-Cloud [Nathuji et al. (2010)] is a QoS aware control framework which tunes resource allocations to mitigate performance interference effects. It uses online feedback to build a multi-input multi-output (MIMO) model to perform closed loop resource management for specifying multiple level QoS in terms of application level Q-states. Lowest Q-State is the minimum performance level required by the application but higher Q-States may be defined by applications willing to pay higher levels of QoS. The underutilised resources are enabled to elevate QoS levels to improve the system efficiency and utilisation up to 35% using the Q-states. Static estimation of the resource requirements is determined based on profiling the virtual machines on a *staging server*. During the placement of virtual machines, the cloud scheduler leaves a prescribed amount of unused resources on each server to utilise in the online control. The unused capacity is called *'head room'*, which is determined based on the typical excesses used in interference mitigation.

Amazon kinesis, is designed to handle real time streaming events from hundreds of streaming event producers such as social media feeds, application logs, click streams, etc. Shard is the base throughput unit of amazon kinesis stream. One shard provides capacity of 1MBsec data input and 2MB/sec data output. One shard can support 1000 PUT records per second. The count of shards are dynamically added or removed proportionate to the CPU utilisation. Amazon Kinesis provides prebuilt libraries to configure the incoming data sources directing it to various applications such as Storm, Amazon S3, Amazon Dynamo DB, Amazon Glacier and others. It acts as a managed service to elastically scale the real-time processing of streaming data at massive scale. The count of shards required for specific application on a stated arrival rate in unknown to the users. The users need to program the real time scaling of the shards or estimate on the count of shards. For example, a method call 'SplitShard' is invoked to split one shard in to two new shards in the stream of events. This increases the capacity of shards to ingest and transport more events. However, given the nature of application, predicting the count of shards will be useful to plan the cost and resources. Splitting of shard requires a logical and analytical framework based on the application requirements, which is deployed to process the event. To the best of our knowledge, the amazon kinesis utilise the CPU usage as the main parameter to scale the shards, however it does not implement the dynamic prediction or estimation of the CPU requirement based on the incoming arrival rate of events.

Google big query is a fully managed cloud based interactive query service for massive datasets. Dremel is the query service that allows SQL-like queries. Big Query provides REST API available through Dremel targeting the third party developers. Dremel parallelise each query and run it on tens of thousands of servers simultaneously. Data is stored in the columnar storage to achieve high compression ratio and scan throughput. Tree architecture is used for dispatching queries across thousands of machines in few seconds. The architecture forms a massively parallel-distributed tree for pushing down a query in to the tree and then aggregating the results from the leaves. By leveraging this architecture, Google implemented distributed design for Dremel and big query share the same underlying architecture and performance characteristics. Big query is designed to handle structured data using SQL. For example a table is defined in big query with column definition and then import data from a CSV file in to google cloud storage and then in to the big query. Big query is suitable for OLAP or BI usage where most queries are simple and done through aggregation and filtering by a set of columns (dimensions). BigQuery is an externalised version of Dremel with very similar features to MapReduce and Bigtable. While MapReduce is suitable for long-running batch processes, BigQuery is preferred for the adhoc OLAPBI queries that requires very high response time. The cloud-powered parallel query database present high full-scan query performance and cost effectiveness compared to the traditional warehouse solutions. The technologies such as BigQuery and MapReduce are more focused on the long-running batch processes. The real time event based processing domain is currently dominated by the continuous query language and continuous event processing(CEP). In this research, the focus is towards using exemplars from real time streaming events, where the arrival rate of events changes dynamically. The frameworks such as Tenzing[Tolosana-Calasanz et al. (2012)], Dremel [Google] , are mostly used in the batch processing. The work in this research is focused more towards creating algorithms to estimate, predict and place the EPNs. The EPNs are the processing units, which are developed using lightweight, java program modules consisting of continuous event processing system.

Cloud computing infrastructure exhibit performance variability based on the collocated applications, resource sharing policy such as EC2 credit share, etc., over the period of time. The same virtual machine instance has different resource characteristics over time due to multi-tenancy and varying workloads in the data centre in accordance to the placement and diversity of the commodity hardware [Iosup et al. (2011)]. Scheduling strategies that assume deterministic and homogenous cloud behaviour fail to deliver expected QoS to the applications running on clouds. Systems such as S4, Storm and Spark provide generic auto scaling solutions such as operator

scaling and data parallel options [Zhou et al. (2006b)]. Several solutions to leverage cloud elasticity has been proposed [Gulisano et al. (2012)]. However, most of the systems [Tolosana-Calasanz et al. (2012), Satzger et al. (2011)]] consider variability in the incoming data and ignore the variability in the underlying cloud infrastructure. In this research, heuristics are developed to consider the changes in the incoming event arrival and the underlying infrastructure through hardware metrics monitoring probes. The count of VMs are estimated dynamically based on various arrival rates of events. Queuing theory model is used to predict the hiring of virtual machines. The algorithms developed in this research are used to predict the response time of the deployed applications to scale the virtual machines hired from the cloud computing, proportional to the arrival rate of the incoming events. The table 2.1 categorises the reference publications in to various topics such as response time estimation, dynamic placement, scalability in cloud and event processing parallelisation. This work uses the concepts from all these areas to find an appropriate solution.

2.6.2 Summary

The enabling factor for cloud computing is the virtualization techniques provided by XEN [Barham et al. (2003)], Hyper V [Velte and Velte (2010)] and others, allowing execution of multiple operating systems and applications to run simultaneously on the same hardware. The performance isolation [Koh et al. (2007), Nathuji et al. (2010)] and CPU fair sharing [Kazempour et al. (2010), Liu et al. (2010)] in a virtualized environment address the resource contention issues associated with the real time data intensive applications. The assumption of the performance isolation in the virtualization environment fails under high I/O requests and heavy CPU intensive jobs. The combined effect of multiple process intensive jobs running in the same virtual machine leads to highly fluctuating response time scenarios. To maximize the benefits and the effectiveness of the virtualized cloud computing environments, deeper understanding of the key factors related to effective resource sharing needs to be achieved to address the performance bottlenecks. Based on this background research in the performance interference aspects, baseline setup criteria such as deterministic CPU allocation for the VMs, are devised for the cloud computing environment to avoid the performance inferences from other virtual machines in the hired hardware.

Several solutions to leverage cloud elasticity has been proposed [Gulisano et al. (2012)]. However, most of the systems [Satzger et al. (2011), Tolosana-Calasanz et al. (2012)] does not consider variability in the incoming data and ignore the variability in the

TABLE 2.1: Related Work Classification

	Response Time Estimation	Dynamic Placement	Scalability Cloud Computing	Event Processing Parallelisation
Luckham and Schulte (2008)		✓		
Etzion and Niblett (2010)	✓			
Arasu et al. (2004)	✓			
Muthusamy et al. (2010)	✓			
Li et al. (2007)	✓			
Tatbul et al. (2003)	✓			✓
Cherniack et al. (2003a)	✓			✓
Iosup et al. (2011)			✓	
Zhou et al. (2006b)				✓
Gulisano et al. (2012)			✓	✓
Tolosana-Calasanz et al. (2012)			✓	✓
Satzger et al. (2011)			✓	✓
Koh et al. (2007)			✓	
Nathuji et al. (2010)			✓	
Kazempour et al. (2010)			✓	
Liu et al. (2010)			✓	✓
Brito et al. (2011)			✓	✓
Osman and Ammar (2002)		✓		✓
Zhou et al. (2006a)		✓		
Cranor et al. (2003)	✓			✓
Griffis et al. (2013)	✓			
Demers et al. (2007)	✓			✓
Chen et al. (2000a)	✓			
Babcock et al. (2002)	✓			
Arasu et al. (2004)	✓			
Sullivan and Heybey (1998a)	✓			

underlying cloud infrastructure. In this research, heuristics are developed to consider the changes in the incoming event arrival and the underlying infrastructure through hardware metrics monitoring probes. The hiring and releasing of VMs are estimated dynamically based on various arrival rates of events. Queuing theory model is used to predict the hiring of virtual machines. The algorithms developed in this research are used to predict the response time of the deployed applications to scale the virtual machines hired from the cloud computing, proportional to the arrival rate of the incoming events.

Chapter 3

Real time streaming application

3.1 Introduction

The modelling and implementation of the event processing is accomplished based on the real time streaming application called ambient kitchen, which was supported by the EPSRC funded SiDE (www.side.ac.uk) project. The terminology event used within this thesis denotes a notable action which could be a transition of an entity or any external information feeding in to the ambient kitchen. The events can identify or trigger actions (cutting, chopping, stirring, etc.) or initiate multiple processes (prompt users, direct assisted actions etc.) registered within the kitchen or to the systems linked with the ambient kitchen. From an event processing perspective, the ability to detect and respond to the events are defined as one of the key goals for the definition of the Event Processing Network (EPN). The conceptual building block for the design of real time information management within the ambient kitchen are the event processing networks(EPNs). EPNs provide core functions of event-processing logic connecting event producers and consumers through events arising from various pervasive sensors and systems. EPNs describe the event processing system as a collection of interactions between event producers, processing agents and consumers. EPNs combine analytical techniques to predict events, mine patterns and embed real-time analytics to trigger decision support systems. The modelling and implementation of EPNs is supported by the requirements driven by the real time streaming application from the ambient kitchen.

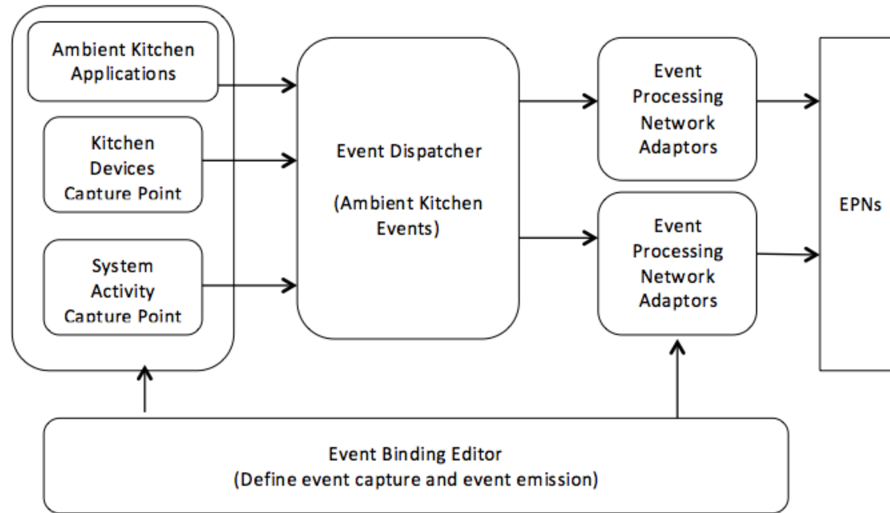


Figure 3.1: Ambient Kitchen Overview

3.2 Ambient Kitchen Overview

Ambient kitchen is a configurable, extensible and portable framework that enables an experiment set up of smart kitchen to support dementia patients. In this research, the ambient kitchen is used as a prototype for the generation and submission of complex real time streaming events workloads in to cloud computing environments. Through the accumulation of statistical information regarding the incoming arrival rate of events generated from the ambient kitchen execution, the framework is able to carry out a response time of the underlying ambient kitchen event processing applications deployed in cloud as IaaS systems.

Ambient kitchen overview is represented in the above figure 3.2. The event-binding editor defines the capture and the emission of the events from the kitchen devices and the system activity. The criteria for capturing events and the data from the kitchen are known as the capture specifications and are saved in the event bindings. Each capture specification relates to a particular capture point, for example, the accelerometers embedded in the knife in kitchen will stream events in terms of X, Y and Z coordinate movements. The event emission options including the required format and the destination are also defined in the event bindings.

Ambient kitchen application checks the capture specifications whenever an event capture point is run. If the capture specifications are active, the devices check the filter criteria and capture the specified events when the pre-condition is evaluated to true. Captured events are passed to the event dispatcher for the emission. Event dispatcher passes each event to the EPN as specified by the event binding. The Event processing

network adaptors formats the events and emits to the destination EPN according to the configuration defined by the event bindings.

3.3 Benchmarking Application

The human activities in the ambient kitchen rely on the statistical analysis of features extracted from the accelerometer events. For each dataset, a set of 23 statistical attributes such as mean and moments are calulated. The events are split in to frames with length of 64 events and 50% overlap of events. An investigation of the food preparation actions with embedded three axis accelerometers in the kitchen utensils (such as knifes, spoon etc.) was carried out using volunteers to use the devices in the kitchen. Analysis of events emerging from these experiments was utilised to generate streaming events for this research. At the application level, the objective is to identify the low level food preparation activities from the kitchen utensils which include identifying activities such as chopping, coring, stirring, slicing, dicing, eating, peeling, spreading, scooping, scraping and shaving.

Figure illustrating Ambient Kitchen

Embedded Sensors

Slice and Dice [Pham and Olivier (2009)] is a first step in the development of a low level activity recognition framework for food preparation. The framework detects activities through four steps: Data Fetcher, Data Segmentation, Feature Computation and Classification.

1. Data Fetcher: The raw data from accelerometer signals are preprocessed using noise removal and filtering techniques. The sensors are embedded inside the

kitchen utensils. During cooking activity, events are streamed from the utensils at a frequency of 1Hz. Events are segmented in to sliding windows for computation.

2. Feature Computation: 23-feature vectors such as mean, standard deviation, entropy, energy, and correlation between two axes are computed from a sliding window.

3. The classification algorithms used in the activity recognition framework are Decision Tree C4.5, Bayesian Network and Naïve Bayes. In the experiment with 20 subjects performing mixed salad and sandwich preparation in the Ambient kitchen, the framework was rigorously evaluated and has shown an overall accuracy of more than 80%.

Figure 3.2 describes the overview of system architecture where the real time streaming events generated from the ambient kitchen are used to detect the presence of the person, actions and behaviour patterns of the kitchen occupants. One of the challenges in this environment is the integration of new devices in to the physical kitchen and standardisation of the widely diverse event processing life cycles. The heterogeneous environment triggers a need for an event processing stack with self-managing properties since solutions depending on the centralised management, configuration and adaptation exhibit a single point of failure. The research motivation is to provide event based infrastructure to seamlessly integrate context aware services, independent of devices and subjects to create a high level situated prompting environment.

3.4 Autonomous Event Processing Network (EPN) Design

The event processing agents are viewed as a collection of event processing elements, event producers and event consumers linked by channels for communication. The EPNs are represented as a graph as shown in Figure 5.1, composed of processing elements constituting the node of a graph. The nodes have input or output terminals for the flow of events from one stage to another.

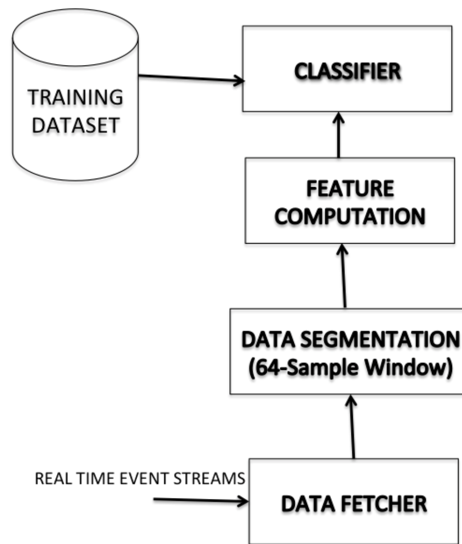


Figure 3.2: Activity Recognition Architecture

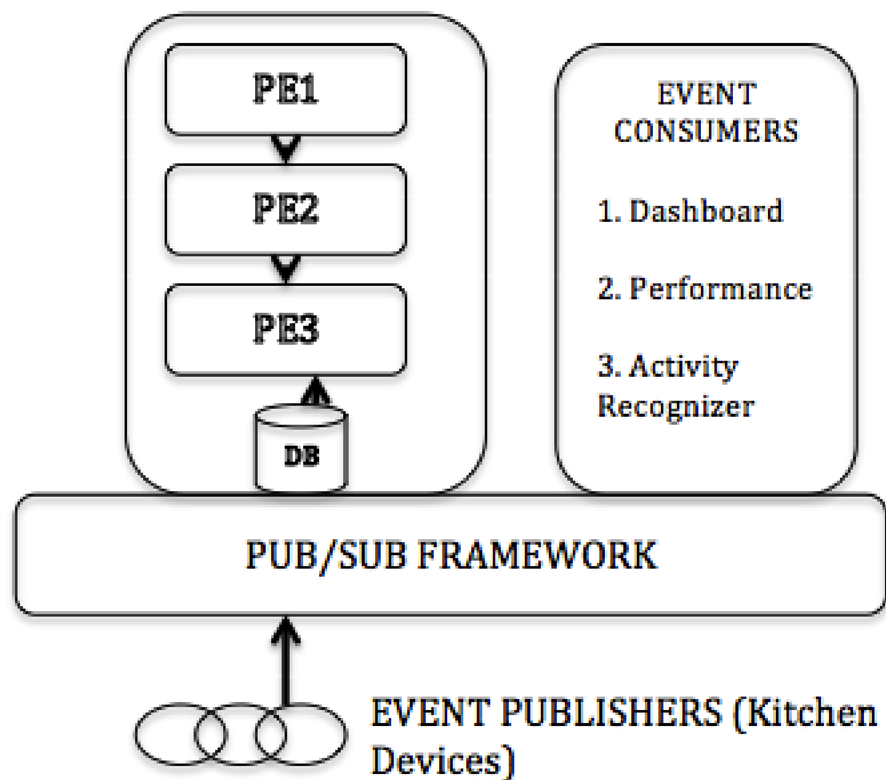


Figure 3.3: Event Processing Network (EPN) overview

3.4.1 Event Producers

The event producers in activity recognition are from a discrete piece of hardware to detect the acceleration. In the demonstration prototype, Pham and Olivier (2009) implemented ambient kitchen using the Nintendo Wi Motes and also developed sensors to exclusively measure acceleration. ADXL330 accelerometers were used to sense the acceleration along three axis to measure motion, shock or vibration. The embedded sensors are set to 40hz providing approximately 40 samples every second.

The acceleration events measure motion of the sensor itself in three different axes namely X, Y and Z. The event types are defined for acceleration data with semantic relations between the device, users and the generated features. The payloads for each event type for the activity recognition experiment are illustrated in Tables 3.1, 3.2 and 3.3. The event producers constitute many sensor devices embedded inside the knife, spoon, utensils etc. A subset of the events generated from these devices is used to evaluate the response time.

Table 3.1: Schema for *Acceleration Events* in activity recognition

Attribute Name	Data Type	Semantic Role
X	Int	Common Attribute
Y	Int	Common Attribute
Z	Int	Common Attribute

The meta-data for the entire event producers such as devices and subjects of the ambient kitchen are encoded within the devices.

Table 3.2: Metadata for *Acceleration Events* in activity recognition

Attribute Name	Data Type	Semantic Role
Device Id	String	Reference to Device Entity
Name	String	Common Attribute

3.4.2 Processing Elements:

Each processing element try to recognise the activities happening within the kitchen such as cutting, chopping, choring, etc. Various stages on the processing of events is described in this section.

The PE1 (Processing Element 1) as illustrated in Figure 3.3, acts as a filter in the EPN and windows the incoming events based on the specified dimension (temporal,

spatial (according to location of devices) and sequence of arrival). The expression that filter work is solely based on the value of the attribute called deviceId and will work as intended against any event type that contains a deviceId attribute. The ordering of the events is based on the arrival of the event from the message queue.

Each set of incoming events is grouped in to a set of 64 or 32 events called a Window. Landmark and sliding windows are a few of the windowing schemes commonly used in event processing. For landmark windows the older end of the events in a window are fixed and the newer end of the window changes with the arrival of the new tuple. In sliding windows both ends of the window move in unison with the arrival of the new tuples. In the activity recognition, a sliding event interval is utilised where each window has a specific count of overlapping events. The new window is opened with half of the tuple from the previous window as shown in figure 6.1. Based on the intensive machine learning experiments in [Pham and Olivier (2009)], the optimum window size is fixed as 64 for the PE1.

In the activity recognition use case, the window of 64 tuples describing activities is abstracted as mean, standard deviation, entropy, roll and pitch for that period. The first 32 tuples of a given window must be the same as the last 32 tuples of the previous window. Accuracy of detection is sensitive to this degree of overlap. Window size is determined based on the previous research [Pham and Olivier (2009)].

The processing element 2 (PE2), as illustrated in figure 3.3, acts as a transformation operator, where the event instance is processed depending on other instances processed by the operators. They differ based on the kind of transformation it performs such as stateless or stateful. It also depends on the single input stream or multiple input stream with functions such as aggregate, sort, resample, map, read, union, join, update, etc., The functionalities differ according to the event processing engine providers. For example, mathematical aggregate functions such as mean, standard deviation, energy and entropy [Pham and Olivier (2009)] are used.

$$\text{Energy} = \frac{\sum_{i=1}^{n} x_i^2}{N}, \frac{\sum_{i=1}^{n} y_i^2}{N}, \frac{\sum_{i=1}^{n} z_i^2}{N} \tag{3.1}$$

$$\text{Mean} = \frac{\sum_{i=1}^{n} x_i}{N} \frac{\sum_{i=1}^{n} y_i}{N}, \frac{\sum_{i=1}^{n} z_i}{N} \tag{3.2}$$

$$\text{Standard Deviation} = \sqrt{energy(X) - mean(X)^2}, \sqrt{energy(Y) - mean(Y)^2}, \sqrt{energy(X) - mean(X)^2} \tag{3.3}$$

$$\text{Entropy} = \sum_{i=1}^{N} p(x_i)log_2p(x_i), \sum_{i=1}^{N} p(y_i)log_2p(y_i), \sum_{i=1}^{N} p(z_i)log_2p(z_i) \tag{3.4}$$

The processing element PE2 transforms the raw data from the accelerometers in to 'features'. Once the initial computations are completed, the features are used to map the semantic roles of the data such as cutting, stirring and other activity. The attributes and the payload of the features are described in the table below.

Table 3.3: Schema for *Features* in activity recognition

Attribute Name	Data Type	Semantic Role
Device Id	String	Reference to device entity
Features	String	Common Attribute
Activity	String	Common Attribute
Current Time	Timestamp	Common Attribute

The Processing Element 3 (PE3) as illustrated in figure 3.3, acts as a pattern detection operator emitting one or more derived events. This detects an occurrence of the specified pattern in the incoming stream. An exhaustive machine learned patterns for the activity recognition experiments run by Pham and Olivier (2009) is used in this experiment. The patterns are stored based on the optimization requirements as a trade off between the performance and recoverability. For example when the pattern is stored in a database the recoverability is higher but may conflict with the performance goal. The implementation of the state management function has various choices such as disk based persistence, in-memory database or mixture of the approaches. This experiment favoured the use of disk based persistence where a compromise is made on the scalability in the number of context partitions or the quantity of the events accumulated within the context partition. Nearly 5000 patterns of activities are stored on disk of the PE3. The output from this stage is an identified activity such as cutting, chopping, peeling etc. which is sent to subscribers for further situated prompting.

3.4.3 Event Consumers

The event consumers in this experiment are java messaging queues which collects the activity recognised by all the devices and sensor equipment embedded inside the ambient kitchen. The experiment is moved to the next activity by aggregating the activities with temporal dimensions to create assistive prompting for the users in the kitchen who are likely to be dementia patients. The consumers act as input to the EPN for the situated prompting experiments. To focus on the distribution of the event processing network (EPN) seamlessly in the cloud, the non-functional aspects of the experiments are monitored using a dashboard. The real time dashboards were designed targeting three major objectives as listed below:

- Assess current environment quickly
- Comprehend the severity of the situation
- Respond or react in timely manner

3.4.4 Dashboards

Dashboards in general provide a quick summary of the situation to visualize the most relevant and important aspects of the EPN performance. The design of the dashboard enhances the display of the real time events with respect to the projected or historic information comprehending the deviation in response time. The alerts or the actions relevant to the specific conditions were designed to appear in the dashboard. The throughput, latency and arrival rates of the event stream were the key factors displayed in the dashboard as illustrated in Figure 3.4 .

3.4.5 Activity Recognition Summary:

EPNs process the signals from the sensors and recognise the activities that the source utensils are engaged in. Note that when activities in the kitchen increase, the incoming arrival rates of the events increase and *vice versa*. Activity recognitions from EPNs are passed to a prompting system which identifies any unduly long pauses in the activity sequences and instructs the object holders accordingly. The event streams from the accelerometers and the recognised activities are used as an exemplar data set to evaluate the response time, reconfiguration and parallelisation of EPNs.

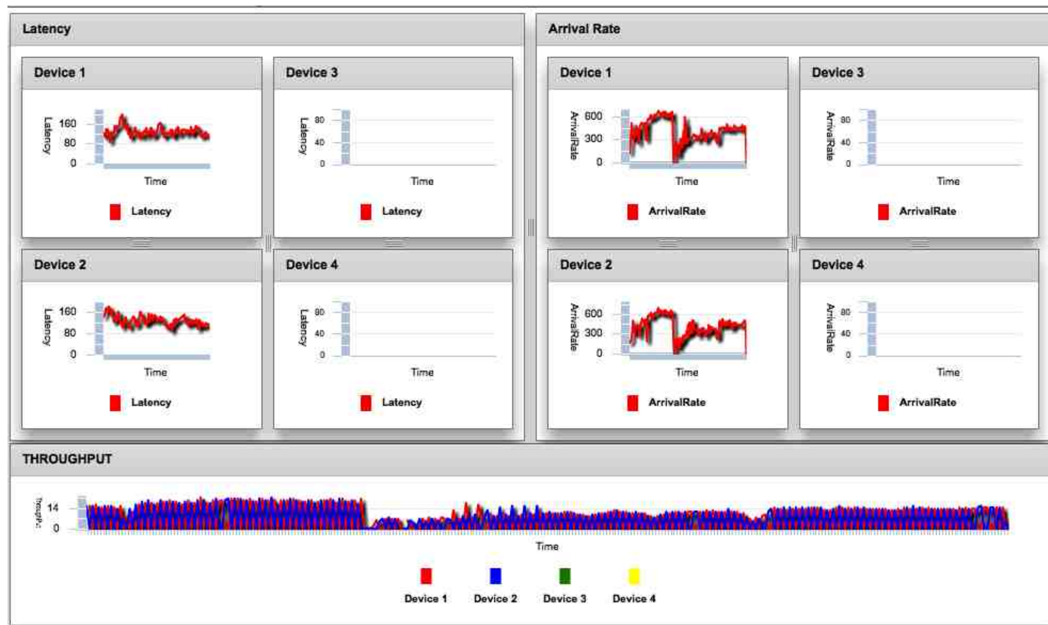


Figure 3.4: Dashboard to visualise non-functional properties of EPN

Classification of Event Processing:

The complexity of the event processing networks has been classified as simple, complex or intelligent event processing based on the inter-relationships between the events. Each of them is defined using the main case study, ambient kitchen.

Simple event processing can be illustrated using events generated from pressure sensors embedded in the floor. The movement of the individuals within a room is estimated using the data from pressure sensors. When an individual walks over pressure sensors in the floor, a signal is generated indicating the occupancy of the floor. Based on the occupancy of the floor and calibrated thresholds, two different states on the floor can been determined:

1. Activity
2. No activity

Sensors transmit events to compute the states of occupation in the floor based on various thresholds. From an event perspective, a simple event processing application detects the breach in threshold level and issues alerts. This is a deterministic example where the derived event flows imply a particular state of occupancy of the room.

Events in the event processing world only approximate the real world scenario. The floor in a building can span across hundreds of square meters covering the building.

An individual entering one segment of the room should exit from the room at some point of time. If one segment of the pressure sensor generates events continuously for a long time, there is a good chance that a fall has occurred. When a pattern of floor movement threshold is detected in a given instance of time, a notification can be issued to high level systems on likely accidents. This is an approximate detection of the real world occurrence where a human agent detects the pattern of user activity. Although an event processing application detects this pattern and sends notification to the high level systems, the detection of this pattern is neither necessary nor sufficient indication that the situation is likely to occur.

In this case of simple event processing, the occurance of the event is an approximation of the situation as there may be false positives or false negatives.

For example, in the ambient kitchen use case, queuing systems or topic based publish subscribe architecture are utilized to solve the complexity of the integration with multiple producers and consumers of event. The illustration in Figure 3.5 uses an exemplar from the ambient kitchen to demonstrate complex event processing flows for multichannel sensors and activity monitoring application. In the aggregated flow of data from sensors, the aggregated levels of computation from the accelrometer based sensors are fed in to the system. If the detected state of the processed events from an accelerometer indicate the utilisation of utensils with regards to meal making for more than a pre-determined time frame, more processing requirement from the EPNs can be anticipated and propagated towards the system design for resource allocation to process the activities in the kitchen.

The measurement from accelerometer levels (x, y and z) are sent through the event channels. When the peak level of activity occurs during lunch time or evening, event processing requirement increases in the network. The processed events from the system are utilized in the subsequent stages via alerts or notifications. The decision support (e.g. mobile phone app) systems subscribe to these feeds to comply with the monitoring of the users in the kitchen.

The second and third flows show two variations of accelerometer and pressure sensor. Both flows originate from sensing devices at the bottom left hand corner of the illustration. Every state in the system generates an event. A local EPN evaluates the events to aggregate events over the period of time. The outliers of events are generated based on the deviation of the aggregated events from the normal profile.

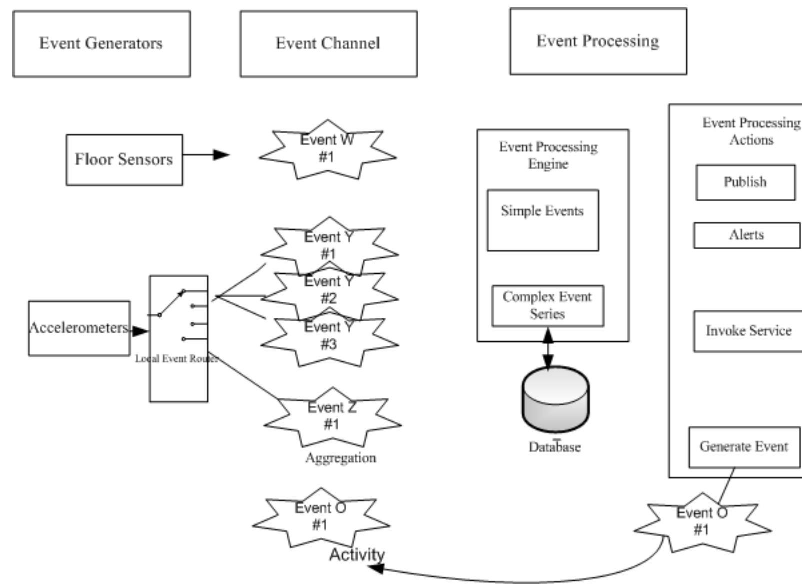


FIGURE 3.5: Ilustration of complex event processing

In complex event processing, the changes in state are monitored during the occurrence of events and applications respond in a timely fashion. The complexity of the event processing application varies based on the following factors:

1. Dynamic variations in the system due to addition of new event sources, new interactions and response criteria
2. Quantity and complexity of event sources
3. Quantity of subscribers involved in the event consumption
4. State and context management
5. Creation of derived events in the multiple stages of the complex event processing. For example, introduction of reflection and introspection leads to the creation of new events in the downstream processing.

The illustrated exemplar from ambient kitchen is an extract from a few low level event flows leading to the creation of an architecture of complex event processing systems. The event processing leads to the identification of low level activities based on the sensor data to monitor the ambient kitchen. The processing of the complex events entails low response time and timeliness of the outcome. Based on the appropriateness of the given problem, the best approach requires a response centric event processing system.

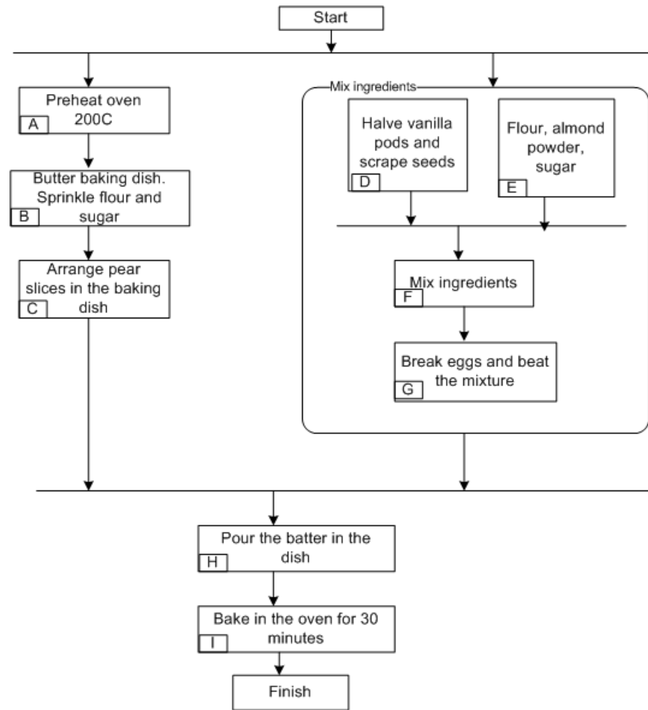


FIGURE 3.6: Ilustration of Intelligent Event Processing (IEP)

Intelligent event processing generally utilizes complex rules of inferences and logical proofs from the events to progress between various stages in the event processing workflow. A logic proof is represented through arguments derived from an application related hypothesis (assumptions). An exemplar from situated prompting scenario of an ambient kitchen in Figure 3.6 illustrates the Intelligent Event Processing as shown in the activity model described below. The ambient kitchen uses embedded sensors within the utensils and equipment to provide assisted living in smart homes. Recipes are used in the kitchen to prompt dementia patients during cooking. Each prompting activity is broken down in to various stages as described in the pseudo code for situated prompting. Progression from one stage to the next is triggered through sensing events from the kitchen. The complexity of the pattern recognition increases with the number of the sensors, event rate, complexity of the application and heterogeneous data sources.

The ambient kitchen example relies on complex pattern matching where each stage is justified by a rule of inference. Most of the rules of inference originate from tautology. Tautology is used for drawing conclusions derived from the sequence of event occurrences. In the stated activity model, it represents a multi step action interacting with a number of different objects and ordering of events. The event flow is augmented by the probability of distribution of actions representing the horizontal bar in Figure 3.6 . In the context of the activity model, a balanced data sheet of various events is

derived through the task-ordering. Using a sample of task-ordering from the activity model, event flow architecture is represented as listed below:

1. Event flow model for the feeds from multiple sensors
2. Computational process based on the fusion of events to aid multi-modal situated prompting.

Each step of the hypothesis follows laws of logic supported through the occurrence of the event. The time taken by each observable event in Figure 3.6 (A,B,C,D,E,G,H,I) and the time elapsed between two subsequent events are two important attributes contributing to the temporal signature of the activity. The time taken to mix the ingredients (event F) will be less than the preheating oven (event A). The activity model is defined as the finite sequence of events over a finite period of time where each event in the activity is the occurrence of the event.

The sequence of events offer scope to define declarative representation of small, incremental, independent logical units of event processing. During the sequential occurence of events, the correctness, termination and the response time present challenges as listed below:

1. Considering the sequence of rules encapsulated in multiple EPNs, the system needs to produce an accurate summary of the incoming and outgoing events from all EPNs.
2. The erratic flow of the incoming and outgoing events might result in the termination of multiple EPNs without producing any output. Careful design of the EPNs in terms of acyclic graphs of event flow is important to attain the successful termination of the system.
3. Predicting the response time in terms of the overall computation would help in designing the count of new strands of parallel computation.

Verifying properties such as correctness, termination and response time require separation of the application logic (such as ambient kitchen) from the execution engine (CEP engine such as ESPER, streambase, etc.) eases implementation of the event flow architecture. However, the research challenge lies in encoding the temporal and structural information gathered in the events, to predict the response time and model the scalability of EPNs in multiple computing nodes.

Table 3.4: Illustration for scalability in context state

EPN Identifier	Function	Outcome
EPN_1	Process signals from oven	A
EPN_2 EPN_3 EPN_4	Process signals from baking dish Process signals from flour Process signals from sugar	B
EPN_5 EPN_6	Process signals from pear Process signals from baking dish	C
EPN_7 EPN_8	Process signals from vanilla pods Process signals from peeler	D
EPN_9 EPN_{10} EPN_{10}	Process signals from four Process signals from almond Process signals from sugar	E
EPN_{11}	Process signals from mixer	F
EPN_{12}	Process signals from mixer	G
EPN_{13}	Process signals from baking dish	H
EPN_{14}	Process signals from oven	I

For example, let us consider the system of two computing nodes implementing the EPNs defining the activity recognition. The output from the activity recognition is fed in to the system of EPNs illustrated in Table 3.4 as defined in the situated prompting in Figure 3.6.

The goal is to derive the outcome I. The system architecture needs to feed states derived from multiple EPNs in a feed forward loop, where the state from EPN_1 is fed to trigger prompts and initiates EPN_2, EPN_3 and EPN_4. The computing requires copies of EPNs and nine steps to derive the final goal.

Variations of the outcomes and processing of events are designed as an acyclic graph where the EPNs are the leaves and the goal is the root of the tree. This research intends to vary the number of EPNs and the distribution of the EPN between multiple computing nodes. For example, a larger system can be generated with 25 EPN where EPN_1 to EPN_{10} could be replicated in two computing nodes to cater for the needs of increased arrival of events. The goal to produce the outcome F is based on the output from the two stages, irrespective of the count of the EPNs. This example has been chosen to demonstrate the scenario for a typical class of challenges associated with distributed EPN placement. In situations of sharing the state between multiple EPNs algorithms have been designed to illustrate the EPN parallelisation and the computation of output.

3.5 Contributions

In this research, an application scenario from the ambient kitchen is decomposed in terms of analytic queries, encapsulated in multiple event processing networks. EPNs are designed to execute goal specific tasks and exchange information throughout the architecture in a tightly or loosely coupled manner. The workload in the event processing networks is listed below:

1. EPNs are designed to uniquely maintain their own reference data, input attributes and state. The streaming events entering the system are determined based on the count of event producers and event consumers. In the design presented here, few EPNs are designed as event producers, consumers or both. The scalability challenges of event consumers and producers addressed in §2.2.8,§2.2.7 are viewed generically in terms of the event arrival rate described in Chapter 6 . Few event consumers and producers are modeled to demonstrate the response centric scalability.

2. The responsiveness of the event processing, such as the count of operations and the frequency of execution, varies with the volume and the event processing. The queries are defined in the EPNs for subscription propagation, event filtering and routing. The complexity in operations as stated in §2.2.6 is defined using responsiveness and optimal placement of EPNs as outlined in Chapters 6 and 7.

3. The incoming events and outgoing events (actions) are sequentially fed or broadcast to the other event processing units during the data flow. The volume of events, correlation of intermediate results, raw events and the processed events are handled to meet the performance criteria. The challenges addressed in §2.2.9,§2.2.10 are addressed using splitting, processing and merging the results from multiple EPNs as described in Chapter 8.

Time critical processing of these event based applications requires detailed analysis in the performance metrics and forms the thrust of this research. The parameters define 'how well' the system performs and address the various optimization techniques related to the non-functional aspects of the system. Non-functional requirements are not related to the nature of the EPNs such as simple, complex or intelligent processing. Rather the system needs to measure *'how well'* the EPNs perform to deliver the overall system related performance targets irrespective of the nature of the event processing. Performance for an event processing architecture relates to an entire

event flow through multiple EPNs or just through a particular part of the EPN. Even though the level of the complexity between the EPNs varies from simple, complex or intelligent event processing, the key factors reliant on the performance of the EPNs can be summarized thus:

1. Response Time: Response time measures the timeliness or latency of the EPNs. The response time can be expressed in terms of determining an upper bound to process the events or an average desirable time for the EPNs to process the incoming events.

2. Predictability: Predictability and consistency is an important aspect of the EPN performance measurement. The performance of the EPN is measured in terms of the response time and the throughput. To maintain the consistency, the average response time or throughput needs to be consistently maintained. The predictability of the EPN response time plays a major role in maintaining consistent performance.

3. Placement of EPNs: The placement of EPNs on suitable computing nodes with sufficient hardware resources is critical for the delivery of desired performance. At run time, the efficient placement of the EPN needs estimation and prediction of the response time. The allocation priority for an EPN in a particular hardware needs to be determined based on the incoming arrival rate. The system may need to prioritize multiple or single EPNs based on the response time and throughput to ensure overall performance. Due to the practical constraints based on the dynamic arrival rate and the varying event processing requirements, few customised placements of the EPNs need to be undertaken based on the available hardware. The most beneficial and customized analysis linked to a specific EPN and arrival rate needs to be executed at frequent intervals.

4. Parallelisation: Along with the optimal placement of EPN, an architecture for parallelisation of EPN according to the logical and sequential flow is vital for response centric placements. Performance can be improved by careful placement of EPNs using vertical or horizontal parallelism. Horizontal parallelism can be implemented using execution of simultaneous multiple instances of EPNs. Vertical parallelism can be achieved by pipelining different stages of a sequence of the EPNs. A response centric architecture to implement the dynamic reconfiguration of the EPNs illustrating the vertical parallelism and inter-EPN parallelisation utilising the horizontal parallelisation is vital.

5. Scalability and Elasticity: Scalability is defined as the system's ability to gracefully provision resources and processing capability to adopt the processing requirements of the incoming event streams. Elasticity is the ability of the system to scale up or shrink the resources without modifying the system architecture. The ability of the system to handle the increase in the arrival rate of incoming events requires scalability and elasticity in both the hardware and the processing units. With the recent progress in cloud computing, an elastic and scalable computing infrastructure is available in a cost effective manner. In this research, the EPNs are defined as the generic processing template to receive, execute, process, forward or disseminate response centric results using scalable and elastic resources from cloud computing.

Chapter 4

Architecture

4.1 Introduction

Real time event processing is characterised by incoming events with varying incoming arrival rates. This needs formulation of long running queries (unlike *ad-hoc* queries) and needs repeated evaluation of the incoming events until the query is terminated. Along with the processing of incoming events, detection of events or conditions to fire a rule/action/trigger in a timely manner is essential to deliver functionalities dictated by the event processing. Many functionalities requires specific quality of service (QoS) requirements, which include response time, accuracy and throughput. These QoS requirements are highly dependent on each other (e.g. increase in latency decreases the throughput); choosing one can affect others dramatically. With the increase in the arrival rate of the incoming events, trade-offs in the QoS metrics need to be balanced by adding scalability aspects to the real time event processing. To achieve scalability matching the response time requirements for an event processing system, a judicious use of the available resources (e.g. CPU cycles, memory) to maximise the impact of the QoS metrics is critical. This research focuses on the key challenges in the response time centric scalability of the real time event processing to maintain low latency and to address the fluctuating arrival rate of the streaming events.

To achieve scalability and responsiveness in real time event processing applications, computation power forms the foundation. The ability to distribute the computations and the incoming events among multiple computing nodes to realize a cohesive output is critical to the system. To analyze and act on the varying incoming event arrival requires distribution of events in terms of parallelisation in multiple computing nodes for

achieving the unattainable performance and responsiveness using a single computing node.

The challenges encountered in distribution and processing of events in parallel are twofold, as listed below:

1. Focusing on improving the timeliness and accuracy of the results.
2. The timeliness of the results needs to be improved by latency and throughput triggered event distribution across various computing nodes.

The accuracy can be improved by careful splitting and processing of the events across multiple computing nodes. Unless the data integrity of the incoming events is carefully analyzed, the distribution of event streams will lead to an erroneous outcome or loss in accuracy.

The event distribution issues need to be approached in a use case centric approach where splitting of the event stream is undertaken as an operation without involving evaluation of complex predicates from multiple attributes in an event tuple. Examination of a few commonly encountered schemes for event distribution in terms of attributes can be used to *fix* loss of accuracy in the final outcome. The goal of this work is to address the non-functional aspects of real time event processing such as scalability and performance. The non-functional aspects are assessed on dimensions relevant to volume of events with varying numbers of input agents, producers, consumers, contexts, complexity of the computation and processing environment. While finding an optimal solution to the performance optimisation is NP-hard, the research approach taken here will be to analyze the possibility of parallelisation for the scalable and responsive real time event processing through hiring nodes from cloud computing.

4.2 Architecture

The architecture as illustrated in figure 4.1 harnesses a methodology to predict the response time of event processing and scale the resources hired from cloud computing through dynamic placement algorithms. The real time streaming events flow in to the system from various heterogeneous event sources. Low latency, throughput and the response time are key factors to process the streaming events. The design is based integrating independent event processing components and enabling the deployment of the event processing components in a rapidly scalable cloud computing infrastructure.

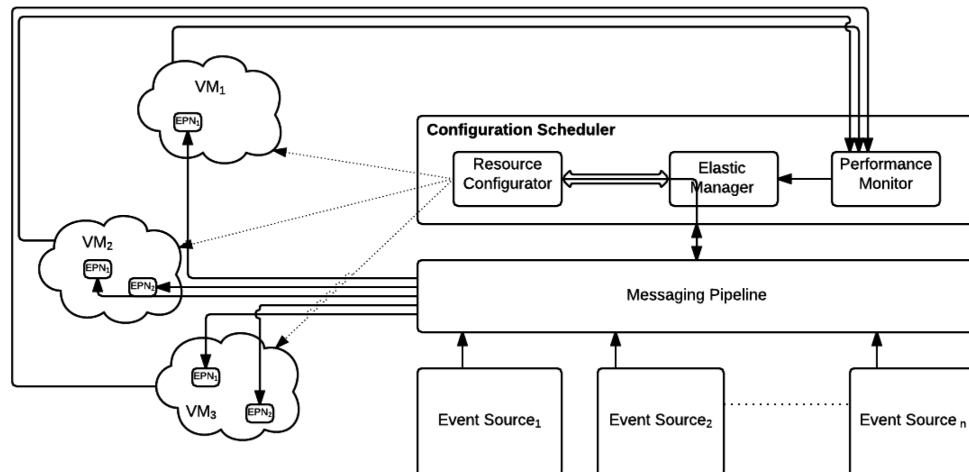


Figure 4.1: Architecture

Event processing components are built using complex event processing engines (CEP). Event processing networks named as EPNs were created using event processing engine called ESPER and utilised to capture, process and disseminate streaming events. Esper filters and analyses the incoming real time events using the Event Processing Language (EPL). It can be run in a single machine or VM containers, without any dependencies on external services. It does not require any particular threading model or external database. It works based on the event capture time and watermark-based time management.The EPNs process multiple sources of streaming events to identify meaningful real time decisions and alerts. Patterns are discovered from low-level through aggregation and merging of the event streams. The huge amount of initial stream of events is processed and reduced to meaningful events in several stages of event processing. EPNs are created to understand the event streams, relevant to the event processing and multiple stages of event transformation. EPNs model the relationship between events and derive knowledge from the events. The following tasks are accomplished by the EPNs

1. Define event sources and patterns based on the event sources, context and multiple stages of event processing.

2. Define high-level plan for the reactions and activities related to the events.

3. Monitor or follow up the entire cycle of events.

As a first step, semantics were defined to integrate the event sources and consumers. As a next step, the algorithms for the prediction of the EPN response time were undertaken. The response time prediction algorithms enable the creation of dynamic processes to adapt the EPNs based on the varying incoming arrival rate of the events. For this purpose, the virtual machines were dynamically hired from the cloud computing to deploy the EPNs. The placement algorithms define the deployment of the EPNs on the multiple virtual machines.

To provide the required level of abstraction on the deployment of EPNs in the cloud computing resources and the underlying technologies, a configuration scheduler is implemented to access the EPNs in a unified manner. The configuration scheduler as illustrated in Figure x provide the necessary event based communication with access to the event sources and virtual machines.

Figure 4.1 illustrate the system consisting of four major components namely event sources, messaging middleware, configuration scheduler and cloud computing hardware infrastructure. The events streams originating from the heterogeneous sources need to be gathered and streamed to various virtual machines registered in the system. Optimum amount of resources need to be estimated and hired to perform the extended analysis and processing of incoming events. The system is intended to provide a generic infrastructure to deliver responsive and scalable real time event processing. The architecture is designed to integrate the components within the system in a loosely coupled fashion.

To provide the infrastructure for processing massive amounts of real time streaming events, a staged approach is suited. The first processing stage involves gathering and pre-screening of raw events and directing the events to multiple EPNs possibly distributed in multiple virtual machines. The second processing stage is located in the EPNs deployed in virtual machines hired from cloud computing. The third stage involves the dissemination of the intelligence extracted from the raw events to multiple event consumers. Such a dynamic staged infrastructure reflects the principles of cloud computing. In this research, the first stage of streaming event emerged from the EPSRC funded, pervasive sensing test bed called ambient kitchen. The second stage was located in the cloud computing infrastructure, involving the continuous queries that were embedded within the EPNs to analyse the streaming events and to create situated prompting. The third stage involved dissemination of the intelligence to the systems embedded within the ambient kitchen. Due to the dynamic nature of the real time streaming events, the event processing must dynamically adapt to place the relevant EPNs in a specific virtual machine or should hire a new virtual machine to

satisfy the response time. The event processing infrastructure is intended to allow the addition or removal of virtual machines by distributing the EPNs. New virtual machines are hired in the system during the increase of arrival rate in incoming events and existing virtual machines are removed from the systems, when the arrival rate of the incoming events is reduced. In addition, monitors were deployed within the configuration to observe the response time of EPNs, arrival rate of the incoming events and processing capacity of the virtual machines. The algorithms to predict the response time and to place the EPNs are situated within the configuration scheduler. Based on the intelligence derived from the algorithms, virtual machines registered within the system are re-configured according to the incoming arrival rate of the streaming events and redirects the incoming events in accordance to the response time prediction.

The event processing system as a whole is distributed as units of EPNs across multiple virtual machines in the cloud. Each virtual machine can host one or more EPNs. The algorithms determine the count of EPNs on each virtual machine based on the response time prediction and incoming arrival rate of events.

Abstracting the knowledge gained through the development and deployment of the EPNs, this research focuses in the development of a generic real time event processing architecture to deploy the EPNs through dynamic resource provisioning, placement and response time prediction. The elastic manager, performance monitor and resource configurator are placed inside the configuration scheduler as illustrated in figure x. The algorithms to predict response time and reconfiguration of EPNs are hosted inside the performance monitor and elastic manager respectively. Throughout the processing stages, the performance monitor probes the metrics (CPU usage in virtual machines and arrival rate of incoming sources) of each virtual machine in an interface independent manner. The performance monitor maintains the interfaces that are designed to probe the event sources and virtual machines, specific to the hardware requirements. The resource configurator maintains the adapters to hire and release the virtual machines. Using the reconfiguration algorithms, the elastic manager, sends instructions to hire and release the virtual machines. Overall, the configuration scheduler deploys multiple instances of EPNs across multiple virtual machines to accomplish the scalable and responsive event processing. The scheduler associates the event sources to their respective EPNs through the messaging middleware. The asynchronous middleware forwards the events from multiple event sources to the EPNs. The event sources demonstrate dynamically changing arrival rate of

events from heterogeneous event sources. In this research, evaluation was done using an open source middleware called Apache active MQ. The messaging middleware forwards the events to EPNs and updates EPNs that are fed with results from the pre-processing of raw events or processed events from multiple EPNs. The middleware is designed to be accessible for all the event sources, event consumers, EPNs and configuration scheduler. This provides access to the processed events that are needed by multiple components within the architecture. Thus, mechanisms to distribute the EPNs to the desired parts of the event processing are specified and integrated within the architecture.

The EPNs are created as an independent task, which are focused towards the requirements of any application scenario. In this research, streaming events from ambient kitchen are used to define the functionalities of the EPNs, which are illustrated in chapter §3. The figure 4.2, illustrates sequence diagram which highlights the deployment of the EPNs in a generic manner with an objective to achieve scalability and responsiveness. Three major components such as event sources, VMs hired from cloud computing and configuration scheduler are involved in delivering the key tasks such as initial set up of virtual machines, hiring of the virtual machines and releasing of the virtual machines. During the initial setup, one instance of virtual machine is hired and EPNs are configured and deployed within the virtual machine by the resource scheduler. This minimum set up is maintained throughout the entire lifecycle of the system operation. The three major phases in the architecture are listed below:

1. During the initial setup phase, resource scheduler injects predefined scripts specified by the performance monitor in to each virtual machine. The monitoring scripts send updated metrics from each VM to the performance monitor.

2. In second phase, the events start arriving from the event sources. The resource scheduler redirects the events to an EPN. The performance monitor collects the metrics and predicts the response time for each EPN. The performance monitor passes the response time prediction, arrival rate and the virtual machine details to the elastic manager. The reconfiguration algorithm within the elastic manager provides the optimum placement plan for each EPN. When new event sources join the system or the arrival rate of the existing event source increases, the response time has an impact. The performance monitor predicts new response time and the elastic manager provides a plan to hire a new virtual machine and one more EPN is deployed in the existing virtual machine. This process continues until the optimum response time is achieved for all the EPNs deployed within the system.

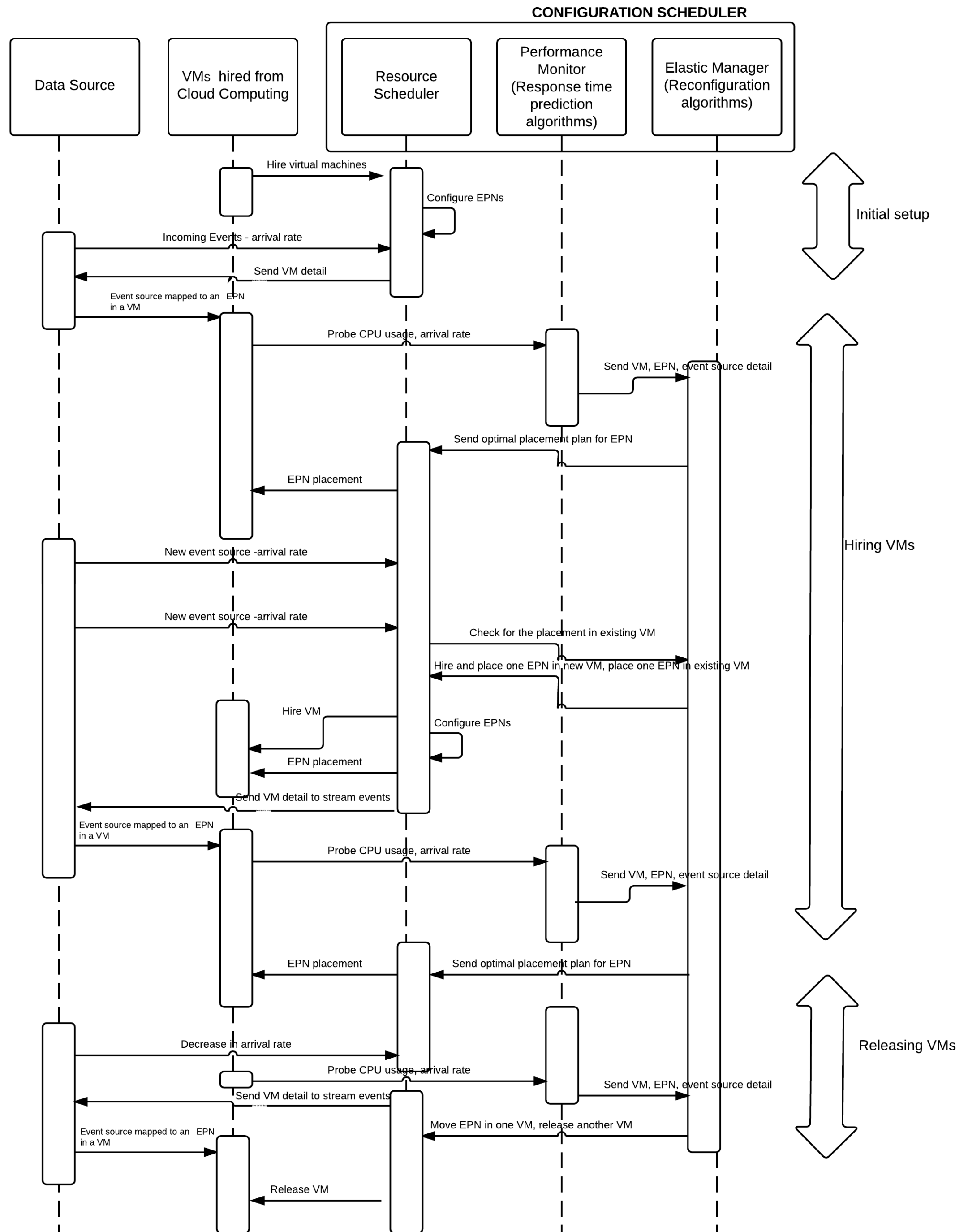


Figure 4.2: Sequence diagram illustrating the data flow in the system

3. In the third phase, when the event arrival rate decreases or few event sources stops sending events, few virtual machines are underutilised within the system. In this phase, the elastic manager predicts the new placement plan for the EPNs and releases few virtual machines.

4.3 System Description

The system processes several event streams, each emanating from a distinct source. These streams are denoted as $s_1, s_2, s_3, \ldots, s_\sigma$ and complete event streams in the system are defined as $\Sigma = s_1, s_2, \ldots, s_\sigma$. The system evaluates q queries, $Q_1, Q_2, \ldots, Q_q$. The state machine that implements the directed acyclic graph, DAG for Q_i is called the event processing network, EPN_i. Evaluating Q_i involves processing one or more event streams and the set of all streams input to EPN_i is denoted as S_i. Note that there is no implication that two EPNs have distinct S, e.g., S_i and S_j may overlap. Also, an input stream to EPN_i can be an output stream from another EPN_j; if so, $S_i \notin \Sigma$. If all inputs to EPN_i are output streams from other EPNs, then $S_i \cap \Sigma = \{\}$. An EPN is also associated with a performance target T. It is said to be distinct if any one of its three attributes is unique: DAG, S or T. All EPNs are considered to be distinct.

The system itself is made up of n virtual machines, denoted here generically as *nodes*, drawn from a cloud computing platform. The number of nodes used, n, is increased (or decreased) when the load increases (or decreases) to an extent that the current configuration over these n nodes is deemed inadequate (or more than strictly necessary, respectively) to meet the performance targets.

A configuration is a mapping from the set of EPNs onto the set of hosts. Figure 4.3 shows a configuration where, EPN_1, EPN_2 and EPN_3 are mapped to (e.g. hosted by) node 1, and the rest of the EPNs are mapped to a distinct node. The system has a configuration scheduler, CS for short, which decides the configuration appropriate to the load conditions and performance targets associated with the EPNs. For brevity, we assume that CS is centralized, hosted on a single node.

4.4 Problem Specification

Event processing is characterized by the continuous processing of streamed data tuples or events in order to evaluate the queries deployed by decision support systems in a timely manner. Event sources can, for example, be pervasive sensors; while the number of sources is normally fixed in an application, the rates at which they generate events can vary widely and often unpredictably, driven purely by the external processes they monitor. Similarly, the number of queries that need to be evaluated over the streams can also vary over time.A fixed number of event sources is registered in the system to validate the accuracy of the response time estimation. Thus, an event processing system with real-time performance requirements must meet targeted response times despite being subjected to these two types of varying loads.

A query evaluation can be modeled as a directed acyclic graph wherein nodes are operators and the links are event streams that are either raw or partially processed by the preceding operators. Early commercial systems analysed in §2.2 used single server solutions and proposed a variety of techniques, such as multi-query optimization, for response-time optimization. Later, distributed solutions §2.5 handled the optimization problem as a load-balancing issue over a fixed set of nodes: moving query operators to nodes where their resource requirements are best met and thereby achieving the best overall response time. Such solutions however have two drawbacks: they require the placement of low-level probes to measure operator execution rates, queue lengths, etc., making their implementation hard and possibly not portable across heterogeneous machines; at times, the load has to be *shed* to meet response time targets. In this research, a novel approach is presented and investigated to responsive and scalable event-processing which avoids both these drawbacks; it leverages the advantages offered by, and is best suited for implementation in, cloud computing platforms.

As a part of the approach, a model for the event processing activity is created from a performance analysis perspective. The rationale behind the model can be succinctly explained as below. Incoming tuples in an event processing engine go through a sequence of operators before triggering an output event. The output latency or the response time therefore consists of three major components:

1. The wait time before encountering the first operator in the sequence,
2. The wait time between operators, and
3. The sum of operator execution times.

When the arrival rate of tuples increases, wait time (1) is seriously affected. When more engines are hosted by a single computing node, inter-operator delay (2) and operator execution times (3) are impacted due to competition for CPU usage. Based on these observations, an event processing engine was modeled as a single queue *server* system wherein the server is the composite operator consisting of all operators within that engine. The waiting time in the queue models (1) and the *processing time* by the *server* models the sum of (2) and (3). Queuing theory is used to predict queuing time (1). Off-line calibration is used to establish inter-operator delay (2) and (3) as the server processing time. Note that (2) and (3) are less affected by variation in arrival rates.

4.5 Event Processing Architecture

The front end of the system has a scheduler as illustrated in Figure 4.3, to which all the event sources are directed to a specific computing node in the cloud according to the policy in the configuration scheduler. The event sources might be included into the system through MOM (message oriented middleware) or asynchronously connected through a socket service. When a scheduler announces the EPN-host mapping, each node subscribes to relevant input streams and transmits its relevant output streams, if any, to nodes of EPN which use them as inputs. In Figure 4.3, nodes 4 and 5 supply their relevant outputs to node 2. All other output streams are archived.

Central to the architecture is the configuration scheduler CS. In a nutshell, each EPN takes macro-level measurements of its own performance and reports periodically to CS which constructs a global view and attempts to re-map EPN to host nodes, if response times of some EPN are either above or far below their target levels; in the latter, new nodes may have to be brought in and in the former case some of the existing nodes may be released. Note that re-mapping EPN requires support mechanisms and extracts a cost, both of which are considered for future work.

4.5.1 Configuration Scheduler

Each node monitors every EPN_i deployed within it. The response time RT_i and the arrival rates of each input stream; the maximum RT_i and the sum (AR_i) of the average arrival rates of all its input streams observed over the reporting interval are sent to

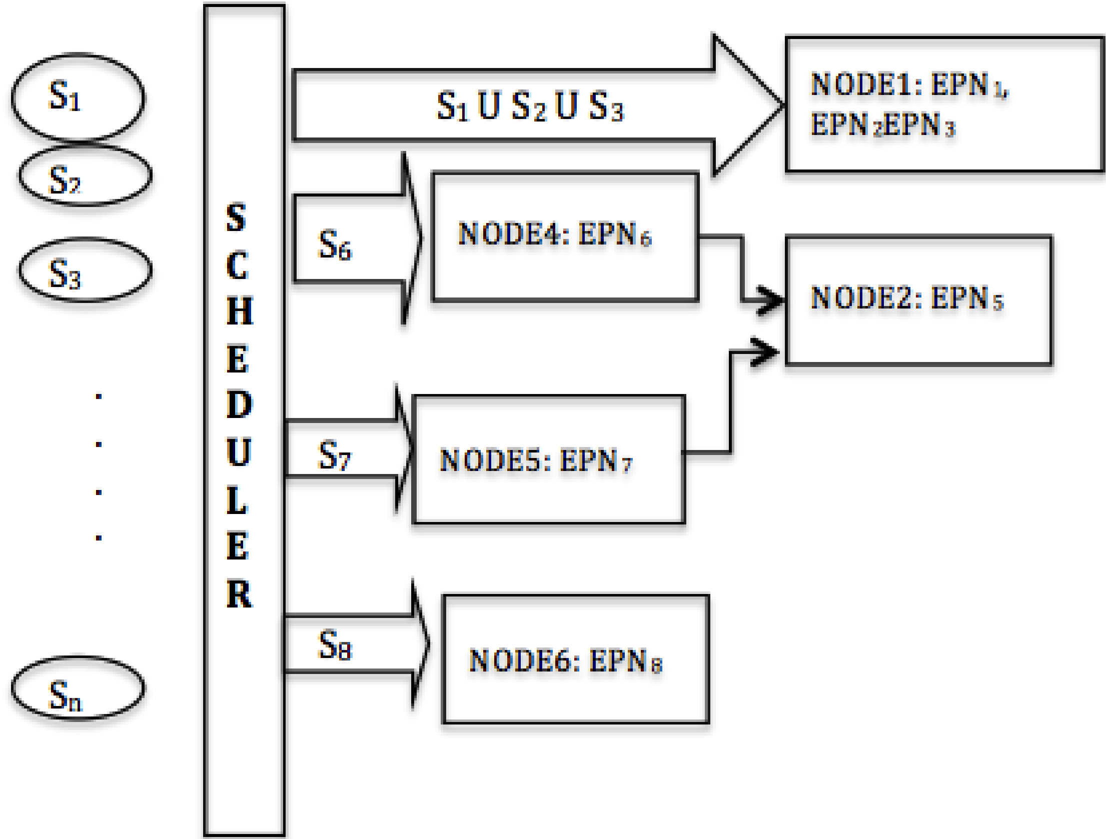


FIGURE 4.3: The Event Processing Architecture

the CS. For example, node 1 in Figure 4.3 that hosts EPN_1, EPN_2 and EPN_3, will report to CS $\{RT_1, AR_1\}$, $\{RT_2, AR_2\}$ and $\{RT_3, AR_3\}$.

Let T_i denote the average response-time target for EPN_i. It is defined as

$$\delta_i = \frac{RT_i - T_i}{T_i} \tag{4.1}$$

where δ_i is the measurable *relative deviation* of RT_i

The lower and upper bounds is defined as δ_i^m: *low water-mark*, denoted as LW . If $LW \leq \delta_i^m$, then EPN_i is deemed to meet its performance target less than the scope of chosen LW .

If all q EPNs in the system are deemed to meet their respective targets, then the current configuration is considered to be working well and CS does nothing; otherwise, CS decides on a new configuration by dividing q EPNs into ζ disjoint sets, $Z_1, Z_2, \ldots, Z_\zeta$,

and by ensuring that the following two constraints are met when all EPNs of every given Z_x, $1 \leq x \leq \zeta$, are hosted within a unique node:

1. ζ is the smallest possible, i.e., the number of nodes used in the new configuration is minimum when all EPNs in a given Z_x, are hosted within a distinct node, and
2. For every EPN in Z_x
   (a) Each EPN in Z_x meets its target response time, and
   (b) The total load exerted by all EPNs in Z_x does not exceed the node's capacity.

Total load is calculated using the queuing theory. During the calibration of the EPNs various arrival rates are fed into the EPNs and the load exerted by each computing node is measured.These two constraints make the new configuration decided by the CS an optimal one. Meeting the first constraint becomes one of *optimal assignment* problem, provided that 2a and 2b can be analytically evaluated (as either true or false) for any given Z_x. Note that the optimal assignment problem is not NP-complete for two reasons: ζ is not fixed and q is finite. It is solved here using a bin-packing algorithm which *packs* the EPNs into the smallest number of nodes (bins), subject to conditions 2a and 2b.

The analytical evaluation of 2a and 2b requires deriving formulae for analytically estimating EPN response times, which in turn makes a simplifying assumption that a node can host any EPN on its own and satisfy both 2a and 2b.

4.5.2 Inter-EPN Parallelism

It is possible that for a situation EPN_i does not meet its target T_i as stated in 2a even though its host node hosts no other EPN. It can occur, for example, if AR_i is very large. On these occasions, Inter-EPN parallelism as depicted in Figure 4.4 EPN_i is hosted on multiple nodes (two nodes in Figure 4.4) and each input stream in S_i is temporally split and distinct (and temporally disjoint) splits are input to distinct hosts. For example, an input stream s_i can be split as: (t to t+100) tuples as s_i^1, (t+101 to t+200) tuples as s_i^2, (t+201 to t+300) tuples as s_i^3, and so on. The splits s_i^1, s_i^3, s_i^5 are sent to EPN_i^1 (in that order) and the rest to EPN_i^2, halving the arrival at each destination. The results from EPN_i^1 and EPN_i^2 are to be *reduced* to the final version.

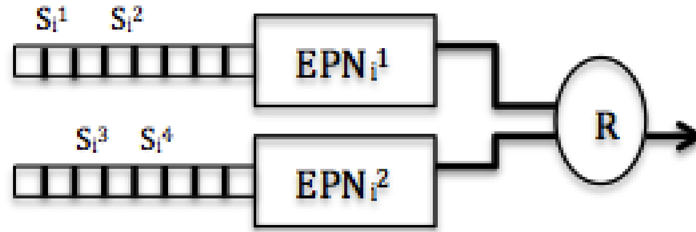


FIGURE 4.4: Inter-EPN parallelism

The approach of Inter-EPN parallelism corresponds to the well-known MapReduce paradigm. In the context of parallelism, a wide range of intra-operator (or partitioned) parallelism is implemented as described in §2.5. The existing solutions for intra-operator parallelism use multi-query optimization by scheduling the incoming workload in a fixed number of nodes. Operators are shared by several queries based on a dynamic data scheme to maximize resource utilization. Maximum predicted arrival rate influences and determines the count of EPNs to make the inter-EPN parallelism.

4.6 Evaluation

In this work an exemplar from the SiDE [SiDE] project is the focus to evaluate the response time estimation and inter-EPN parallelisation. Exemplars from the connected community theme are used to evaluate the problem and to test the solutions.

The ambient kitchen is created with a vision of ubiquitous computing where sensors are woven in to the kitchen surroundings. The kitchen utilises RFID readers and tags on all movable non-metallic objects, IP cameras, wireless accelerometers, pressure sensitive floors, projectors and speakers to prompt the users. Ambient kitchen is constructed in the university research laboratory to emulate a pervasive computing prototyping environment which is used to carry out research in machine learning, activity recognition, distributed event processing etc., to aid research in social inclusion.

The technical basis of the infrastructure is to support people with cognitive impairments. An investigation of food preparation actions was carried using embedded three axis accelerometers in the kitchen utensils used by volunteers. The analysis of low level food preparation activity using continuous streaming data from the kitchen

utensils included identifying activities such as cutting, chopping, coring, stirring, slicing, dicing, eating, peeling, spreading, scooping, scraping and shaving [Thomaz et al. (2011)].

Slice and Dice [Pham and Olivier (2009)] is a first step in the development of a low level activity recognition framework for food preparation. The human volunteers were asked to perform tasks such as salad preparation and sandwich preparation with the tagged ingredients. Their actions were recorded by video cameras and annotated manually in a temporal window. Based on the actions in the ambient kitchen, the activities were classified using algorithms.

Streaming events generated from the ambient kitchen are used to detect the presence of the person, actions and behaviour pattern of the kitchen occupants. One of the challenges in this environment is the integration of new devices in to the physical kitchen and standardisation of the widely diverse event processing life cycles. The heterogeneous environment triggers a need for the event processing stack with self-managing properties since solutions depending on the centralised management, configuration and adaptation exhibit a single point of failure. Varying periods of activity and inactivity prevail in the kitchen leading to fluctuating event generation. The research motivation is to provide response centric and scalable event based infrastructure to seamlessly integrate context aware services, independent of devices and subjects to create a high level situated prompting environment.

Chapter 5

Algorithms for Responsiveness

5.1 Introduction

The aim of this chapter is to develop generic algorithms that allow event processing networks to scale across multiple computing nodes. Given a range of computing resources available through utility computing, the response time prediction is critical for the deployment of the event processing networks in a stated hardware. The deployment of the EPNs needs accurate and efficient evaluation of the computing resources and performance to provide scalable and adaptive solutions. This chapter will evaluate the optimal algorithms to place the event processing networks in multiple computing nodes by maintaining an ideal response time. Generic algorithms for the estimation of the response time, optimal configuration selection, chaining and parallelisation of the EPNs are presented.

5.2 Analytical Estimation of EPN Response Times and CPU Usage

Analytical estimations make a simplifying assumption that a node can host any single EPN on its own *and* satisfy both 2a and 2b. If this assumption does not hold, then inter-EPN parallelism, similar to intra-operator parallelism used in Gulisano et al. (2010), would be necessary. This is an online model where few of the parameters(arrival rates) are dependent on dynamic real time values from the system. More detailed remarks are presented at the end of this section.

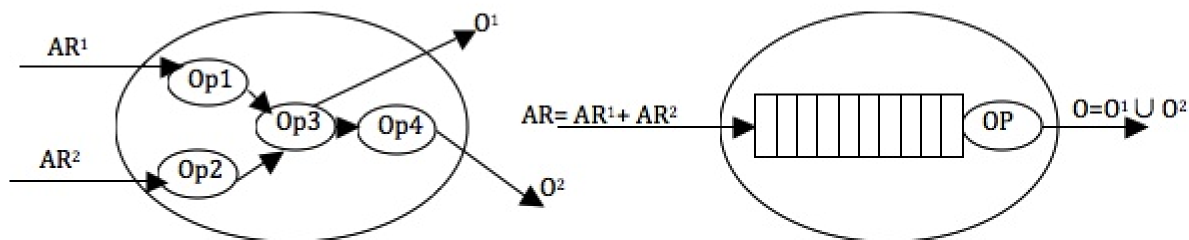


FIGURE 5.1: (a) Internals of an EPN. (b) 1-server, 1-queue model of the EPN

Consider a node hosting the EPNs of some $Z_x = \{EPN_1, EPN_2, \ldots, EPN_k\}$. Estimations for this scenario are done in two parts as listed below:

1. Modelling a single $EPN_i \in Z_x$ as a single-queue, 1-server system.
2. Modelling all EPNs of Z_x as a M/G/1 multi-class queuing system.

5.2.1 Modelling and Calibrating a Single EPN

Figure 5.1(a) presents the DAG (directed acyclic graph) structure of a simple EPN chosen as an example. There are two input streams with arrival rates AR^1 and AR^2. After they are acted on by distinct operators (Op1 and Op2), they are joined at Op3 which produces an output stream O^1 to the environment and an input stream to Op4 which then generates O^2. Note that each Op would have its own buffer to store the incoming events which are not shown in Fig 5.1(a).

Irrespective of its DAG structure, the EPN of Fig 5.1(a) is modeled as a single server system that receives a single input stream with arrival rate $AR = AR^1 \cup AR^2$, and generates a single output stream O $= O^1 \cup O^2$; the incoming event tuples are queued and the tuple at the head of the queue is processed by a super operator OP composed of Op1, Op2, Op3, Op4 (see Fig 5.1(b)). The tuples that get past the head of the queue, in the model, are *processed* by OP as per the logic of the EPN of Fig 5.1(a) and generate O $= O^1 \cup O^2$.

The above modelling approach is applied to estimate the average response time of, and the CPU load exerted by, any single EPN. First, a few metrics are defined or recalled, some of which are measured dynamically and the rest established through off-line calibration. Let EPN_i be any EPN in the system.

Event Arrivals denote the arrivals of events of streams in S_i and are taken to be Poisson at the rate of AR_i that is supplied to CS at the end of each reporting interval.

Processing Time denotes the total processing time a tuple or a window of tuples needs to undergo to produce an output $O^j \in O$ after a tuple in the window has just gone past the head of the queue as in Figure 5.1(b). Note that it does not include the time that a tuple spends between its arrival and reaching the head of the queue *viz.* the *queuing delays*. Moreover, the processing time depends on the nature of DAG_i that EPN_i implements and also on the particular path that a tuple takes within DAG_i for it to be processed and an appropriate output to be generated.

Given the non-deterministic nature of tuple-flows, the processing time is modeled as a random variable of some unknown distribution. When EPN_i generates several outputs, the one that takes the maximum processing time will be of interest. For this output, b_i is defined as the *average processing time* and $M_{2,i}$ as the *second moment* of the processing times.

The *Processing Load* imposed by EPN_i on the host node is ρ_i. Specifically, ρ_i is the fraction of time EPN_i uses the node's CPU. Thus, $\rho_i = AR_i \times b_i$ and this relation is used to establish $M_{2,i}$ and b_i through *calibration* as described below.

EPN_i is hosted on its own on a node and subject to *small* arrival rates AR_i (to eliminate or at least minimize queuing delays); the CPU usage for each rate is observed and the resulting processing times are also computed from the observations. From these times, the most processing intensive output is identified, and $M_{2,i}$ and b_i are established. EPNs typically process input events in groups or windows of, say, w tuples; if so, the arrival rate during calibration should not exceed w events per second.

5.2.2 Modelling Multiple EPNs in a Single Host

Recall that a single node hosts k EPNs of Z_x. The collection of EPNs is abstracted in the same way as the collection of operators of a single EPN; more precisely, the k EPNs of Z_x are replaced by a *composite* EPN and the input events of various EPNs are placed in a single FIFO queue; when an event or a window of appropriate events gets past the head of the queue, the (virtual) processor of the node processes it by executing the appropriate EPN_i. The model thus becomes a M/G/1 multiple class queuing system [Gelenbe and Mitrani (2010)].

Using the well-known M/G/1 results, the expressions are derived for metrics of interest, when a node hosts k EPNs of $Z_x = \{EPN_1, EPN_2, \ldots, EPN_k\}$. Note that the total arrival rate (AR) at the node is the sum of the arrival rates of the k hosted EPNs ($\sum_{i=1}^{k} AR_i$); similarly, the total load (ρ) on the node is the sum of the load exerted by every $EPN_i \in Z_x$. Thus,

$$AR = \sum_{i=1}^{k} AR_i, \quad \rho = \sum_{i=1}^{k} \rho_i = \sum_{i=1}^{k} AR_i \times b_i \tag{5.1}$$

The average response time estimated for any $EPN_i \in Z_x$ as W_i:

$$W_i = \frac{(AR \times M2)}{2(1-\rho)} + b_i, \tag{5.2}$$

where $M2 = \frac{1}{AR} \sum_{i=1}^{k} M_{2,i} \times AR_i$. Similar to δ_i^m defined by expression 7.1. The δ_i^e denote the *relative deviation* of W_i, with e indicating that δ_i^e is based on estimation.

$$\delta_i^e = \frac{W_i - T_i}{T_i} \tag{5.3}$$

If W_i estimates RT_i reasonably accurately, $\delta_i^e \approx \delta_i^m$, and 2(a) and 2(b) of §5.2 can be evaluated; more precisely, given that the EPNs are divided into ζ disjoint sets, $Z_1, Z_2, \ldots, Z_\zeta$, using a unique node to host each Z_x, $1 \le x \le \zeta$, leads to a viable configuration, if

$$\forall Z_x, 1 \le x \le \zeta : \forall EPN_i \in Z_x : LW \le \delta_i^e, \; \textit{and} \tag{5.4}$$

$$\forall Z_x, 1 \le x \le \zeta : \rho \le \rho_{th} \tag{5.5}$$

Remarks. Condition 5.4 states that every EPN in every Z_x must have its estimated deviation within the water marks and Condition 5.5 requires that the total load imposed on every Z_x must not exceed ρ_{th}. They verify 2(a) and 2(b) of Section 4 respectively, provided that each node being used for hosting a set of EPNs is (i) a single CPU or a single core machine, and has a CPU that is as powerful as the CPU of the one used for the calibration of EPNs.

If the CPUs of the host nodes are more powerful, the actual response times would be smaller than the estimates and hence the evaluation of Conditions 5.4 and 5.5 would be pessimistic and would result in more nodes being used than strictly necessary. On the other hand, if the calibration has been done using nodes with more powerful CPUs, response-time targets may be missed frequently.

In inter-EPN parallelism, a given EPN_i is hosted on multiple nodes and each input stream in S_i is temporally split and each split is input to one of the hosting nodes. For example, let EPN_i be hosted by two nodes as EPN_i^1 and EPN_i^2; an input stream $s_i \in S_i$ can be split as: first 100 tuples (1 to 100) as s_i^1, the next 100 tuples (101 to 200) as s_i^2, (201 to 300) as s_i^3, and so on. Odd splits, (s_i^1, s_i^3, ..., s_i^{2n-1}, ...), are sent to EPN_i^1 (in order) and even splits to EPN_i^2, thus halving the arrival at each node. The results from EPN_i^1 and EPN_i^2 would have to be 'reduced' for the final result.

5.3 Optimal Configuration Selection

Recall that each EPN_i in the system §5.2, is represented within CS by five parameters: the observed response time RT_i (as periodically reported to CS), the target response time T_i (given), the observed total arrival rate AR_i (reported to CS), the processing time b_i (calibrated) and an estimated response time W_i. When CS observes that one or more EPNs have their $\frac{(RT_i - T_i)}{T_i}$ falling below the low water mark, it seeks a different, optimal configuration. This selection process itself does not require halting of any EPNs and can proceed in parallel; however, once a different optimal configuration is found, implementing the latter requires moving some EPNs to different nodes and re-directing event streams to appropriate nodes; the latter may involve duplicating a stream in the extreme. What follows describes an algorithm for determining an optimal configuration and ignores the cost of implementing this configuration which would be a topic of our future research. A higher level decision system may find the reconfiguration cost excessive in relation to the performance benefits and decide to stick with the current configuration for a few more reporting intervals. The assumption is made that infinite number of commuting nodes are available to hire.

The reconfiguration algorithm associates each working node Z_x with a list E_i of all EPNs hosted in N_i. An EPN with $\frac{(RT-T)}{T}$ exceeding the low water mark is marked as donor and the node is entered in the donor list. EPNs with $\frac{(RT-T)}{T}$ falling below the low water mark is marked as an acceptor and the node is added to the acceptor list. If the node consists of one or more donor EPN, it is entered in the donor list.

The nodes consisting of acceptor EPNs are entered into the acceptor list. In the new configuration, an acceptor may take in more processing load and a donor must rid itself of some or all of the under-performing EPNs which it is currently hosting. (An EPN under-performs when its $\frac{(RT-T)}{T}$ exceeds the high water mark and is the most under-performing one if its $\frac{(RT-T)}{T}$ is the largest among the EPNs co-hosted with it.) The algorithm has two phases.

The purpose of the first phase is to empty the donor list and is skipped if the list is empty to start with. The donor list is always kept arranged in the non-increasing order of the total CPU load (ρ) that the EPNs impose on the donor nodes. The acceptor list, on the other hand, is always kept arranged in the non-decreasing order of the node's total CPU load. Further, the list E of EPNs for each donor is kept arranged in the non-increasing order of their $\frac{(RT-T)}{T}$. Thus, the first node in the donor list, say node D, is always the most heavily loaded, the first node in the acceptor list is always the least loaded, and the first EPN in D's E list, E_D, is the most under-performing EPN in node D. This EPN ought to be moved out of D and is denoted as EPN_D.

Starting with the first node in the acceptor list, each acceptor node A is tried by checking if $E_A \cup EPN_D$ satisfies the constraints 2(a) and 2(b) by using (5) and (6). The first acceptor found to meet these constraints becomes the new host for EPN_D. If no node in the acceptor list is found to satisfy these constraints, a new node is hired from the cloud to host EPN_D.

Let the chosen host for EPN_D be denoted as the candidate node N_C. In the new configuration, E_D is set to $E_D - EPN_D$ and E_C to $E_C \cup EPN_D$. (Note that if N_C is to be hired fresh, its E_C should be initialized to $\{\}$). N_C is entered into the acceptor list, if it is to be a newly hired node. Using (5) and (6), the donor status of node D is reassessed and D is either retained in the donor list or moved to the acceptor list accordingly. Note that a node in the acceptor list cannot leave the list during the first phase.

Nodes D and N_C giving up and gaining EPN_D respectively call for re-arranging the lists and identifying new D and EPN_D. This process of finding a new host for EPN_D is repeated so long as an EPN_D exists, i.e., until the donor list becomes empty.

The purpose of the second phase is to attempt to reduce the number of acceptor nodes, when the latter is more than one. In this phase, a list is created as a copy of the original without the first node (i.e. the least loaded one). The first node is treated as a donor node D whose EPNs are all considered as under-performing ones. The algorithm of the first phase is repeated to move the EPN_D of the D node to one of

the acceptor nodes in the list. However, if EPN_D cannot be moved to any of the nodes in the list, then a new node is not hired but the attempt is considered to have failed. On the other hand, if all EPNs of D can be moved using only the nodes in the list, then the latter becomes the new acceptor list . The process is repeated until a failure is encountered or the list is empty. The acceptor list and the E lists of the nodes in it provide the new optimal configuration. Implementing the new configuration would involve hiring new nodes (if at all), moving some EPNs and redirecting input streams.

This algorithm's rules are presented in the pseudocode shown in Algorithm 1.

The EPNs require sophisticated handling of processing state. No faults are being considered in the underlying infrastructure.Tracking of the states across the nodes requires aggregates to be maintained for an unbounded time window. This is challenging due to the ever-growing state to be stored. Performance of computation on this state is likely to degrade overtime. Real-time event processing requires global aggregation over all nodes, which limits the options to distribute processing. The configuration selection algorithm assumes a stateless condition. During the inter-EPN parallelisation, the states are discussed in detail using algorithms specifically designed to handle the states between the nodes. Chapter 8 describes the algorithms. When the load exceeds a pre-determined threshold, the node is declared as the donor node. The threshold is calculated in the calibration process.

5.4 InterEPN Parallelism

In diverse application contexts, to improve the scalability of the event processing networks, distributed processing of incoming event streams in multiple computing nodes is used. The approach will be to place the continuous queries for processing of the named events and aggregate relevant response from multiple event processing networks. The mechanism for the distributed processing of events in multiple nodes involves forwarding the partial results through the network in a directed manner. The intermediate nodes handling the real time events partially process the events and once the query is fully resolved, a completed response is sent back to the querying node.

We propose algorithms to process the real time events by preserving the syntax and semantics of the distributed outcome of the query. The challenges in the solution to obtain higher scalability would result in output that does not match with the centralized output. Consider the example where chopping, pouring, switching on the cooker and stirring of food item arising from multiple sensors, processing under

Algorithm 1 Configuration Selection algorithm main()

Require: $EPN_i \leftarrow$ List of EPNs in the system
Require: $RT_i \leftarrow$ Observed Response Time
Require: $T_i \leftarrow$ Target Response Time
Require: $AR_i \leftarrow$ Arrival Rate
Require: $W_i \leftarrow$ Estimated Response Time
Require: $b_i \leftarrow$ Calibrated Processing Time
Require: $E_C = Z_x$ EPN list in a node
Require: E_D Donor EPN list to be marked with host node to move
Require: EPN_D Donor EPN to move
Require: N_C Candidate node to move
Require: E donor list
Require: A acceptor list
Require: D_{ps} pseudo-donor list to store underperforming EPN, subset of highly loaded EPN in Acceptor list
Require: A_{da} deutero-acceptor list i.e., least loaded node

1: **repeat**
2: **if** Node contains $EPN_i \leftarrow$ with *low water-mark* $\geq \frac{(RT_i - T_i)}{T_i}$ **then**
3: Mark a EPN in EPN_D
4: add node to D (donor list)
5: sort D in descending order of total cpu load (ρ)
6: **end if**
7: **if** Node contains $EPN_i \leftarrow$ with *low water-mark* $\leq \frac{(RT_i - T_i)}{T_i}$ **then**
8: Mark a EPN in EPN_A
9: add node to A (Acceptor list)
10: sort A in ascending order of total cpu load (ρ)
11: **end if**
12: **repeat**
13: virtually place first EPN from EPN_D in first node of A
14: find the total cpu load (ρ) after the virtual placement of EPN_D
15: **if** totalload ≤ 0.8 **then**
16: mark new host for EPN_D as the candidate node N_c
17: $E_D = E_D - EPN_D$
18: $E_C \cup EPN_D$
19: **end if**
20: **if** N_C is hired as new node from cloud **then**
21: set $E_C = \{\}$
22: $N_C \cup A$
23: **end if**
24: **until** donor list D is empty
25: **until** *low water-mark* $\geq \frac{(RT_i - T_i)}{T_i} \leq$ *low water-mark*

10 different computing nodes should send a prompt called 'switch off cooker' after 20 minutes. In a centralized processing scenario, each state of the system is stored and ordered temporally, characterizing the progression of the state from one level to the next level resulting in a final prompt. To aggregate the same query processed in multiple instances require a parallelization strategy and the granulation of the parallelization of the data and the aggregation of the data should be defined.

The logic behind the parallelization of the continuous query involves broadcasting the data to some significant part of the network. Mechanisms are devised to split the data into multiple physical data streams provisioned and transferred in to the network to finally reach the target node to get the final results. A query is a directed acyclic graph where the node is an operator and the edges pave the way to the data flow.

In Chapter 8 various algorithms on parallelizing and chaining of EPNs will be analyzed along with the overhead associated with each of the processing scenarios.

Chapter 6

Evaluation 1: Response Time Estimation

6.1 Introduction

To address the responsiveness, a generic approach is pursued where processing of the event streams is undertaken by an event processing network without involving evaluation of complex predicates from multiple attributes in an event tuple. Uniform, increasing and bursty event workloads are feasible options in information management for the real time event processing scenarios. In uniform workload, a steady stream of events arrives with an average system load. In an increasing workload the intensity increases, however during every level of intensity shift, the average event arrival increases proportionately. In a bursty workload, spikes of intense activity appear in an unpredictable fashion leading to system instability. The only way to handle bursty workload is to allocate excess resources to manage the spikes of workload. Intentionally, the spiky event workload is not considered in this research. The uniform and increasing workload uses a poisson arrival distribution with a predictable average system load. Based on these factors, poisson arrival rate is assumed to predict the response time. Few schemes for event distribution are examined to 'fix' loss of accuracy in final outcome, scalability and performance in terms of response time optimisation. The responsiveness of the application or the event processing networks are assessed based on dimensions relevant to the volume of events with varying number of event producers, consumers, complexity of the computation and processing environment.

6.1.1 Response Time Measurement

Analytical estimations make an assumption that each node can host any single EPN on its own and satisfy the response time requirements. Let us consider a scenario where the node hosts EPNs of some $Z_x = \{EPN_1, EPN_2,...,EPN_k\}$. Estimations for this scenario are done in two parts:

1. Modelling a single $EPN_i \epsilon Z_x$ as a single server queue
2. Modelling all EPNs of Z_x as a M/G/1 multi-class queuing system

An event processing network (EPN) is composed of multiple event processing elements (PE) or agents, specific to the activity recognition as illustrated in Figure 3.3. This EPN acts as the fundamental processing agent to evaluate response time. The individual event processing elements are composed of queries encapsulating statistical functions such as 3.4, 3.3, 3.2, 3.1 implemented in a modular approach tied together to identify the activities.

The range of scenarios is very broad in the ambient kitchen and presents varying operational criteria in terms of event types. The three different event processing elements involved in the activity recognition – namely windowing, feature extraction and pattern matching – are labelled as distinct measurement points. The 'arrival rate' is measured as the quantity of input events received from heterogeneous sensors placed in the ambient kitchen within a given time period. This experiment measures arrival rate as the quantity of events generated by the event producers every second. The throughput is measured as the quantity of computed/derived events (i.e. activities recognised in the ambient kitchen). The throughput measurements in this experiment are performed every second.

The following micro-benchmarks are used in the performance evaluation targeted towards response time estimation for the events generated from the ambient kitchen.

- Identify the deviation between arrival rate and throughput of the activities recognised from the event processing infrastructure.
- Identify and measure the throughput for variable event arrival rate.
- Identify and measure the latency for variable event arrival rate.

- Identify the percentage of deviation in measured throughput and predicted throughput due to the increase in arrival rate of the events.
- Identify the non-functional requirements on the system performance based on the optimisation.
- Identify the scalability criteria to add additional computing node or processing power to satisfy the response time due to the increase in arrival rate.
- Identify a threshold in a temporal window based on the scalability criteria to deploy new processing or computing nodes.

The events from various sensors in the ambient kitchen experiments were used to generate event streams and fed in to the processing engine in Figure 3.3 . The rate of events fed in to the machine was varied to understand the criteria listed in the above micro-benchmark. More event sources were added to increase the rate of events. The number of activities generated every second from the system is measured as throughput.

Multiple utility classes were designed to represent each device such as tablespoon, teaspoon, fork, knife for eating, knife for chopping, floor boards, cupboard doors and so on. Accelerometers were embedded within the devices as shown in Figure 3.3.

Each device is programmed as a utility class in the system. There can be several objects of a given utility class. Thus, each of the input streams, $s_1, s_2, s_3, .., s_\sigma$ (refer to Section 4.3), emanates from a distinct object in the ambient kitchen and σ can be around 600. In this research, the response time estimation is evaluated using the streaming events from the ambient kitchen. The count of event producers was bounded to an approximate count of 600.

Each event producer was designed to generate the streams of events denoting event arrival. The streaming events were classified in to windows. The experiment used a predetermined window of computation based on the order of arrival of the incoming event streams. As an experiment objective, the machine learning algorithms for activity recognition dictates retaining the recently arrived 64 events of an event stream in one computational window. The window size of 64 is determined as the optimal value to identify the activity (cutting, chopping, etc.) undertaken by the object. The system enters all the arriving events based on the sequence of arrival into a window. When the window is full, the oldest 32 events are pushed out of the window. The

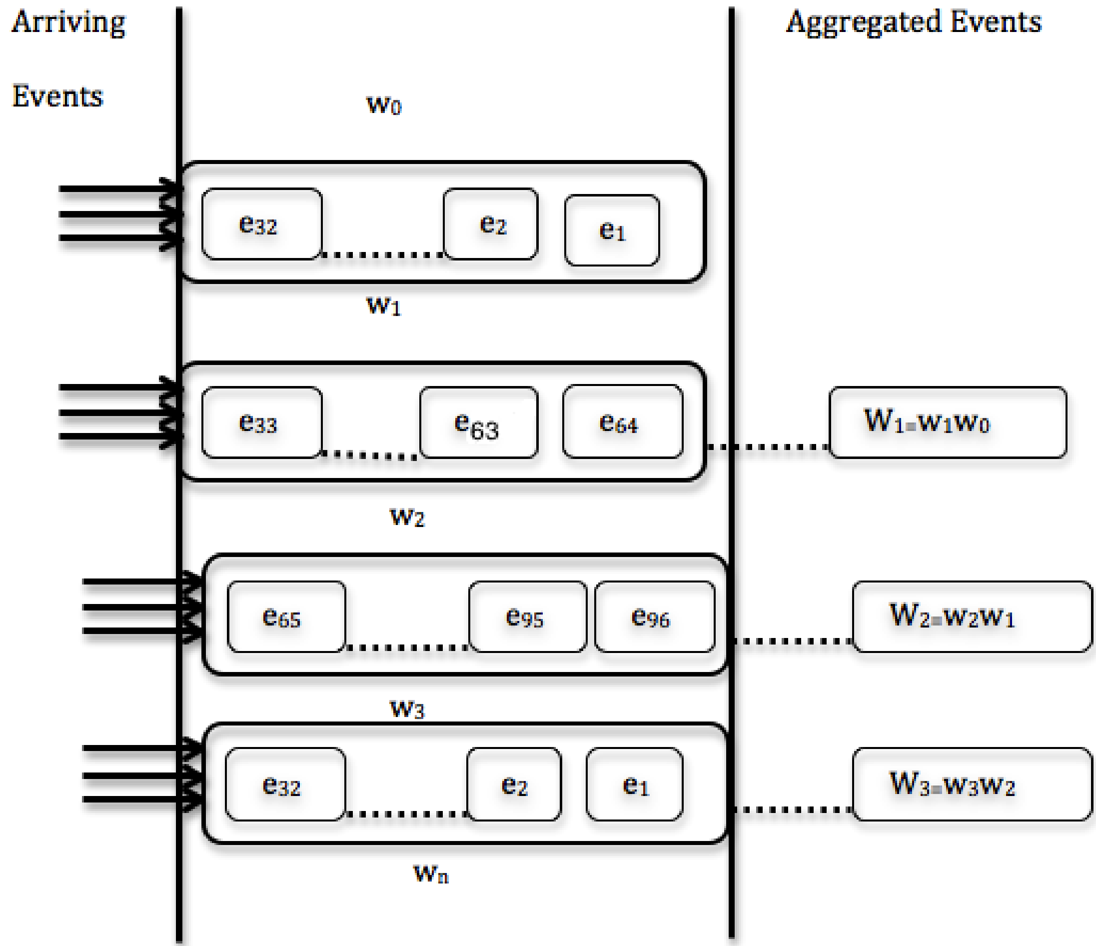


Figure 6.1: Event Processing Using Sliding Window

system keeps a record of all incoming and outgoing events from the window based on the temporal ordering. The processing of the event stream is illustrated in Figure 6.1.

The events e_1 to e_{32} are collected in the window W_1 according to their arrival order. It passes to the aggregation phase where computation for features is made after the collection of 64 events.

for $i \geq 1, w_i = e_i + e_{i+31}$

On arrival of the next consecutive events, e_{33} to e_{64} next window w_2 is formed. When the two consecutive windows have accumulated 64 events, W_i reaches the aggregation phase, it combines with previous window W_{i-1} to compute the features.

for $i \geq 1, W_i = w_i + w_{i-1}$, for example during the first window W_1 is formed as an aggregation of events from $\{w_1, w_0\}$.

Computations are made to derive features from the events. According to the sequence of arrival, the most recent window is retained, and the older window is removed from the computation. Thus a *sliding window* is created with a window length of 64 events. The window comprises of the most recent 32 events and new 32 events according to the sequence of arrival as illustrated in Figure 6.1 .

The latency is computed according to the arrival of the events during each stage of computation. Processing Element 1 windows the events based on the sliding window technique and pushes the data to Processing Element 2 (PE2). PE2 performs the computation in the data to derive features and in turn passes events to Processing Element 3 for pattern matching. The complete process is represented as a data flow graph by pushing the processed data to the next node and subsequently joining the static data store for pattern matching. The pattern matched data progress to the activity recognition phase. To measure the latency in each stage of the event processing, each window W_i of events is taken in to consideration. This latency is called L_i^1, measured in ms or seconds based on the event arrival. The subscript 'i' in the latency measurement L_i^1 denote the windows, whereas the superscript in the latency measurement L_i^1 denote the stages such as processing elements (PE1, PE2 and PE3) within the EPN.

The time taken for the windowing operation in Processing Element 1 (PE1) is computed as the difference between the event with the lowest timestamp and the event with the highest timestamp. For example, for window W_1, which is formed as an aggregation of events from w_1, w_0. The event e_1 in w_0 will have the lowest timestamp and the event e_{64} in w_1 will have the highest timestamp. So the timestamp for the window in the stage PE1 is represented as WT_i^0 and computed based on the lowest timestamp of the window w_0. When events leave PE1, the outcome gets time stamped WT_i^1 as illustrated in figure 6.2.

WT_i represents the waiting time between the processing elements. The WT_i influences the delay int he processing. To measure the latency, the waiting time between the processing elements need to be recorded accurately. These delays are based on the processing power of the computing node. To maintain the accuracy of the response time estimation, these delays are subtracted from the latency calculation. Latency is calculated as

$$L_i^1 = WT_i^1 - WT_i^0$$

When events leave PE2, the outcome gets time stamped WT_i^2

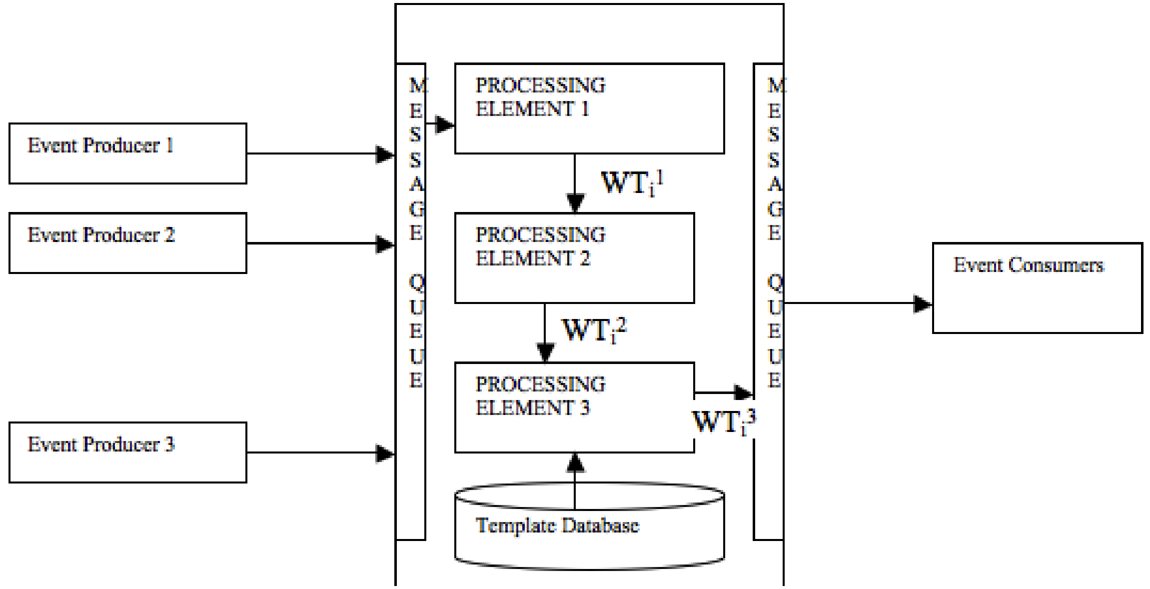


FIGURE 6.2: Latency computation in various stages of event processing

$L_i^2 = WT_i^2 - WT_i^1$

When events leave PE3, the outcome gets time stamped WT_i^3

$L_i^3 = WT_i^3 - WT_i^2$

L_i^1 is the latency measurement in the first operator of the data flow graph, i.e. PE1 in this example. Once the windowing is completed, the computation of the feature is carried out. At the end of the feature extraction in Processing Element 2 (PE2), the latency L_i^2 is computed. The pattern matching time at the end of PE3 is denoted by L_i^3. The latency L_i is measured using

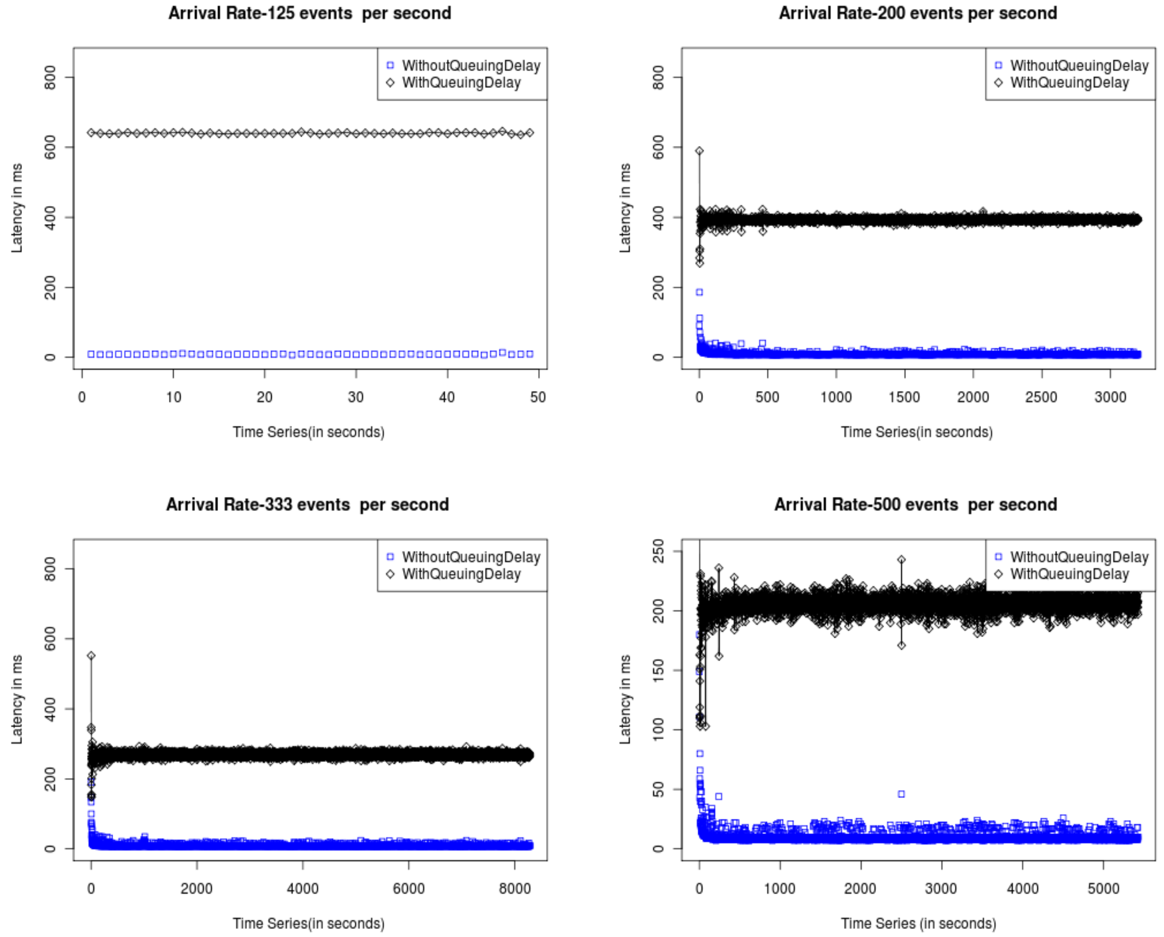
$L_i = L_i^1 + L_i^2 + L_i^3 + ...L_i^n$

L $= \frac{\Sigma_1^n L_i^n}{n}$

The graphs show the variation of latency in a few milliseconds. Under low arrival rate, the system consistently reports back the same latency over the period of time. Lots of variation in the latency (jitter), is an indication of system instability. The latency is measured with queuing delays and without queuing delays as illustrated in figures 6.1 and 6.2. The performance of the system is predictable when the queuing delays are removed from the calculation.

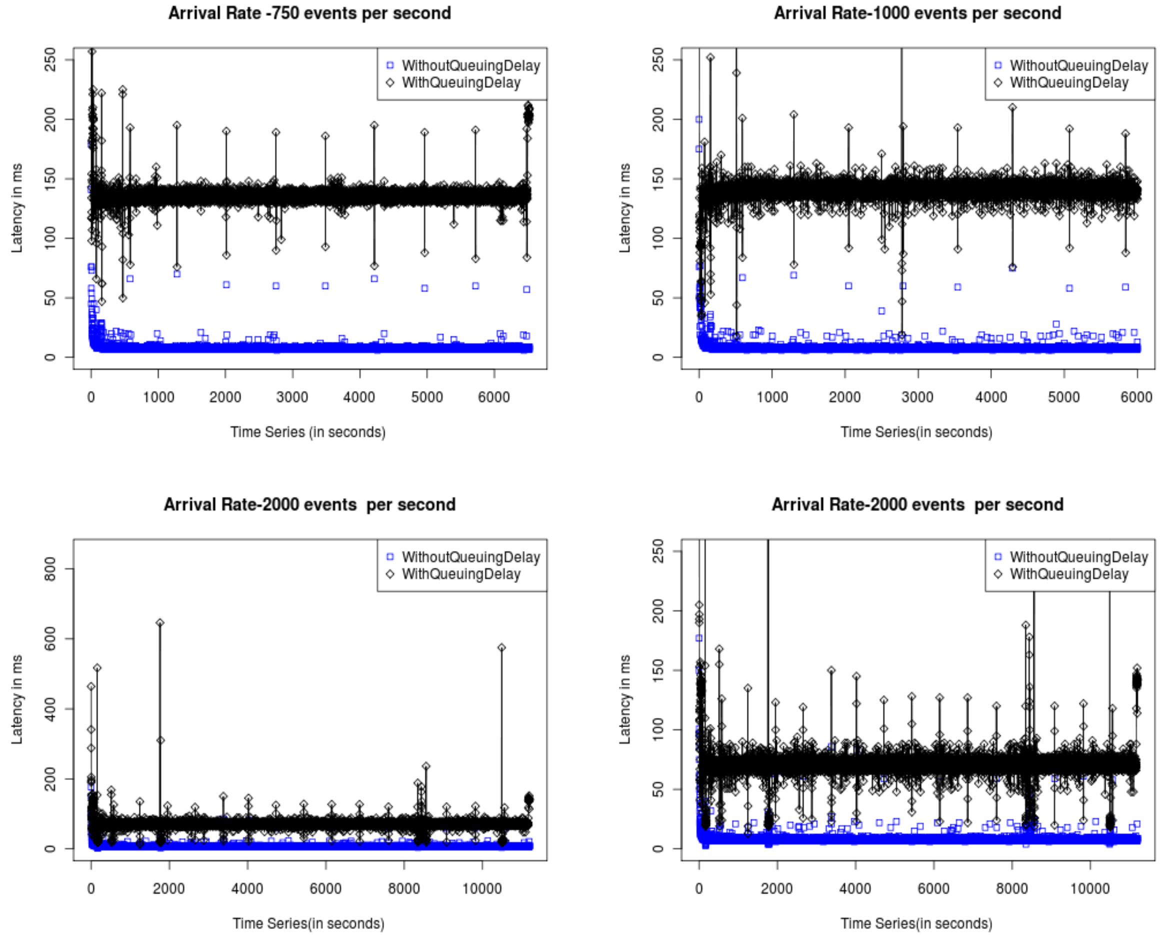
The average latency without the queuing delay was approximately 9ms. The variations in the latency occur with the increase in the arrival rate leading to regular jitter of around 4ms. The variation increased to 50ms in the higher arrival rates.

TABLE 6.1: Latency under low arrival rates



Observing the performance under various arrival rates, when the variation increased by 20%, the system response time started fluctuating. To measure the latency, the events were sent under different arrival rates starting from 100 events to 3000 events per second. Constant response time can be viewed in the graphs with arrival rates of 125, 250, 333 and 500 events per second. Under low arrival rate with $\lambda < 500$ events, the latency is stable without much jitter. Under higher event rate, for example, when the arrival rate is 2000 events per second, the variation in latency is extremely high leading to an unstable response time. In this condition, the events are designed to be redirected to multiple computing nodes. The range of variation in the latency forms an important criterion for the reconfiguration of the EPNs across various computing nodes as described in Chapter 7.

TABLE 6.2: Latency under high arrival rates



6.1.2 Calibration and prediction of EPN Response Time

The above modelling approach, illustrated in §5.2.1and§5.2.2 is applied to estimate the average response time and the CPU load exerted by any single EPN. First, a few metrics are defined or recalled, some of which are measured dynamically and the rest established through off-line calibration.

Event Arrivals denote the arrivals of events of streams in S_i and are taken to be Poisson at the rate of AR_i that is supplied to CS at the end of each reporting interval.

Processing Time denotes the total processing time a tuple or a window of tuples needs to undergo to produce an output $O^j \in O$ after a tuple in the window has just gone past the head of the queue as in Figure 5.1(b). Note that it does not include the time that a tuple spends between its arrival and reaching the head of the queue *viz.* the *queuing delays*. Moreover, the processing time depends on the nature of DAG_i that

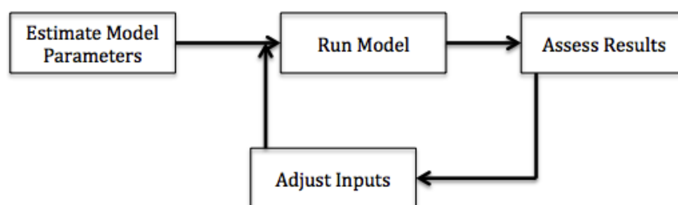


FIGURE 6.3: Steps Involved In Calibration of the response time.

EPN_i implements and also on the particular path that a tuple takes within DAG_i for it to be processed and an appropriate output to be generated.

Given the non-deterministic nature of event arrival, the processing time is modeled as a random variable of some unknown distribution. When EPN_i generates several outputs, the one that takes the maximum processing time will be of interest. For this output, b_i is defined as the *average processing time* and $M_{2,i}$ as the *second moment* of the processing times.

The *Processing Load* imposed by EPN_i on the host node is ρ_i. Specifically, ρ_i is the fraction of the time EPN_i used the node's CPU. Thus, $\rho_i = AR_i \times b_i$ and this relation is used to establish $M_{2,i}$ and b_i through *calibration* as described below.

EPN_i is hosted on its own on a node and subject to *small* arrival rates AR_i (to eliminate or at least minimize queuing delays); the CPU usage for each rate is observed and the resulting processing times are also computed from the observations. From these times, the most processing intensive output is identified, and $M_{2,i}$ and b_i are established. EPNs typically process input events in groups or windows of, say, w tuples; if so, the arrival rate during calibration should not exceed w events per second.

In the design as illustrated in Figure 5.1, the arrival rates from the input streams differ from each other due to distinct characteristics of the multiple sources of data. Each stream of data is directed to the Event Processing Network (EPN), which has operators specific to processing the schema of the events. The input tuples are processed by the operators within the EPN and generate the output. The modelling approach is to estimate the response time of each event processing network under a given condition of arrival rate and the specification of DAG.To estimate the response time for the EPN, identification of the inputs and the desirable output for model acts as an important aspect to begin the calibration process. The four stages involved in the process of the calibration are illustrated in figure 6.3 and the steps below:

1. Estimation of the model parameters: Identification of the inputs and how the inputs are related to the given problem acts as a first step in the model calibration. In this system the main input parameter identified during the start of the experiment is the event arrival rates. The event arrival rate needs to be mapped to the data source and in turn will be mapped to the schema of the event processing(for example, ambient kitchen use case) . A fixed number of predetermined event sources and event processing networks related to the activity recognition application were used to initiate this process.

2. Model Execution: To run the model the target need to be identified to calibrate the model. In this research, response time is identified as the desirable target. Model was run using various arrival rates and the response time (latency) was measured at every stage. The aim of the calibration process was to predict the response time under varying arrival rates. Figure **??**, **??** illustrates the impact on the metrics (such as network read, network write, disk read, disk write, average load, memory consumption) of the hired computing nodes for a high and low arrival rates. The average load and the network read were identifies as the parameters which could impact the response time. The model design was focused to incorporate the attributes such as arrival rate and the CPU utilisation.

3. Assessment of the results: The goodness of fit in terms of the predicted response time and the measured response time acts as a factor to decide the suitability of the calibration. Figure 6.4, illustrate the minimum deviation between the prediction and measurement of the response time.

4. Adjustment of Inputs: The accuracy of the prediction can be improved through regular interventions based on any changes to the event processing or the changes to the data schema. One of the main factor identified in the process is the CPU utilization and the average service time b_i . The table 6.3 summarises the latency prediction. The CPU load exerted by each EPN is noted and the metrics to predict the response time were event arrivals, processing time and processing CPU utilisation. For each arrival rate, the CPU utilisation ρ_i is observed. Based on this value, the tuples in the system $(\frac{\rho}{1-\rho})$ and the average service times b_i $(\frac{\rho}{ArrivalRate})$ are calibrated. The ρ is converted to address the seconds using $\frac{\rho}{100}$. This process was repeated for many arrival rates using single EPN and multiple EPNs deployment to derive the average service time, which in turn was used to estimate the CPU and response time for any incoming events. The latency is calculated using $(tuplespersecond + 1) \times (averageservicetime) \times 64tuples \times 1000$

Table 6.3: Latency Calibration

ArrivalRate	CPU (percentage utilisation)	Tuples in seconds	ServiceTime in seconds	Latency in milliseconds
40	1.88	0.0191	0.00047	30.65
50	1.74	0.0177	0.00034	22.66
100	4.42	0.0462	0.00044	29.59
125	5.08	0.0535	0.00041	27.40
250	6.3	0.0672	0.000252	17.21
500	11.9	0.1350	0.000238	17.28
1000	34	0.5151	0.00034	32.96

6.2 Experiment Setup

The experiments were set up using a KVM (kernel based virtual machine) cloud, an open source cloud implementation utilizing the operating system's kernel. KVM private cloud allows greater performance than other cloud computing solutions, which rely on user-space drivers. The core objective of this research is performance centric scalability of applications. The performance requirements for the response time predictions require the virtual machines to be constructed with managed CPU, network routers and switches. During the infrastructure administration phase, a network with virtual machines and the pool of IP addresses were allocated during the initial set up. Eight virtual machines were created. With the KVM hypervisor, migrations of the EPN were performed between virtual machines. KVM architecture hosts the virtual machine images as regular Linux processes, so all the standard Linux security measures to isolate the images were utilized to build this private cloud. The architecture supported the memory management features of Linux. The private cloud used the operating systems image and pool of IP addresses to deploy the KVM instances in to a virtual data center.

The performance of the KVM cloud can be improved by pinning the processors, configuring huge pages, setting the cache modes and tuning the memory settings. Workloads that run the guest operating systems can benefit from processor pinning. Processor pinning enables guest operating systems to pin the virtual processors with one or more physical processors. Through this processor, deterministic processing power is allocated for each virtual machine. A few categories of storage intensive streaming events were configured to use raw disk volumes avoiding the page cache in

the host machine. Memory tuning of the KVM is achieved by disabling zone reclaim and through setting the swap value to zero.

In the evaluation set up for this research, the cloud utilized the guest operating system of KVM. The operating system was configured with eight virtual processors on the system and eight physical processors. The initial test iteration was performed without virtual processor pinning. Another test iteration was conducted using each virtual processor pinned to a single physical processor.

Response time predictions were run in various stages to fulfil the following objectives:

- Monitoring of various hardware aspects (CPU, RAM, memory swap, load average) with respect to varying arrival rates.
- Response time calibration of EPNs under various arrival rates.
- Prediction of the EPN response time under varying arrival rates.
- Validation of calibrated and predicted response time of the given EPN.

The experiments were conducted using single core single EPN, single core multiple EPN, multi core multiple EPN and multicore single EPN deployments. To understand the trade-offs involved in identifying the performance of the EPN, the experiments relied on the datasets collected from the ambient kitchen. The data collected from the ambient kitchen in a 10 day trial from eight devices. Each device is set to send 100 events per second. Ten selected participants were requested to use the ambient kitchen to generate the data set. The complete data collection was recorded on video to verify the generated actions using the devices. The videos were used to generate annotation for manual verification of the recognised activities. The information associated with the users, such as pictures, location, description, etc. amounts to approximately a few terabytes of data.

In order to generate the response time predictions, the trade-offs involved in using specifications of the computing node were identified. The focus was primarily in understanding the load of the system including read and write operations. In the next stage impact on memory, storage and network bandwidth was measured. In the third stage, the amount of CPU utilization was recorded. All the possible parameters required for the performance measurements were logged and analysed for varying event arrival rates. During each experiment, the number of metrics measured was

increased and a collective set of metrics (memory, CPU, network i/o, etc.) related to performance were automatically recorded during the event processing experiments.

At a given instance, the results were correlated between the variation in latency and the load. This observation on average load was the key finding to reduce the number of hardware probes in the experiments. During the reconfiguration of EPNs among multiple computing nodes, the average load (CPU utilization) was taken in to consideration to predict the response time of an EPN in a given computing node.

EPNs were created using the developers editions of open-source event processing engine called Esper, CEP engine. A standalone JVM of the ESPER is mounted in various computing nodes. Multiple combinations of configuration parameters (memory and CPU) were used to tune each engine to achieve its maximum performance. Figure 1 shows the components involved in the response time estimation. The sensors from the application communicate with an intermediary process called Adapter via plain socket or messaging bus (apache MQ server). The Adapter then converts these messages into the native format of CEP engines and transmits them using their respective application programming interfaces (API). The input streams were generated and submitted using the ambient kitchen framework in chapter 3, a set of logging methods were have developed measure the latency of the event processing networks. Initially during the start of the calibration process, input event generation components and the event processing networks under test ran in a single machine to understand the network latencies and jitter. CPUs affinity was set to minimize interferences between the sensors from ambient kitchen, adapters and CEP engines. In the next stage of the response time measurement, CEP engines ran in a single dedicated CPU, while the application specific sensors and adapter ran in the remaining virtual machines.

Tests consisted in hosting a single EPN at the CEP engine. They began with an initial one-minute warm-up phase, during which the event arrival rate increased linearly from 100 events per second to a pre-determined maximum throughput. After warm-up, the tests proceeded for at least 10 minutes in steady state with the event arrival and injection rate fixed to understand the maximum throughput. Tests requiring more time to achieve steady state (e.g. using long time-based windows) had a greater duration. All the measures reported represent averages of at least two performance runs after the system reaches a steady state. The maximum arrival rate of 2000 events per second was sent by running successive tests with observations on increasing latency and decreased throughputs until CPU utilization was maximized or some other bottleneck was reached.

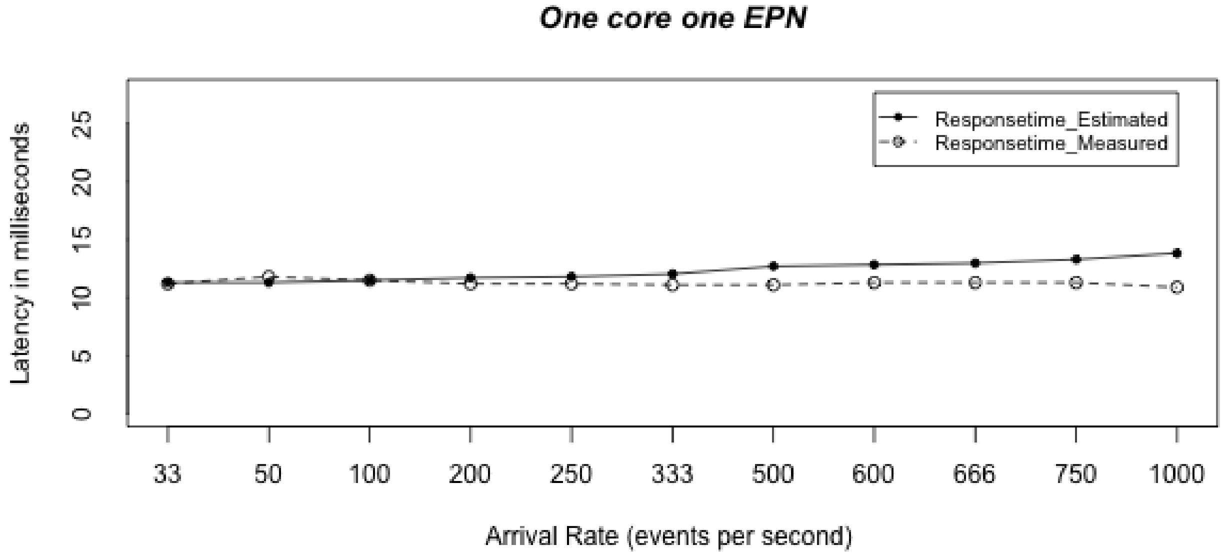


FIGURE 6.4: Comparison of Predicted and Monitored Response Time

The table 6.3 describe CPU is measure as a percentage utilization. The tuples processing time is estimated using the queuing theory based on the arrival rate. The third column in the table illustrates the estimated processing time of tuple for various incoming arrival rate. The fourth column illustrates the total service time for the complete end-to-end application output (for example, activity recognition in ambient kitchen).

6.2.1 Validation of Response Time Prediction by Measurement

To start with, a single EPN was hosted on a single node to estimate the processing time of the EPN. The calibration process was started using a low event arrival rate below 40 events per second; the average processing time of the EPN was established as b (§6.1.1). The accuracy of the calibrated processing time b was assessed by running the experiments as described below. The response time RT was measured by varying the arrival rate AR over a larger range (up to 2000 events per second). For each AR, the response time was analytically estimated using equations (5.1) - (5.2) that are modified for the case wherein a node hosts only one EPN – the calibrated one. The modification simplifies (4) to: $W = b\ [\ \rho/(1-\rho\ ) + 1]$. (Note: this simplification does not hold when several EPNs are hosted on a single node.) Figure 6.4 plots the measured and estimated response time for various arrival rates.

The latency achieved in this experiments increased with the increase in incoming arrival rate. The estimated latency for each arrival rate were calculated and compared with the actual latency measured in the experiments. The deviation between the estimated and the actual latency started increasing with the increase in the arrival rate. In both cases, computing nodes were not fully utilizing the available resources (utilization of its CPU was between 50 and 90 percentage) when its client API adapter became the bottle-neck. Dedicating more CPU-cores to the adapter (up to 7) did not solve this issue. So the EPNs had to be hosted on the multiple machines to maintain the stable or desired response time under varying arrival rate.

6.3 Conclusion

The design, deployment and testing of the EPN reponse time prediction is acheived based on the queuing theory principles. The models examined in this chapter are of type M/G/1. In some computing nodes, there can be more than one poisson process of event arrival from various devices. EPNs have varying arrival rates. The performance measures of interest are arrival rate (λ), load of the computing node (ρ) and the calibrated processing time of the EPN (b). When the load of the computing node approaches 1, both the average delays and the average queue size grow without bound. In a heavily loaded system, even a small increase in traffic can have an extreme effect on performance. The shape of the processing time distribution function affects performance only through the second moment. Even a lightly loaded M/G/1 queue can perform poorly when the variability of the demand is high. For fixed values of λ and b, the best performance is achieved when the second order moment $b^2 = 0$ i.e. minimum variability in the processing time, all EPNs require the same predicted amount of computing resources (CPU).

Chapter 7

Evaluation 2: Reconfiguration

7.1 Introduction

Managing and monitoring the computing nodes becomes critical in plenty of event based architectures such as the ambient kitchen. For example the concept behind the ambient kitchen is to create an intelligent assistive living using low level pervasive sensing probes, activity formulation or relating context to deliver the situated prompting either as dissemination or user intervention. To maintain a comprehensive catalogue of activity interventions, a service will require parallelisation techniques to aggregate and curate events emerging from multiple event sources embedded as pervasive sensors.

The reconfiguration of the EPNs are designed to reduce and optimally utilize the available computing resources. To optimize the operational cost of the event processing hardware resources in a platform such as the ambient kitchen, the clusters of computational resource need to be hired and fired based on the demand from users. In an ideal world of ambient kitchen deployment, the dynamic properties about event sources (sensor location, time of sensing, frequency of sensor event transmission, embedded objects, provenance of sensing events, etc.) need to be sorted, updated and added to cope up with the real time streaming requirements. Planning the service or computing resource capacity ahead of time becomes harder in such applications. On-demand computational power is available to address an upsurge in events during the evening hours or far fewer in the rest of the day is essential to deliver the service. This example serves as one of the practical use case, dependent on the estimation of the service and acquisition of computing resources.

This chapter focuses in the estimation of the computational resources for each EPN and identifies the EPN placement with dynamic movement across multiple computing nodes matching the response time criteria. The key criteria for the online deployment is to provide a high degree of flexibility and low number of hardware probes to maintain the simplicity of the system. The estimation of the resources to process the streaming events from the various event sources should not cause any additional overhead to the hardware or compromise the response time. The requirement for the real time systems is scalability in terms of adding more computing resources to increase the throughput of the event processing networks without compromising the response time of the EPNs. This chapter analyzes the core algorithm design, implementation and testing for the hiring and firing the computing nodes to process the streaming events in multiple computing nodes. The experiments and evaluation facilitate the algorithms to demonstrate the reconfiguration of EPNs based on the response time prediction.

7.2 Design of Configuration Scheduler

The placement of event processing networks requires dynamic movement to improve the response time across continuous iterations. The event streams involve execution of long running calculations on the performance of the event processing networks. Many applications exhibit significant run time variations that require incremental or decremented processing requirements.

The reconfiguration algorithm determines the count of computing nodes required to process the EPNs and the order in which the EPNs are reconfigured within the available computing nodes. The algorithm focuses on the EPNs whose performance is affected. When a single node hosts multiple EPNs, the configuration scheduler places the EPN in an appropriate computing node to meet the response time target. The reconfiguration algorithm maintains the optimum resources by scaling up, scaling down or optimal movement of EPN. The configuration scheduler delivers the target response time T of the applications by hiring the minimum number of cloud computing nodes. The high level architecture of the configuration scheduler as illustrated in Figure 4.3 follows the three main aspects listed below to meet the target response time even when a single node hosts multiple EPNs.

1. Shifting some EPNs among existing hired nodes

2. Hiring more nodes when necessary

3. Releasing nodes when arrival rates decrease

The design goals for the reconfiguration of the EPNs focus on minimizing the latency of the event processing. The configuration of data sources and the computing nodes are achieved using a decentralised and asynchronous architecture. All computing nodes share the same functionality and responsibilities. There is no central computing node with specialised responsibilities. The reconfiguration of the EPNs are based on five parameters: the measured response time RT_i (as periodically reported to CS), the target response time T_i, total arrival rate AR_i, the processing time b_i and an estimated response time W_i. The main functions of the configuration scheduler are summarised below:

7.2.1 Grouping of EPN:

The configuration scheduler finds the optimum set of EPNs to host in a single computing node. Each computing node hosts multiple EPNs. For example, consider a scenario where $Node_i$ hosts EPN_1, EPN_2 to process events from data source A_1, A_2, $Node_j$ hosts EPN_3 to process events from data source B_1 and so forth. Due to the fluctuating arrival rate, load sharing needs to be performed frequently to maintain a stable response time target (T). Grouping of the EPNs across two nodes can be accomplished to utilise the minimum number of nodes. Grouping of the EPNs is dependent on the individual response time. The grouping of the EPNs should be determined using the relative deviation δ_i^m of the measured response time (RT_i) and the average target response-time (T_i) for each EPN_i.

7.2.2 Optimum response time:

The average performance of the computing node is influenced by the arrival rate (AR_i) and complexity of the EPN. The complexity of the EPN is identified using the average service time, b and the second order moment, $M_{2,i}$ simplified to be the squared coefficient of the service time variance b_i^2. The service time for each EPN is obtained through the calibration process described in § 6.1.1. When the arrival rate increases, both the queuing delay and processing delay increase leading to a heavily loaded system. For fixed arrival rate of events AR_i and b, the best performance is achieved when $b_i^2 = 0$. The reconfiguration of EPNs across multiple computing nodes should provide a stable service time to satisfy the response time target.

7.2.3 Balanced resource utilization and response time:

Basic performance measurement of any EPN is dependent on the steady-state utilization of the hardware resource in the computing node. In a single node, multiple EPNs with varying complexities are likely to compete for resources. Consider the arrival of events in n EPNs . The total load utilization of the computing nodes is

$U = \Sigma_{i=1}^{n} \rho_i$

Regardless of the complexity of processing, all EPNs should maintain the target response time. The reconfiguration algorithms should maintain the stable response time under varying event arrival.

7.2.4 Optimum resource utilization:

In continuous event processing, the arrival of events is estimated based on a single data source or by merging events from large number of independent sources. Events are assumed to arrive according to a poisson process with arrival rate λ. Events wait in an unbounded M/G/1 queue. The computing nodes undergo an alternating periods of being idle and busy. An idle period starts with the departure of the event leaving an empty queue and ends with the arrival of the next event.

A busy period in the computing nodes start with the arrival of events until an empty queue is achieved. A busy period in the server of any order should not affect the response time and optimum number of the computing nodes should be hired to maintain the response time of EPNs.

7.3 Implementation of Configuration Scheduler

Each node monitors for every EPN_i the response times and the sum of arrival rates of incoming event streams fed to that EPN_i; the average response time and the largest total arrival rate observed for EPN_i over the reporting interval are recorded as RT_i and AR_i respectively, and are reported to CS at the end of each interval. For example, $Node_1$ in Figure 4.3, which hosts EPN_1, EPN_2 and EPN_3, will report to CS $\{RT_1, AR_1\}$, $\{RT_2, AR_2\}$ and $\{RT_3, AR_3\}$.

Let T_i denote the average response-time target specified for any given EPN_i. We define:

$$\delta_i^m = \frac{RT_i - T_i}{T_i} \tag{7.1}$$

where δ_i^m is the *relative deviation* of RT_i and m indicates that δ_i^m is measurable.

We define lower and upper bounds for δ_i^m: *low water mark* is denoted as LW . If $LW \leq \delta_i^m$, then EPN_i is deemed to meet its performance target within the scope of chosen LW and HW.

If all q EPNs in the system are deemed to meet their respective targets, then the current configuration is considered to be working well and CS does nothing; otherwise, CS decides on a new configuration by dividing q EPNs into ζ disjoint sets, $Z_1, Z_2, \ldots, Z_\zeta$, and by ensuring that the following two constraints are met when all EPNs of every given Z_x, $1 \leq x \leq \zeta$, are hosted within a unique node:

1. ζ is the smallest possible, and
2. (a) $LW \leq \delta^m$ holds for each EPN in the system, and
   (b) The total load exerted by the EPNs of every Z_x does not exceed a load threshold, $\rho_{th} \leq 1$.

These two constraints make the new configuration decided by the CS an optimal one. Meeting the first constraint becomes a problem of *optimal assignment*, provided that 2a and 2b can be analytically evaluated. To evaluate 2a and 2b for a given configuration choice, formulae are derived to *estimate* the average response times and the CPU load. If estimations are accurate, then the evaluations of 2a and 2b will be accurate.

The optimal assignment problem can be NP-complete; hence, it is solved using the well-known heuristic algorithm known as the *bin-packing* algorithm. The algorithm packs the EPNs into the smallest number of nodes (bins), subject to conditions 2a and 2b. This heuristic algorithm is presented in this section.

When CS observes that one or more EPNs have their δ^m exceeding HW or falling below LW, it seeks a new appropriate configuration. This selection process itself does not require halting of any EPNs and can proceed in parallel; however, if it is decided that the new configuration be implemented, then some EPNs will inevitably have to be moved to different nodes; also, some event streams will have to be re-directed and some others may have to be duplicated and streamed to multiple nodes

The elements of the configuration scheduler consist of four major components, as illustrated in Figure 7.1. The components residing inside the configuration scheduler have the five main goals listed below:

1. Run EPN efficiently with a guaranteed response time even if they share multiple EPNs within the same computing node.
2. The EPN should meet response time targets performance reliability with event rate fluctuation.
3. When multiple EPNs are placed on the same processing node, the individual EPNs should be executed or re-configured without starving of resources. As a generic rule of thumb, relative deviation from the target response time is used move the EPNs.
4. The reconfiguration algorithm should be utilizing the minimum number of computing nodes satisfying target response time.
5. Support the reconfiguration in runtime with a minimum delay and without requiring a restart of services.

The configuration scheduler is executed as an independent process that acts as the response centric placement agent for the EPNs in an appropriate computing node. In a constant interval defined by the user, the configuration scheduler handles the critical sections of EPN placement through shrinking and scaling of the computing nodes. The compelling attribute of streaming events is the fluctuating event arrival rate. In some cases, a desirable response time can be achieved when multiple EPNs are placed in a computing node. In a few cases, due to the increasing arrival rate, the placement of the multiple EPNs does not provide the desired response time. Based on the incoming arrival rate, the EPNs need to be placed in multiple computing nodes. Feasibility of this approach is more affordable due to the availability of low cost and affordable infrastructure through cloud computing. Each EPN can be placed exclusively in one computing node or multiple EPNs can be placed in multiple computing nodes. To achieve the optimum parallelization of EPNs in multiple computing nodes, an architecture is designed as illustrated in Figure 7.1, with the major components listed below:

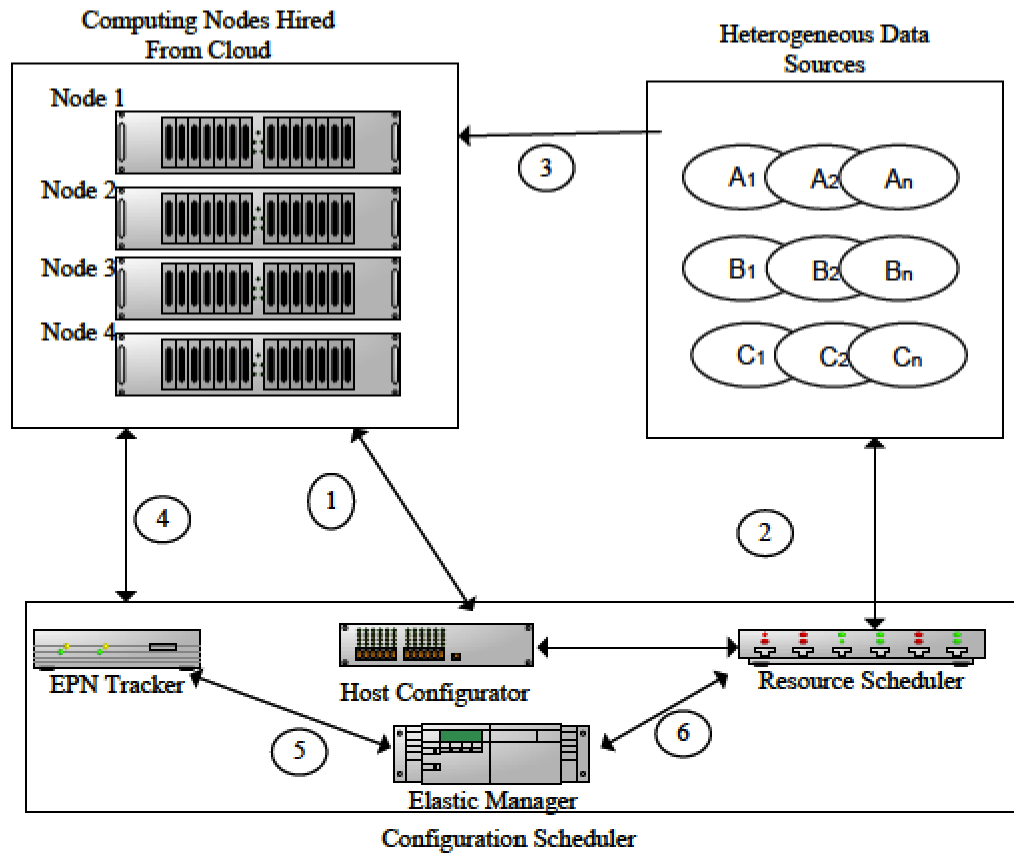


FIGURE 7.1: Element of configuration scheduler

7.3.1 Resource Scheduler:

The resource scheduler is a computing module residing inside the configuration scheduler. The resource scheduler assigns the incoming new event sources to EPNs residing in computing nodes. Each computing node is configured with a number of mapped communication interfaces (such as communication sockets) based on the capacity of hardware resources. Each EPN placed in the computing nodes listens to a particular TCP/IP socket or a messaging queue. The resource scheduler maintains the metadata of the computing nodes, list of communication interfaces of the EPNs residing on the computing nodes and the list of event sources. The details of the computing nodes are propagated to the event sources through the host allocation messages from the elastic manager and act as a bridging module to connect event sources and computing nodes.

The heterogeneous event sources connect to this module and receive the details of the computing nodes and the EPNs to stream the events. In order to map the event

sources with the appropriate EPNs, the resource scheduler maintains the list of meta-data about the EPNs such as input attributes and identification of EPNs. Whenever a new event source sends a connection request, the EPN identifiers and the input attributes are sent to the elastic manager to initiate the communication. Based on the instruction from the elastic manager, the event sources are mounted in a computing node.

Resource schedulers are set up with an automatic subscription from the elastic manager about new placement of the EPNs or movement of the EPNs. At any instance of time, the resource scheduler maintains the updated list of the EPNs available in the system including the details of the host computing node, such as IP address and web sockets configured for EPN.

7.3.2 EPN Tracker:

A single configuration scheduler is used. The EPN tracker is a monitoring module placed inside each configuration scheduler. This module collects information about each EPN (arrival rate, total load of the computing node, response time of each EPN) from all the computing nodes registered in the system in regular intervals. EPN tracker calculates the total load imposed by individual EPNs placed in their node and the relative deviation from the target response time. EPN tracker maintains the four state variables namely acceptor, donor, pseudo-donor and deutero-acceptor. Each EPN is categorized under one of the states. The role of the EPN tracker is to assign EPN to the computing nodes whose response time prediction (W) matches with the average target response time (T).During the execution of the system, the EPN tracker provides the updates of the EPNs to the elastic manager.

7.3.3 Host configurator:

The host configurator maintains the registry of the computing nodes hired or fired by the system. Each computing node is configured using virtual machines from cloud computing, with a specified amount of hardware, storage, memory, CPU and related resources. The host configurator needs to register the details of the hiring or release of the computing nodes such as credentials, keys, ids, usernames, certificates, passwords, codes to access and control virtual machine features, functionality and cloud computing account. During the system initialisation virtual machines are registered as the computing nodes. The hiring and firing of the computing nodes are abstracted to all

other modules in the configuration scheduler and only the list of the hired nodes with the details of the IP address and communication ports are visible to other modules.

Each computing node has an independent operating system to execute the EPNs deployed in its node. The host configurator is a module residing inside the configuration scheduler to manage the hired computing nodes. The resource allocator module redirects the EPNs during the runtime, whereas the host configurator makes the initial acquisition and release of the computing nodes through interaction with the various cloud service providers. The host configurator determines the definition of environment including the selection of IP address, routing configuration and port forwarding rules during the initialization of the system. The main tasks performed during the configuration of the host computing nodes are listed below:

7.3.3.1 Hiring and registration of computing nodes:

A virtual machine hired through the cloud service provider, contains a few parameters such as network I/O, CPU, etc., which are inter-dependent on the other virtual machines running in a host server. For example, the CPU availability is dependent on the utilization and sharing of the other VMs within the cloud. The set up of the environment should contain static CPU allocation, which will not limit the ability of the computing node (virtual machine) to deliver the optimum response time. Static resources are specified in terms of CPU, memory and specification of the computing node. This is defined based on the nature of the application. In this research, ambient kitchen requirements are used to specify the resources.

7.3.3.2 EPN instances:

The host configurator maintains the snapshots of the computing node or the virtual machine containing the EPN to provide easy and efficient access to the EPNs. The instance of the EPNs is made available during the system initialization.

7.3.3.3 Local autonomy:

Each computing node is hired with specific preallocated resources. This includes the memory to run the EPN, network I/O, communication socket end points, hard disk and related parameters.

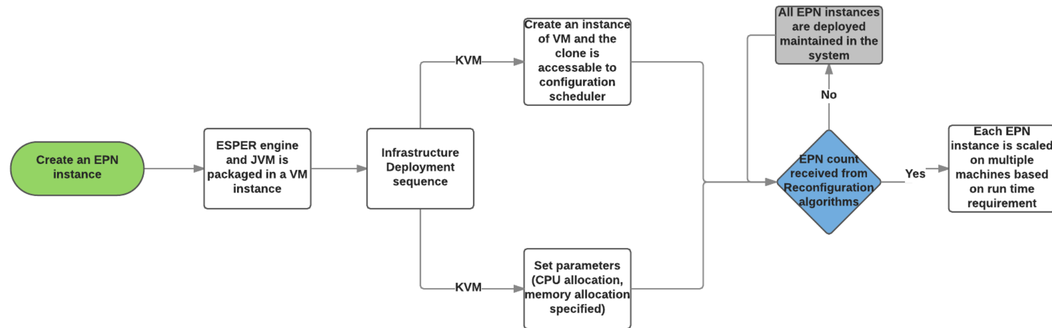


FIGURE 7.2: KVM set up of the cloud

7.3.3.4 Parameters for the computing node:

The host node allocator maintains the parameters to set up the computing node. To test the reconfiguration of the EPNs, the hypervisor should not force the machines hired through the cloud service provider with a limited or varying CPU or memory resources. Given a global and varying provisioning of resources, the EPN might deliver exceptionally poor response time in one instance and a better response time during another instance. A deterministic solution cannot be achieved when there is interference from the hypervisor in terms of the resource allocation. Computing nodes are hired with the deterministic CPU allocation.

7.3.4 Elastic Management (Shrink/Scale):

Elastic Management (EM) computes the optimal placement of the EPN, thereby shrinking and scaling the computing nodes. This module computes the reconfiguration algorithm and sends the notifications to the EPN tracker. Suppose the system contains q EPNs, each EPN is divided between ζ disjoint sets placed in independent computing nodes. The movement of the EPN in to a new node or in an existing node is sent to the resource scheduler.

The figure 7.2, illustrate the KVM-based set up of the scalability of EPNs.

7.4 Evaluation of Configuration Scheduler and Reconfiguration Algorithm

The configuration scheduler is designed to enable performance centric placement and processing of events over a set of computing nodes. The system processes event streams arising from multiple event sources at the same instance of time. This provides the opportunity to optimally map events to computing nodes and maximize the utilization of the computing resources. The scheduler use the reconfiguration algorithm 5.3 to optimize the placement of EPNs with the specific schema. This is independent of the complexity of the jobs and supports different types of data schema proving greater control on the performance constraints (response time). This framework allows optimal use of the computing resources over a varied set of workloads (small jobs to large jobs and everything in between). The core functionality of the reconfiguration is to support the optimal placement of EPNs. Each computing node is assessed based on the load and the predicted response time. The algorithm used for determining the next, appropriate configuration to place the EPN is a heuristic one and works in three parts. To demonstrate the algorithm and the evaluation of the configuration scheduler, three varying levels of event arrival rates of the EPNs were simulated. The varying arrival rates were used to test the algorithm and evaluate the scaling or shrinking of the computing resources. In each case, the target response time and relative deviation from the response time was monitored. The experiments were designed in three parts, executed over a period of hours and the following concepts were examined.

1. Placement of EPN within the existing computing nodes
2. Hiring new computing nodes to meet the incoming load
3. Firing computing nodes and moving existing EPNs to under utilized nodes

In **part1** i.e.. placement of EPNs, during system execution, the EPN tracker builds three types of ordered lists that provide a structured global view on the performance status of EPNs along with the total load on each host node. The EPN tracker scans every node N_j, and builds the EPN list EL_j, which contains the details of all EPNs hosted by N_j. In addition to this, based on the relative deviation from the performance metrics, a *donorlist* DL consisting of computing nodes is created to give up some of the existing EPNs. The *acceptorlist* AL indicates the list of computing nodes that

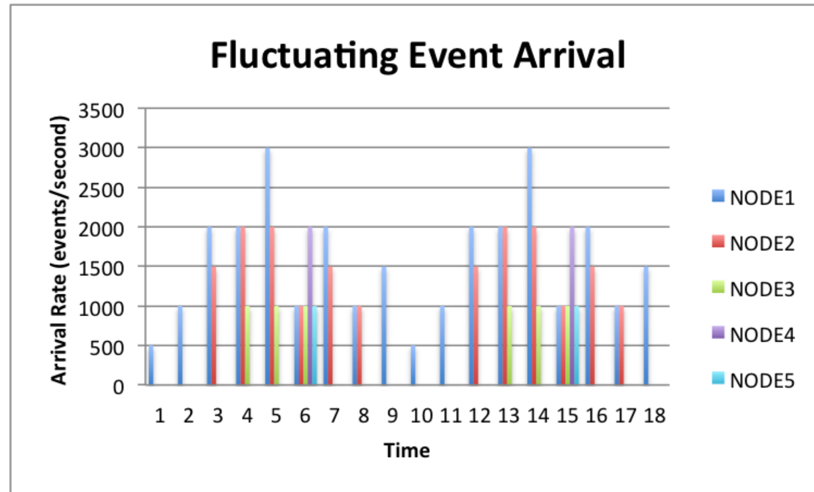


FIGURE 7.3: Fluctuating Arrival Rate in EPNs

can possibly host additional EPNs. The metrics defining the hardware resources (CPU, load (ρ)) are subscribed by the EPN tracker to create the *acceptorlist* AL and the *donorlist* DL. To demonstrate the functionality of the algorithm in detail, an experiment is set up with five EPNs subscribed to events from multiple event sources. The arrival rate of events keeps fluctuating over the period of time considered and there are moments when there are no events arriving for a particular EPN. In 7.3, during the first hour, only one EPN receives events and only one node is hired by the system. The arrival rate is steadily increased during the next three hours for all five EPNs.

As a continuous monitoring task, relative deviation for every node N_j, δ_i^m is computed by the EPN tracker. The elastic manager subscribes to the EPN tracker to receive the information on AL and DL. For each EPN_i hosted by N_j, the EPN list EL_j is ordered in the decreasing order of relative deviation, (δ_i^m - HW). For example, if node N_j is hosting no EPN_i with $\delta_i^m - HW > 0$, it is marked as an *acceptor* node and is entered in the acceptor list. Otherwise, it is marked as a *donor* node and is entered in the donor list. The donor list is ordered in the decreasing order of the total load (ρ) imposed on the nodes, while AL is ordered in other way round; in the increasing order of the total load on the nodes.

Discussions : The elastic manager initiates the reconfiguration process at pre-determined intervals. At the end of part1, DL will be empty if an execution of the algorithm has started because some EPN had $\delta^m < LW$; similarly AL will be empty if each node had some EPN with $\delta^m > LW$ when the algorithm starts. Both DL and AL cannot, however be empty when EPN placement completes.

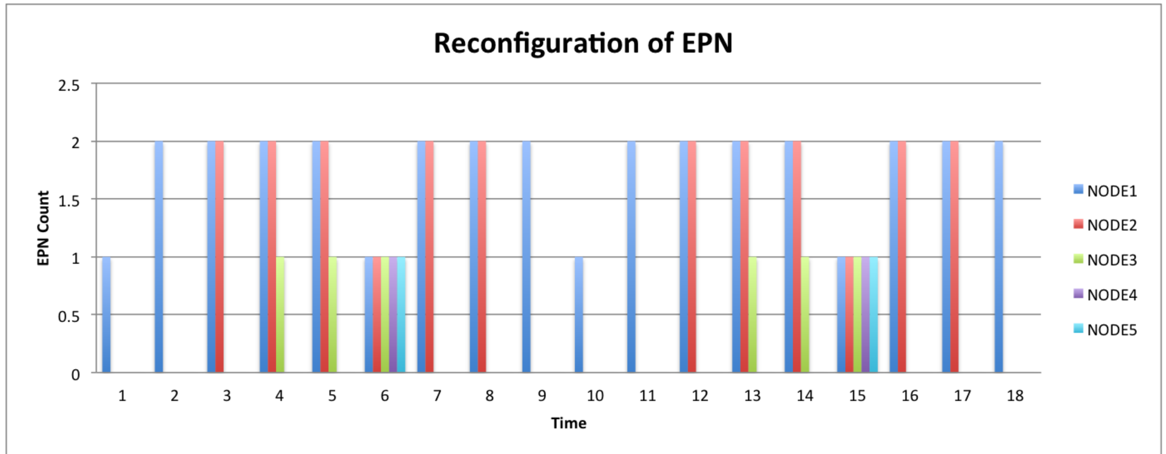


FIGURE 7.4: Reconfiguration of EPN in fluctuating load

The incoming events go through an alternating period of the event bursts. Let us consider the case of average length of time period as stated in Figure 7.3. The placement of EPN is completely dependent on the sequence of arrival rate for each EPN. At every arrival instant, the δ_i^m deviates by an amount equal to the processing requirement of the incoming events. The *donor list* DL is initially empty until time $t = 2$. The arrival rate to the EPNs in Figure 7.4 is varied under time *t=3*. The AL is empty in this instance. The deviation, δ_i^m computed with the predicted load, is greater than the HW. A new node is hired at instance of time, *t=3*.

The objective of **part2** is to make the non-empty donor list empty. Let D and A denote the first node in DL and AL, respectively. D is the most heavily loaded donor node, hosting at least one EPN whose $\delta^m > HW$. A is the most lightly loaded acceptor and hence it becomes the first candidate to be tried for the possibility of hosting an under-performing EPN in D. This forms the basis of part2 which is skipped if the donor list is empty to start with. In Figure 7.4 at time *t=5*, nodes 1 and 2 are identified as donors, with node 3 as the acceptor. However, based on the estimated deviation **Step 2.4** is executed, by hiring the new node to place the EPNs.

Step 2.1: The elastic manager first checks the EPN_D, i.e., the EPN with the largest deviation (δ^m - HW) in donor list, EL_D. The EPN_D is virtually placed in the first node in the acceptor list, $EL_A = EL_A \cup \{EPN_D\}$. Computation is made on the load exerted by all EPNs already exisiting in the acceptor node as stated by

$$Z_A = \{EPN_i : EPN_i \in EL_A\}$$

The outcome is checked to establish whether the relative deviation is bounded between the high and low water marks. The total load exerted by the EPNs should not deviate

from the load threshold. If they hold, then execute Step 2.2. If they do not, reset $EL_A = EL_A - EPN_D$; if there is a node that is ordered immediately after A in AL, then set A to be that node and execute Step 2.3; otherwise, execute Step 2.4.

Step 2.2: In this step the EPN from the donor list satifies the movement in to the computing node in the acceptor list. Given the new placement of the EPN, the donor list is updated by setting

$EL_D = EL_D - EPN_D$

This indicates that EPN_D is to be moved from computing node in the donor list to the computing node in the acceptor list in the new configuration. Note that the contents of EL_D and EL_A are changed and therefore δ^m computed (in Step 1.1) for the EPNs in these two lists are no longer valid. So, a sub-step is *re-computed*(δ) for nodes D and A. The donor status of node D is re-assessed, if the status of D is changed, DL and AL are updated. The nodes of AL is ordered. If DL is not empty, nodes of DL are ordered. The D and A are set as the first node in DL and AL respectively. The Step 2.1 is executed. If DL is empty, part3 is executed, where the shrinking of the existing computing nodes in the acceptor list needs to be planned.

Sub-step *re-compute*(δ): If EL_j of node N_j does not reflect the EPNs that N_j is hosting in the current configuration, then $Z_x = \{EPN_i : EPN_i \in EL_j\}$ is set. For every $EPN_i \in Z_x$, δ_i^e is computed using the expression $\frac{W_i - T_i}{T_i}$ and $\delta_i^m = \delta_i^e$. The entries of EL_j are ordered in the non-increasing order of $(\delta_i^m - HW)$ and returns to the calling step.

Step 2.3: With the new A, Step 2.1 is executed.

Step 2.4: A new node is hired using host configurator and added to the acceptor list A. EL_A is initialised to $\{\}$. A is entered as the first node in AL. Step 2.1 is executed.

Discussions. Step 2.1 attempts to replace the mapping of EPN_D to D by trying to map EPN_D to A. If re-mapping is feasible, Step 2.2 maps EPN_D to A; otherwise the next node in AL is tried in Step 2.3. If AL has no suitable A at all for re-mapping EPN_D, Step 2.4 hires a new A. These steps are repeated until an EPN_D exists, i.e., until DL is not empty.

Part3 Implements the optimal shrinking of the resources using alternative placements for the EPNs from an under utilized computing node. The elastic manager uses the algorithm to reduce the size of AL, when the size is more than one. Recall that the first node in the ordered AL is the least-loaded and hence it is the first candidate to

be tried for the possibility of being freed from use. It is called the *pseudo-donor* node and denoted as D in part3. A new acceptor list called *deutero acceptor list*, DAL for short, is created as a copy of AL but without the first node in AL.

Let D and A be the first node in AL and DAL respectively. (Note: D is not in DAL.) EPNs of D are tried, one by one, to be moved to some node in DAL. If all EPNs of D cannot be moved out, part3 stops after restoring ELs of nodes in AL; otherwise, the new AL becomes DAL and the new DAL is the new AL without the first node. If the new DAL is not empty, part3 is repeated; else, the algorithm stops.

Step 3.0: Discard earlier checkpoint, if any; Checkpoint EL_j of every $N_j \in AL$;

Step 3.1: Let EPN_D be the EPN with the largest $(\delta^m - HW)$ in EL_D. $EL_A = EL_A \cup \{EPN_D\}$. $Z_A = \{EPN_i : EPN_i \in EL_A\}$ is computed and conditions 5.4 and 5.5 is checked for Z_A. If they hold, then Step 3.2 is executed. If they do not hold and if A is the last node in AL, EL_j is restored from each $N_j \in AL$ and the algorithm is terminated. If Conditions 5.4 and 5.5 do not hold for Z_A and if A is not the last node in AL, $EL_A = EL_A - EPN_D$ is reset. New A is set to be the node next to the current A in DAL and Step 3.1 is re-executed.

Step 3.2: $EL_D = EL_D - EPN_D$ is set. Sub-step *re-compute*(δ) is executed for A. The nodes of DAL is ordered. If EL_D is not empty, then A is set as the first node in DAL and Step 3.1 is executed. If EL_D is empty, then $AL = DAL$ and new DAL is set. If DAL is not empty, new A and new D is set and step 3.0 is executed. If DAL is empty, the algorithm is terminated.

At the end of the algorithm execution, elastic manager provides the reconfiguration plan to the resource scheduler. The EPN tracker updates the AL and DL based on the reconfiguration. If new nodes need to be hired or released during this process, host configurator is invoked.

7.5 Validation of Multiple EPN Reconfiguration

CS (configuration scheduler) was programmed to produce a new schedule only when triggers were applied; i.e., CS would not act whenever it observed δ^m for some EPN not satisfying $LW \leq \delta^m \leq HW$. The objective is to study the EPN performance when configuration is changed only in response to changes in AR; if that is sufficient, then proactive reporting would not be necessary and the reporting overhead can be reduced.

Table 7.1: Configurations in Response to Triggers

Hour (H)	Trigger Nature	Node 1	Node 2	Node 3	Node 4	Node 5
1	AR_1 @ $EPN_1 = 500$	EPN_1				
2* 2	AR_2 @ $EPN_2 = 500$	EPN_1 EPN_2				
3	AR_1 @ $EPN_1 = 1000$ AR_2 @ $EPN_2 = 1000$ AR_3 @ $EPN_3 = 1000$ AR_4 @ $EPN_4 = 500$	EPN_1 EPN_2	EPN_3 EPN_4			
4	AR_4 @ $EPN_4 = 1000$ AR_5 @ $EPN_5 = 1000$	EPN_1 EPN_2	EPN_3 EPN_4	EPN_5		
5	AR_2 @ $EPN_2 = 2000$	EPN_1	EPN_3	EPN_5	EPN_2	EPN_4
6	AR_5 @ $EPN_5 = 0$ AR_4 @ $EPN_4 = 500$ AR_2 @ $EPN_2 = 1000$	EPN_1 EPN_2	EPN_3 EPN_4			
7	AR_1 @ $EPN_1 = 500$ AR_2 @ $EPN_2 = 500$ AR_3 @ $EPN_3 = 500$	EPN_1 EPN_2	EPN_3 EPN_4			
8	AR_1 @ $EPN_2 = 0$ AR_4 @ $EPN_4 = 0$ AR_3 @ $EPN_3 = 0$	EPN_1				

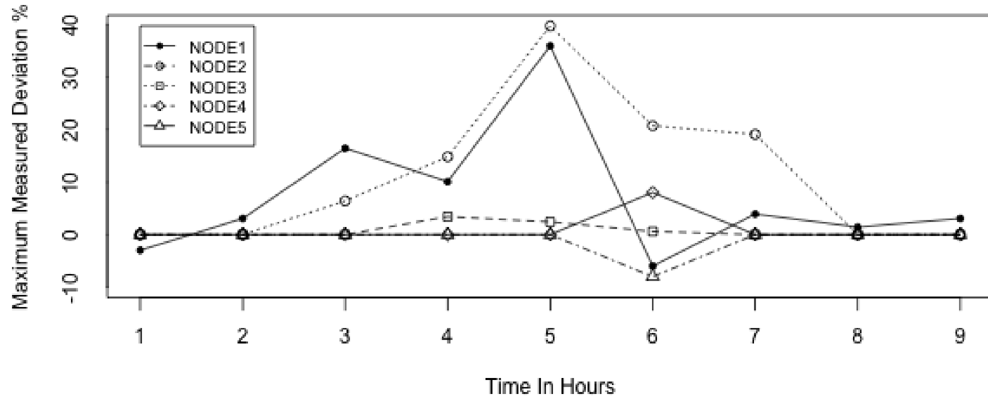


Figure 7.5: Deviation Measurement in Algorithm Results

Before the first hour, $H = 1$, AR of all 5 EPNs is zero, i.e., the system of EPNs was idle; AR_1 was set to 500 (events per second) at the start of the first hour, i.e., at $H = 1$.

The first two columns of Table 7.1 present the triggers applied at various H values. Column 2 indicates the changes applied over what existed earlier; this means that

when a trigger, say, at $H = 4$ does not involve EPN_1, AR_1 was not changed at $H = 4$ and hence AR_1 retains the value it had just before $H = 4$ which is 1000.

It can be seen from Table 7.1 that more nodes are hired as AR increases for all EPNs, and when $5 \leq H < 6$ each EPN is by itself on a single node. Starting from $H = 6$, triggers reduce the AR at some EPNs; consequently, nodes are gradually released except at $H = 7$ when the rate reductions administered are not sufficient to release any node.

Figure 7.5 plots, for each node, the maximum of δ^m of the EPNs hosted locally at $H = 1, 2, \ldots, 8, 9$. (At $H = 9$, AR_1 was set to 0, bringing the system to an idle state.) HW (set to 20%) was exceeded during $4.3 \leq H \leq 6$, the period when EPNs were receiving events at peak rates.

7.6 Conclusion

The resource sharing between the EPNs placed in a computing node involves sharing of the physical resources. Although the resource sharing improves the overall utilization of the computing node, the contention of the resources often leads to performance degradation. The reconfiguration algorithms are designed to mitigate the effects of performance degradation during the multiple EPN placements on a computing node. Placement of the EPNs has been used to resolve the response time degradation and to maximize the utilization of the available computing nodes. EPN placement is triggered by monitoring the usage of the resources in the computing node using the reconfiguration algorithm

The research leads to two inferences: the role of the configuration scheduler is illustrated to ensure that measured response time targets (RT) meet the pre-determined target (T) during the fluctuating incoming arriving rates faced by the EPNs; secondly, nodes must report both RT and AR to CS in a proactive manner as proposed in Section 7.1, not just in response to significant changes they observe in AR. The latter arises due to the fact that W tends to under-estimate RT at large values of RT and hence adjustments to existing configuration are necessary even if a large AR remains constant

Chapter 8

Evaluation 3: Inter-EPN Parallelization

8.1 Introduction

Challenges arise when EPN hosted by itself on a single host is unable to meet response time (RT) targets. To improve the response time of a single EPN, it is proposed that the event streams are distributed across multiple instances of the same EPN, with each instance hosted by itself on a distinct machine. This chapter presents a framework for parallelizing the incoming events using the same copy of the EPN in multiple computing nodes using a split, process and merge technique. In cases where the processing of event streams is stateless, an obvious parallelization solution can be easily adopted, i.e. deploy multiple copies of the EPN over different computing nodes, split the incoming streams and merge the results at constant intervals. However, when the processing of events is stateful, which is rapidly becoming a common case in complex event processing scenarios, enabling parallelization to avoid the bottleneck in performance is difficult. Computing aggregates such as maximum or minimum does not create challenges to parallelization. When complex predicates and longer windows of events are required to compute results, careful design of distribution and aggregation of the events is vital. The parallelization requirement for a single EPN can be measured in various dimensions as listed below:

1. Handling maximum number of events.
2. Memory requirements.

3. Extent of complexity in pattern matching.

The volume of streaming events and the complexity of the underlying rules involved in event processing leads to replication of EPN in multiple computing nodes. A handful of processing requirements such as sum, average or aggregate functions can be parallelized and aggregated using arithmetic functions. Enabling parallelization to process stateful events or computational interdepencies requires state sharing and careful design of the splitting and merging of events. The need to have a shared computational state in loosely coupled nodes makes it difficult to implement distributed event processing across the EPNs. The EPNs will incur additional overhead in synchronization and access to the shared states between multiple instances of the same EPN. The complex event processing and enabling mechanism coupled with additional overheads incurred in the EPNs make it challenging to determine the suitability of the parallelization and the optimal level of the parallelization. The research challenge is to build a parallel, distributed EPN where the incoming events are partitioned across multiple nodes satisfying the following conditions:

1. Replicate instances of EPN in multiple nodes in order to meet response time targets.
2. The system always delivers the same result of an ideal centralized execution of the EPN.

The core of the problem is to distribute the incoming events and aggregate the results. The design of inter-EPN parallelization should define the forwarding rules to distribute the partial results throughout the network of EPNs in a directed manner. The intermediate EPN instances handling the events forward the response to the aggregating node once the processing is completed.

Algorithms are proposed to process the streaming events across EPNs (inter-EPN) by preserving the syntax and semantics of the distributed outcome of the query. In a centralized single EPN scenario, each state of the system is stored and ordered in the temporal sequence characterizing the progression of the state from one level to the next resulting in the final outcome. To process events in multiple instances requires appropriate parallelization and aggregation strategies of the output. The parallelization strategy and aggregation is defined with the help of use cases. A use case centric approach is pursued where splitting of the event stream is undertaken as an operation without involving evaluation of complex predicates from multiple

attributes in an event tuple. A few schemes are examined for event distribution in terms of distribution operators that can be used to 'fix' loss of accuracy in the final outcome and address non-functional aspects such as scalability and performance of event based systems. The latter is assessed on dimensions relevant to our given use case such as volume of events with varying input agents, event producers, complexity of computation and processing environment.

The design behind the Inter-EPN parallelization involves broadcasting of the incoming events to some significant part of the network. Mechanisms are devised to split the events in to multiple physical event streams redirected to the network of EPNs. The target node or processing node obtains the final results. Analysis illustrates the impressive speedup in the response time by introducing inter-EPN processing in spite of the complex rules. The chapter is organised in three sections. The first section describes the concepts behind parallelization with example queries that pose difficulties and the ones that does not cause problems to parallelize. Section 2 explains the use cases to highlight different challenges in each case and discuss the algorithm sequences to achieve parallelism. Section 3 evaluates the results and limitations.

8.2 Parallelization of EPN

Parallelism involves partitioning, distributing and processing the incoming events in multiple computing nodes. Partitioning has its origin in centralized systems where a single file or tablespace were too large to handle within a single piece of hardware. Distributed databases used data partitioning by placing relational segments in multiple sites exploiting factors such as spreading the I/O bandwidth through reading and writing in parallel.

The most common strategy is to distribute tuples through *round-robin, range* or *hash* partitioning. Round robin partitioning is a technique where tuples are split sequentially in multiple computing nodes. Range partitioning is a technique where subsequent events are transferred to one computing node, move forward to the next node and returns to the first node. Range partitioning uses associative scans to distribute the tuples based on a particular range of the attribute value. Hash partitioning is a technique where tuples are distributed by applying a hashing function to an attribute value.

Both hash and range partitioning specify the distribution of the tuple on a few computing nodes or a disk, avoiding the overhead of complete search in all the computing

```
SELECT person_name , avg(accel-x)   /* the output attribute(s) */
FROM accelerometer_logs      /* the input relation */
WHERE person_id='201'; /* the predicate */
```

FIGURE 8.1: Example of the scan of acceleration logs to compute average

nodes registered by the system. The range partitioning clusters the tuples with similar attributes in the same partition; however, the hashing tends to randomize the incoming data rather than clustering it. The problem with the range partitioning is in *data skew*, where all the data is placed in one partition or the computing node and the *execution skew* in which all the execution occurs in one partition. Hashing and round robin partitioning are less susceptible to the issues surrounding skew. Some partitioning techniques use frequency of event or tuple access as one of the factors to partition and spread the data access to the partitions rather than the actual number of tuples or the attribute.

For example the system receives an event stream from accelerometers tied to the individuals residing in a care home. One of the basic arithmetic functions such as average could be used on the events as illustrated in Figure 8.1. In Figure 8.2, the events are re-directed between three nodes. The label within each node represents the identification of the computing node. The three partitioning techniques stated above are demonstrated as illustrated in Figure 8.2 to narrate the simple use case of computing average. The events are split in three different ways and the results are obtained through aggregation in the final node. Each node performs the sum of the events in x axis, called accel-x, and maintains the count of the events. The overall output is computed using the aggregate node, through the average operation as described below:

$$\text{Average} = \frac{\Sigma_{i=0}^{3} accel-x}{\Sigma_{i=0}^{3} eventcount_i}$$

The position of the events in the multidimensional space is determined by the list of attribute values defined by the multi-dimensional partitioning schemes. The partition dimension or the attribute can be decided based on the performance requirements such as throughput or response time. The number of nodes can be increased based on the events arrival. Grouping computing nodes by adjacent occurrences of event types through hashing techniques or clustering events through range partitioning or sequential event distribution in round robin partitioning might improve the performance specific to a few use cases. This example of computing an average through partitioning of events is a simple one, where computational dependency between events is very minimal. Whether partitioning distributes events to few nodes or all nodes is

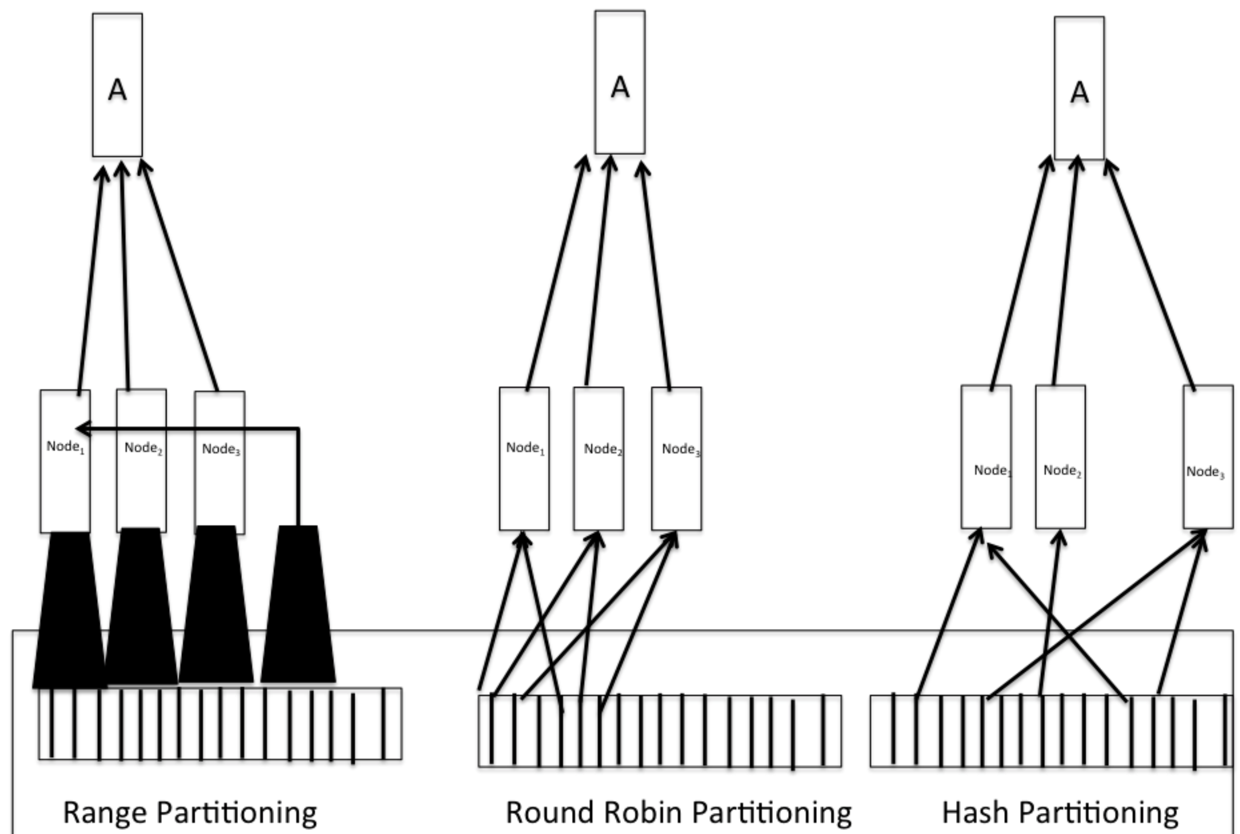


Figure 8.2: Illustration on types of stream partitioning process

an orthogonal issue. Each sophisticated partitioning improves the response time and is reserved for future work.

The main objective behind parallelization of the EPN is to distribute the incoming events to the multiple EPNs residing in multiple computing nodes with automated redirection of the events. Each EPN deployed in a computing node performs identical tasks on the arriving events. Each attribute of the event can be viewed as a co-ordinate along the (attribute) dimension. Each event tuple consists of a list of co-ordinate values and is viewed as a data point in the multidimensional space. Given the nature of the use cases, a range partitioning is used as the default strategy to test the Inter-EPN parallelization. The semantic integrity of the results is one of the main areas of this qualitative research on Inter-EPN parallelization. The results emerging out of the Inter-EPN parallelization should be equivalent to the results obtained by processing the events in a single computing node. From the performance point of view, one seeks to balance the incoming events to each computing node registered in the system.

In order to implement the Inter-EPN parallelization, application independent algorithms were implemented in Java to direct the input event streams and to manage the output event aggregation. The event processing engine (ESPER) maintain local memory holding the previously computed disjoint intervals of the arriving events to a certain point in past. In this research, it is assumed that incoming events arrive in an orderly manner. EPNs use continuous queries to filter the event streams and/or push the results further down to other event streams. The EPNs make use of the views which are similar to the SQL tables to hold multiple events for the pattern recognition. This research use sequence-based views with expiry policies (pre-determined count of incoming events) through sliding windows. The features from the event processing engine is used to create the sequence based views of the incoming events. Complex processing such as aggregation and grouping are performed on the range of the events in a particular sequence of events.Based on these reasons range partitioning is use exclusively in this research mainly focusing on the ambient kitchen use case. Other types of partitioning can be utilised to improve efficiency. However, this is beyond the scope of this research.

8.3 Implementation of Inter-EPN Parallelization

The complex computational dependency between the events poses challenges during the inter-EPN parallelization. Two approaches for parallelization of EPNs are analyzed: the first one uses a decentralised or asynchronous notification mechanism (a message passing system), where the event aggregator is dynamically assigned to one of the computing nodes complying to the rules defined in the algorithm. The second option is to express the computation as centralised or map reduce functions. In the map reduce based parallelization, the master node acts as the centralised computation and aggregation point where periodic checkpoints of the incoming events are directed to the map and the reduce operators. Following the constraints specific to both paradigms, the communication, correlation and aggregation mechanisms can be mixed. In this section, we explain the implementation of inter-EPN parallelization with the help of a simple use case. Each approach optimizes the processing between the EPNs by selecting the group of EPNs and optimally allocates the computing resources.

Within the asynchronous or decentralised notification mechanism, a search for a pattern or an event can be initiated by any computing node in a cluster, which will declare itself as an aggregator. For example, let us consider the computation of average as

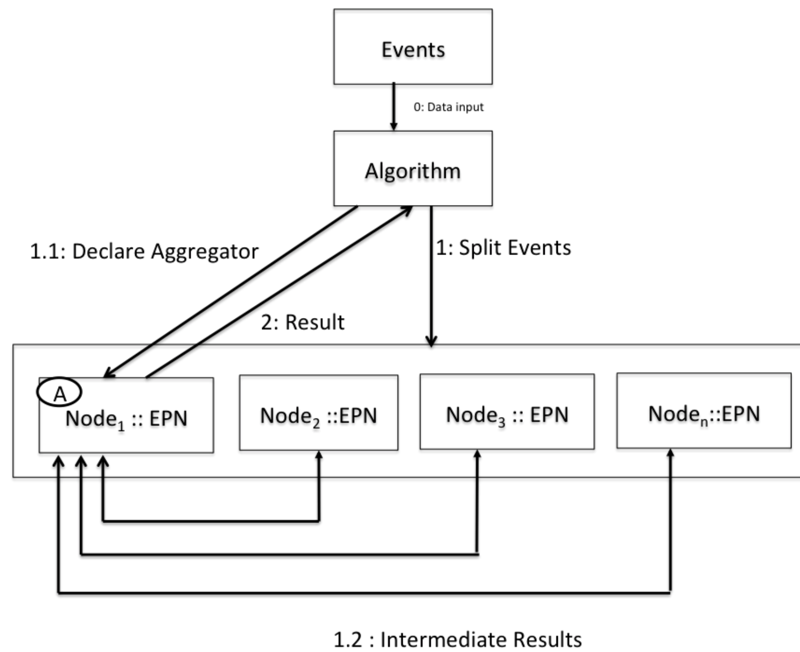


Figure 8.3: Decentralised algorithm for inter-EPN parallelization

illustrated by use case 8.1, in a decentralised manner. Any arbitrary node receiving the event first declares itself as aggregator and tries to collate the sum and the count of events from the rest of the nodes. Once the node declares itself as an aggregator, all other computing nodes in the cluster subscribe to notification from the aggregator. When the computation is accomplished based on the n depth traversal of the directed acyclic graph representing the process flow in the EPN, the aggregator node resets its state and will be ready to accept instructions from a new aggregator in the network of computing nodes, as illustrated in Figure 8.3.

Some of the key challenges to implementing decentralised event processing are listed below:

- During the process execution, a dynamic decision should be made to pick one of the computing nodes as an aggregator.
- During complex inter-dependency between events, the aggregation of the events could be undertaken in multiple nodes based on the pattern in the incoming events.When one or more computing nodes try to declare itself as an aggregator, a race condition occur. In order to prevent race conditions, one of the node is fixed as an aggregator arbitrarily.
- During a few instances of complex use cases, aggregator could be arbitrarily fixed to maintain the stability of the system.

The overall outline of the decentralised event processing to acheive inter-EPN parallelization is listed in the steps below:

Step1 The copy of the EPN is instantiated in idle computing nodes. The count of computing nodes required to hire in the system is identified through the reconfiguration algorithm described in chapter 6.

Step1.**1** The decentralised parallelization algorithm initiates the search. Let us consider the use case which computes average. Any arbitrary node receiving the event checks for the availability of the aggregator. If an aggregator is found, the node computes the sum and the count of the events. In the circumstance of no aggregator, the node declares itself as an aggregator and notifies all other nodes registered in the system.

Step1.**2** The $Node_1$ in Figure 8.3 , passes the notification to process the incoming events. The rest of the nodes in the system will act as the processing node. The results are forwarded to the aggregator nodes ($Node_1$).

Step2 Once the results are computed by the aggregator, the algorithm resets it as one of the processing nodes. Based on the arrival of the events, any node in the network would declare itself as an aggregator.

As an alternative approach, centralised or map reduce paradigm uses the customized event processing routine (EPN) for the given use case called ''Map' by receiving the input events along with key/value pairs and generates intermediate events. The map reduce library groups the set of intermediate events and passes them to the reduce function. The reduce function, which is also a customized event processing routine (EPN), groups the events based on the intermediate key and merges the events to a smaller set of values. Typically zero or one output value is produced per Reduce invocation. Splitting the events in to several groups (Map) provides the opportunity for distributing the EPN on several machines: within the same group, where a few of the EPNs are executed on virtual machines on the same server or in a different location.

Figure 8.4 shows the overall outline of the map reduce operation for the inter-EPN parallelization. The map reduce engine or appropriate API, is integrated in to the user defined program called mappers and reducers. The master receives the incoming events, communicates with the mapper, reducer and the algorithm containing the instances of the EPN. The simple case such as finding the average can be easily

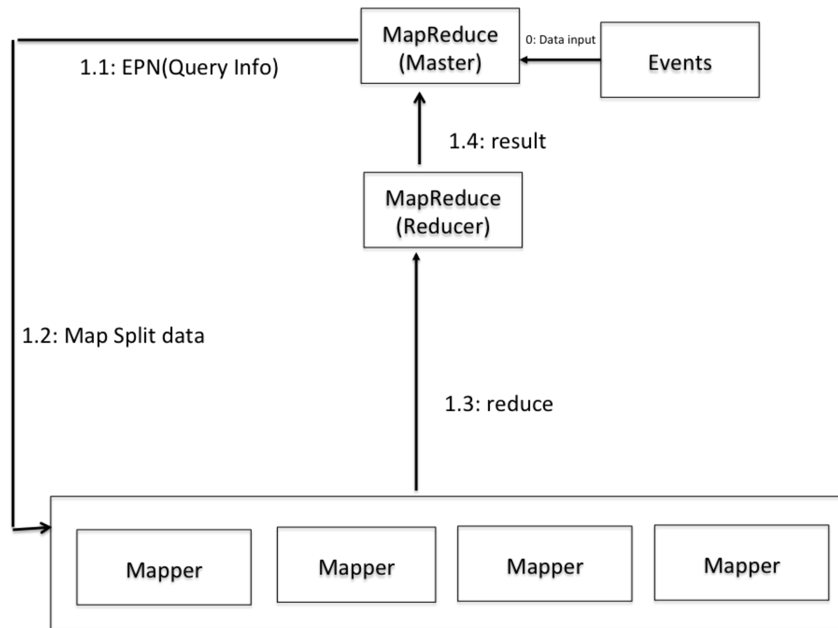


FIGURE 8.4: consecuetive events split using map reduce

accomplished through the master node. The master splits the events based on a user defined criteria. Each mapper processes the event and sends the sum and count of events. The reducer sorts, merges and forwards the results to the master. The sequence of actions is illustrated in the figure corresponding to the listed numbers.

Step1 The master splits the input events in to M pieces based on size or count of events as defined by the parameters defined by the user.

Step1.1 The master picks the idle nodes and assigns a few nodes as map and reduce task. The count of idle nodes can be picked based on the reconfiguration algorithm defined in chapter 7. The master starts multiple copies of the EPNs in the mappers.

Step1.2 A mapper is assigned a task to process the contents of the incoming events. It uses the query information from the master to process the events. The intermediate events are passed to the reducer as illustrated in **Step1.3**.

Step1.3 Periodically, the buffered events from each mapper are passed to the reducer. The reduce worker reads the intermediate events, sorts and groups the results. If the amount of intermediate events is too large to fit in the memory, additional computing nodes can be used. The master defines the architecture for the reducers during the system initialisation.

```
SELECT person_name , avg(accel-x)   /* the output attribute(s) */
FROM  merged_accelerometer_logs.win:length(64)     /* the input relation */
group by  person_id output every 32 events; /* the predicate */
```

FIGURE 8.5: Example of the scan of acceleration logs to compute rolling average

Step1.4 The reduce worker iterates intermediate results and generates the final output.

Step2 : When all the map tasks and reduce tasks are accomplished, the master sends the final aggregated results to the algorithm.

MapReduce framework are used to maintain the pool of EPN jobs and the count of EPNs subject to the change in the workload. The incoming data is split in to Map tasks. Map and Reduce task are implemented as a separate Java Virtual Machine (JVM) residing within each virtual machine. Multiple map tasks connect with multiple reduce task as specified in the Inter-EPN parallelization algorithms. The streaming data is directed to each EPN hosted in the virtual machines. Total number of Map tasks are proportionate to the number of EPNs hosted by the virtual machines. Each Map task performs a user-defined Map-functions and generates the intermediate key-value pair data. The intermediate events are organized on the cache within the virtual machines. Each of the virtual machine consist of key-value data pairs, whose keys are classified in to one group. The hash function is used to aggregate the events belonging to the same group across all the virtual machines and merged together.

Many implementations of the EPN parallelization are possible. The correct choice depends on the complexity and the computational dependency of the events. For example, in the ambient kitchen activity recognition, a rolling average is computed. Average is computed in windows of arriving events. The 32 events from the current window event stream are merged with 32 events in the previous window of events. Instead of one final result (average), the system needs to deliver continuous results (average) to derive the pattern for the activity recognition. The query is illustrated in 8.5. During the parallelization of the rolling average, each computing node should pass the last 64 events to the new computing node registering in the system to parallelise the event processing.

The complexity of the use case in each event processing scenario determines the criteria for parallelization. The next section describes algorithms using a decentralised (asynchronous) notification mechanism or centralised (map reduce) approach using range partitioning to improve the EPN performance. The algorithm design is defined for a few complex use cases to demonstrate the inter-EPN parallelization.

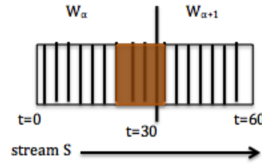


FIGURE 8.6: consequetive events split between two windows

8.4 Use case analysis

8.4.1 Consecutive Events:

Consecutive events are one of the common scenarios to identify periodic event occurrences in the incoming event streams. For example the case in the introduction, §3.6, depends on the occurrence of series of events over a period of time to deliver the prompting in the ambient kitchen. The system relies on the single or multiple occurrences of the particular event, in a particular sequence from the incoming stream of events. In this section, we define the algorithms to identify consecutive event occurrences as defined by the rule below:

Raise an alert whenever at least s, s >1, events of a specific type, say A, occur consecutively in the incoming stream.

Typically, whenever a type-A event is observed, next *(s-1)* events are type-checked to see if an alert is necessary. If window W has s type-A events, an alert is raised, the observation pointer moves the search of next s consecutive type-A events, and the processing continues. In the case illustrated in Figure 8.6, a search criteria for *s=5* is used to generate an alert. A subset of *s=5* events from window (W) is queried on the stream in search for type-A events.

Splitting a stream into multiple *windows* and letting multiple hosts process the latter in parallel is a technique that will be the focus here. So, assume that stream S is simply split into windows of fixed number, say W windows containing a predetermined number of tuples in each window. Let the windows be numbered as $W_0, W_1, ..., W_\alpha, W_{\alpha+1}, ...$ Assume also that there are n parallel processing nodes, N_0, N_1, .., N_{n-1}, and that window W_α is sent to node N_i where i = α *mod n.*

When stream splitting is opaque to tuple attributes, there is always a possibility that consecutive tuples of type-A are split across distinct windows situated in the same computing node, as shown in Figure 8.6 . All five consecutive type-A events are

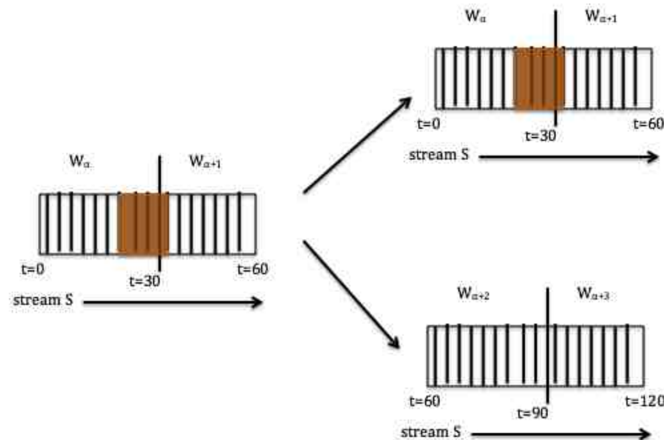


FIGURE 8.7: Exactly s, events split between two windows

placed in two adjacent windows in the computing node. Processing all the windows (W) in a single computing node resembles the raising of an alert using a centralised system for the entire event processing within one single node. Consider a scenario where windows W_α and $W_{\alpha+1}$ in Figure 8.7 are processed by distinct nodes, say N_1 and N_2. If all windows containing the consecutive events have been sent to the same node, then an alert would be raised. These considerations do not arise when a single node is involved in processing the two windows due to the following reasons:

1. Windowing of incoming tuples is basically for convenient tuple processing
2. Successive windows are always sent to the same host

To achieve the same effect in Inter-EPN parallelization, simply replicating the event processing networks (EPNs) on n distinct hosts is not enough for complex queries such as the one being dealt with here. If a generic aggregation of events is regarded to be enough, an alert will be raised by neither of the computing nodes whenever the appropriate tuple sequences are split and sent to different computing nodes.

An estimation for a simple case below shows the occurrences of the consecutive tuple in the head or the tail of two distinct computing nodes as shown in Figure 8.8. A tuple has equal probability of taking any position within a window as stream splitting is random. So, the probability of an alert being raised given that s tuples occur consecutively in the stream S is the same as the likelihood of all these s tuples being accommodated within a given window during the random stream splitting process. Assuming s=5, consider s consecutive tuples, for example in Figure 8.8, four tuples

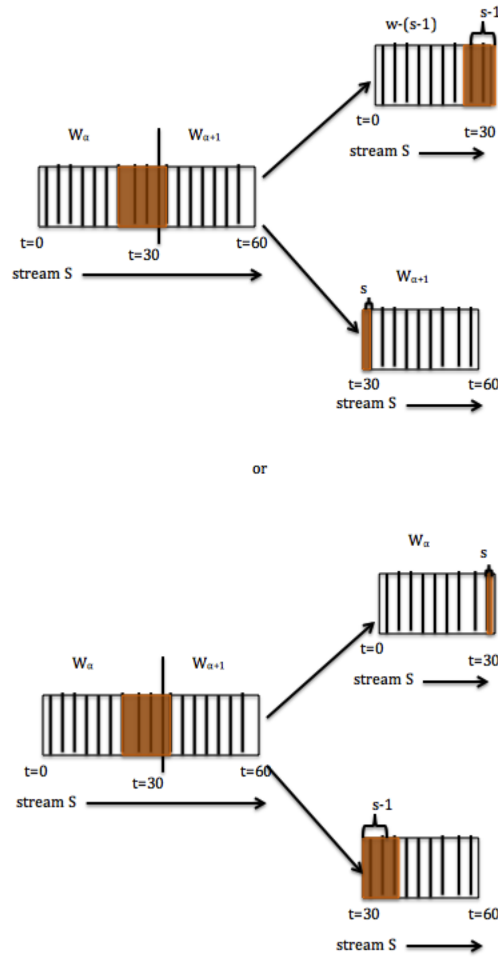


FIGURE 8.8: (s-1) events split between two windows

are placed in W_α and one tuple is placed in $W_{\alpha+1}$. If s is a modest five and a window is packed with 100 tuples, then the probability of failure due to the above distribution of events in this setup is 4% for every occurrence of s consecutive tuples. That is, the first of these tuples should be within 1 .. *(W-(s-1))* of a given window, so the remaining *(s-1)* tuples are placed in another node. So, the probability of an alarm being raised is: $\frac{(W-(s-1))}{W}$. The probability of failure for any given s consecutive occurrence is: $\frac{(s-1)}{W}$.

The three possible occurrences of the consecutive events are summarised below:

Case1 : Exactly s tuples consequetively occur in the head or tail of the window as shown in Figure 8.8. One event is placed in the tail of the window and the other events are placed in the head of the next window.

Case2 : More than s, i.e. $s+k$, tuples consequetively occur in the head or tail of the window, where $k < s$. Two choices for raising the alert in this situation are

illustrated in Figure 8.9. Under the first choice, the first of s tuples falls in the head of the window. This will ensure s tuples appearing consecutively in this window and the alert is raised. Under the second choice, the first of k tuples falls in the tail of the window and the remaining tuples fall in the head of the next window. This will lead to s tuples appearing consecutively in the next window. If *(s+k)* or more tuples occur consecutively, then

Probability of failure is $\frac{(s-1)}{W}$ - $\frac{k}{W}$, $0 \leq$ k $\leq$ s-1

If the query is for s consecutive tuples and if *2s-1* or more tuples occur consecutively, then an alert will be raised at least once in spite of inter-EPN parallelization. Note that the failure probability decreases if more than s+1 or more tuples occur consecutively and becomes zero only if (2s-1) or more tuples occur consecutively.

Case3 : Say, s is set to 50 and window size (W) of 100 tuples is used; the failure probability raises to 49%. This shows that the choice of W for a given failure probability is influenced by s which is a query parameter specified by the end user. The occurrence of 2$\bar{s}$ or more is represented as a proxy query for $\bar{s}$, which is determined using the following steps:

2$\bar{s}$ - 1 $\leq$ 50

2$\bar{s}$ $\leq$ 51

$\bar{s} \leq 25.5$

$\bar{s} = 25$

For example, the occurrence of s or more events is illustrated in Figure 8.9. There are two possible ways to split the events, where s events occur in the tail of the window and remaining *s+2* events occur in the head of the next window. Alternatively, the *s+2* events occur in the tail of the window and the remaining s events appear in the head of the next window. Whenever s events occur in the head or tail of the window (W), a search for the remaining events needs to be triggered to find the computing node which is either successor or predecessor of the processing node. If the consecutive s or more events occur in the middle of the window, the alert would be raised in the same window and there is no need to trigger a search.

The algorithm used to determine the s consecutive events is split in to two parts. **Part1** determines the starting point to trigger the search of the consecutive events.

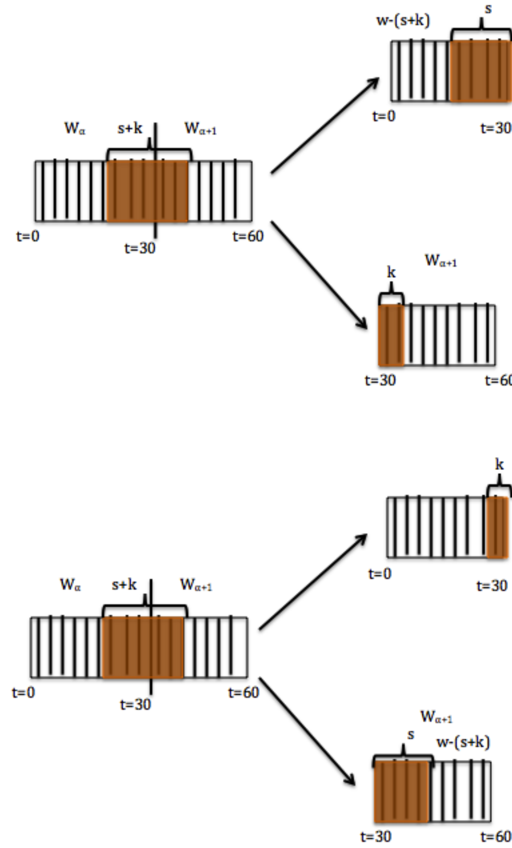


FIGURE 8.9: (s+k) events split between two windows

Part2 involves the aggregation of the events from the windows in multiple nodes shortlisted through the search operation. Let us consider C as the count for the number of consecutive occurrences of event type A. Both the parts are illustrated through the following steps as listed in the algorithm.

if Count(C) $\geq s$ **then**
  raise an alert
end if
if (Count(C) $< s$) and tuples occur either at head or tail of the window **then**
  if Count(C) $> (\bar{s} = \frac{s}{2})$ **then**
    Raise a potential alert ((H/T),C)
  else
    Raise a supportive alert ((H/T),C)
  end if
end if

Part2 is the aggregation process, executed on receiving the alert

if Potential alert (H,C_1) is received **then**

Check for the supportive alert (T,C_2) from predeccessor node
if $C_1 + C_2 \geq s$ **then**
raise alert
end if
end if
if Potential alert (T,C_1) is received **then**
Check for the supportive alert (H,C_2) from succeeding node
if $C_1 + C_2 \geq s$ **then**
raise alert
end if
end if

8.4.2 Time Limited Search

Raise an alert whenever an event say type A is followed by at least s, s≥1, non-consecutive events of specific type B, within δ time of event A.

Every occurrence of type A should trigger search for s occurrences of type B within δ time. Figure 8.10 illustrates the query for a pattern where *s=2* type B events are of interest. Whenever two consecutive type B events occur within the δ time of event A, then an alert should be raised. No alerts are raised if occurrences of type B does not occur within δ time. Two occurrences of event B are searched for after the first occurrence of type A . Type A followed by two non-consecutive type B is the main event of interest. Any type B occurrence which is beyond the δ time is of no interest and alerts are not raised. If event type A with timestamp t_i occurs, a search for event type B should be initiated by a broadcast notification with a label $\{A, t_i\}$.

In the figure 8.11, the event type A would trigger a search for the occurrence of succeeding type B with a label $\{A, t_i\}$. In this case, the windows W_α and $W_{\alpha+1}$ are split in two different computing nodes. Both type A and type B occurs in two windows. Under the parallelization, the two windows are placed in two nodes and the occurrences of the event are unknown to each other. The occurrence of the first two type B events (BI and B2) needs to be correlated with the type A event ordered by the timestamp.

In parallelization of event processing, independent occurrences of type A or B need to be carefully aggregated. Suppose the event $\{A_1, t_i\}$ is generated from Node N_i and

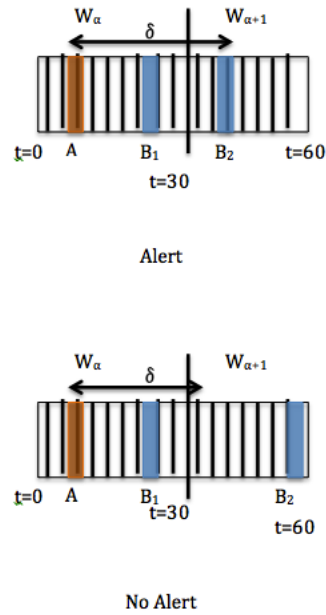


FIGURE 8.10: Search for the pattern ABB within δ time

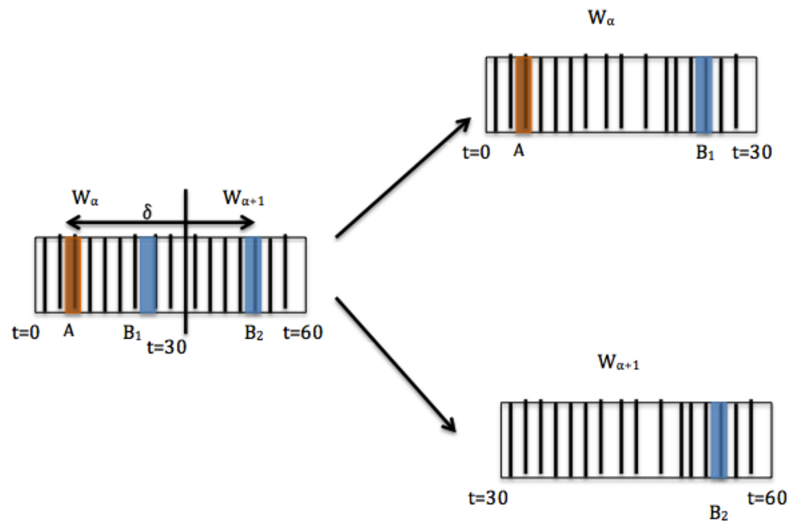


FIGURE 8.11: Search for the pattern ABB within δ time

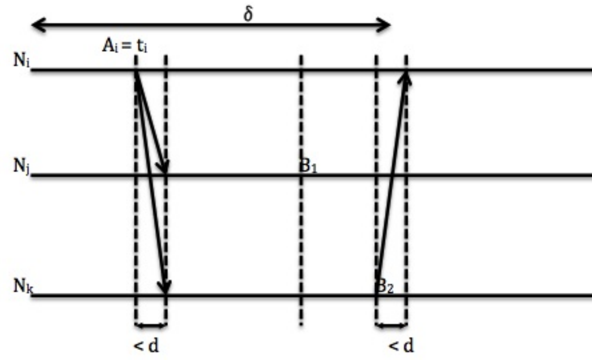


FIGURE 8.12: Illustration of communication delay for the pattern ABB within δ time

there is a worst case communication delay for the search broadcast notification to reach Node N_j. The worst case communication delay is defined as d for all messages, as illustrated in Figure 8.12. When a type A event occurs in node N_i, it needs to send a broadcast notification to all other nodes registered in the system and receive a response from other nodes registered in the system. Under a worst case communication delay, d in the network communication between the nodes, it will take an additional maximum of 2d time to send the notification and to receive the broadcast message. If s consecutive events occur in the system within the δ time, then the total time taken is $\{2d + \delta\}$.

The algorithm used for the time limited search of the non-consecutive events works in three concurrently executed parts. During the system initiation, every node N_i builds the following three parameters:

1. Search list, SL is maintained by the node and an entry is made whenever a node intitiates or receives a search.
2. Buffer b, is maintained locally in each node to store all type B events.
3. A response list, RL is maintained for each type B event stored in the buffer.

Part1 is executed whenever a node N_i encounters a type A, type B or a search request for type B.

Step1.1 Whenever type A is observed with t_i

Step1.1.1 Enter $\{\{A_i, t_i\}, N_i\}$ in to the SL

Step1.1.2 Broadcast $\{\{A_i, t_i\}, N_i\}$ to all other nodes registered in the system.

Step1.2 Whenever type A is received from other nodes, the search $\{\{A_i, t_i\}, N_j\}$ is entered in the SL where (j $\neq$ i)

Step1.3 Whenever type B is observed with t_j, enter $\{B, t_j\}$ in the buffer.

Part2 is executed on the buffer of every node N_i

For every $\{B, t_j\}$ in buffer

  if there is a $\{\{A, t_i\}, N_k\}$ in the SL and $t_i < t_j \leq t_i + \delta$ **then**

    if $\{A, t_i\}$ is in RL for the $\{B, t_j\}$ **then**

      do nothing

    else

      Inform search initiator of $\{\{A, t_i\}, N_k\}$.

      For $\{B, t_j\}$ enter the corresponding $\{A, t_i\}$ in the RL

    end if

  end if

EndFor

Part3, is executed on the SL of every node N_i

For every $\{\{A, t_i\}, N_k\}$ in the search list SL

  if $N_k = N_i$ **then**

    if s responses is received for $\{\{A, t_i\}, N_k\}$ **then**

      Alert is raised and delete the search $\{\{A, t_i\}, N_k\}$ from SL

    else if (2d+δ) time is elapsed after the entry was made into SL **then**

      Delete entry from SL

    end if

  else

    Remove $\{A, t_i\}$ after (d+δ) time following its entry in to the SL

  end if

EndFor

Part4, is executed on the buffer of every node N_i

For every entry in buffer d, check

  if $\{$B,$t_j\} >$ d **then**

    Delete the entry from buffer and RL

  end if

EndFor

8.5 Evaluation

Splitting an event stream and letting multiple hosts process portions of the stream in parallel is often seen as an effective way of improving timeliness in large-scale distributed event based systems. The Inter-EPN parallelism described in §8.2 argues that, unless some underlying integrity and semantic issues are paid attention, the approach has potential for loss of accuracy while processing even simple, not so uncommon, classes of query operators. This evaluation section examines event schemes that can be used for *fixing* such a loss of accuracy. The algorithm as described in §8.4 is used to evaluate the timeliness and efficiency of the distributed event processing. For brevity, assume that a single stream of events or tuples is continually arriving to be processed and, for simplicity, in the temporal order of event generation times.

8.5.1 Inter-EPN Parallelization of Consecutive Events:

The first use case evaluation is based on the consecutive events as in §8.4.1, the events emerging from the accelerometers carried by individuals are examined for identification of activities over a period of time. Events are distributed to multiple computing nodes. The single stream of events or tuples continually arriving to be processed, for simplicity, is ordered in the temporal sequence based on the arrival time. To introduce inter-EPN parallelization, an example is given which monitors the acceleration with respect to X, Y and Z co-ordinates through a sensor. The sensors are embedded inside the utensils in the kitchen and the events are streamed. The sensors periodically notify the measurements and a few rules are listed to highlight the problem in the inter-EPN parallelization. Consider the following definitions:

1. Three event types X, Y and Z are generated from the accelerometer.
2. Each event type is streamed at a frequency of 100 events per second. A rolling window of 64 events needs to be created. At the occurrence of every 32 event, the preceding 32 events need to be grouped.
3. Events need to be temporally ordered.

The rule for the use case illustrated in Figure 8.13 is defined as :

Compute statistics on every s *consecutive events of type X, Y and Z, where* s=*64. On the occurrence of every* s=32 *event, preceding consecutive events are grouped.*

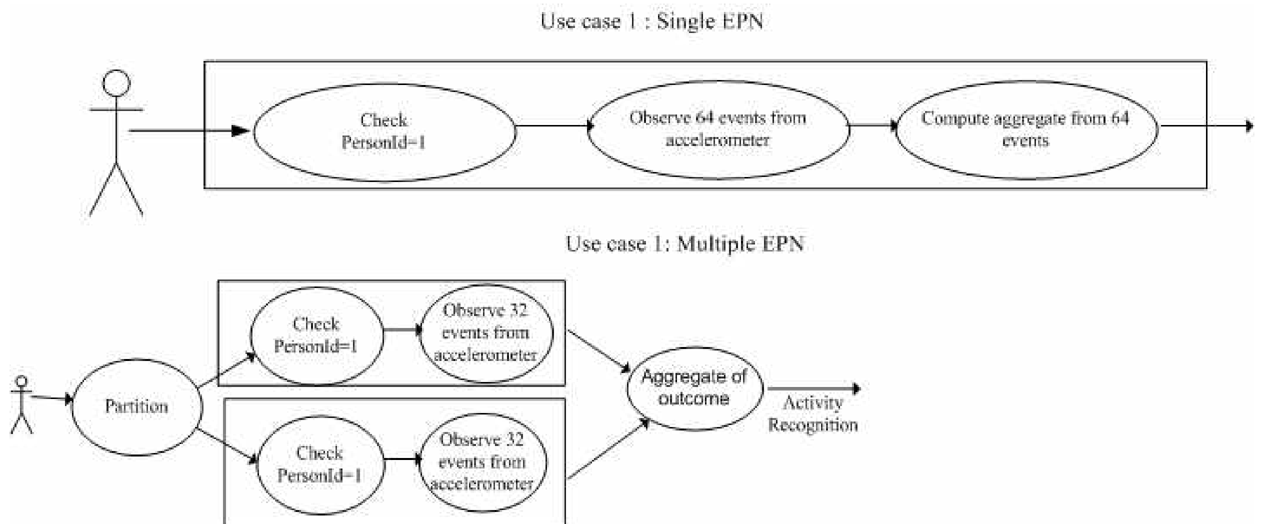


FIGURE 8.13: Illustration of event distribution

For example, the system starts with event arrival from one person and scaled to 100 persons sending 100 events per second approximately. This leads to the replication of the EPNs, distribution of the incoming events every few seconds or based on the particular tuple counts and aggregation of the outcome. The methodology of partitioning the incoming events is described during the use case evaluation, in Figure 8.2. The aggregation of events for the stated use case can be performed using pre-determined aggregation functions. This is an example of a complex query where the distribution of the events and aggregation of results involve transfer or the maintenance of the events/state among the EPNs.

In this application, the events are timestamped (within the event processing engine) and ordered in the EPN according to the timestamps. Processing events from the source creates the derived events in the subsequent stages. The new derived events are defined by (i) the attributes and type (ii) specific sequence of events whose occurrence has lead to the complex event, and (iii) defining the new value for the event. The event processing rule for the use case in Figure 8.13 is defined as :

Compute activity recognition on the events emerging from the accelerometer grouped by a device and the source of origin.

The steps utilised to evaluate the use case are listed below:

1. Detect 32 consecutive events from three different data sources X, Y and Z . The occurrence of the thirty two consecutive events in a stated sequence is used to generate a window.

2. When 32 type X events are detected in any of the computing nodes, it broadcasts a message to all other computing nodes to search for the preceeding 32 events satisfying the temporal order. Implement the algorithm illustrated in §8.4.1). In this case, the parameter s is defined to be 32. There might be few cases where part of 32 events fall in the head or tail of the partition. The algorithm in §8.4.1) states the appropriate rules to address it.

3. Once a pattern seek is initiated, EPNs distributed in multiple computing nodes respond when the match is found. The waiting time to seek a match can be decided based on the probability for the event to happen at n^{th} attempt.

4. The pattern matching occurs whenever 32 consecutive events of type X, Y and Z are detected, which may occur in any of the EPNs placed in the distributed computing node.

Incoming data can be split from one computing node succeeding to another computing node as illustrated in Figure 8.2. For example consider a cluster of four nodes where the row keys are numbers in the range of 0 and 100. Each node is responsible for processing the events under the wrapping range. This restricts only a subset of the computing nodes to search for the occurrence of the consecutive events. All the events are restricted to check the nodes bound to a subset of the range. The number of initiated search conditions is likely to reduce under efficient partitioning.

In this use case, features in the activity recognition are computed based on the event streams using the continuous queries as illustrated in 8.14. Each continuous query embedded inside the EPN processes the events emerging from the accelerometers, placed throughout the ambient kitchen. The simplest method of evaluation is based on the event occurrence in the time domain measures. Events at any point in time or intervals between successive normal-to-normal intervals (NN) are used for the computation of statistics. Inferences on statistical variation from the precomputed pattern of activity recognition are used on a series of computed events, particularly events measured over a bound count of event window or longer bounds based on the occurrence of the primitive events (acceleration X, Y and Z in this example), which determines the activity recognition. Event selection in a stated rule is grouped based on individuals or device identifier, which bind the value of instantaneous events or complex statistical time domain computation.

```
select avg(x) as meanX, avg(y) as meanY, avg(z) as meanZ,
avg(Math.atan2(y,Math.sqrt(x*x+z*z))) as MeanPitch,
avg(Math.atan2(x,Math.sqrt(y*y+z*z))) as MeanRoll,
stddev(x) as stdDevX, stddev(y) as stdDevY,
stddev(z) as stdDevZ,
stddev(Math.atan2(y,Math.sqrt(x*x+z*z))) as StddevPitch,
stddev(Math.atan2(x,Math.sqrt(y*y+z*z))) as StddevRoll,
avg(x*x) as EnergyX, avg(y*y) as EnergyY,
avg(z*z) as EnergyZ,
avg(Math.atan2(y,Math.sqrt(x*x+z*z)) * Math.atan2(y,Math.sqrt(x*x+z*z)))
as EnergyPitch,
avg(Math.atan2(x,Math.sqrt(y*y+z*z)) * Math.atan2(x,Math.sqrt(y*y+z*z)))
as EnergyRoll,
min(curt) as l1_min, max(curt) as l1_max
from MergedStream.win:length(64)
group by deviceid output every 32 events

select (count(*)*sum(x*y)-sum(x)* sum(y))/
 ((Math.sqrt(count(*)*sum(x*x)-sum(x)* sum(x)))
* (Math.sqrt(count(*)*sum(y*y)-sum(y)* sum(y)))) as corX,
(count(*)*sum(y*z)-sum(z)* sum(y))/
((Math.sqrt(count(*)*sum(z*z)-sum(z)* sum(z))) *
(Math.sqrt(count(*)*sum(y*y)-sum(y)* sum(y)))) as corY
(count(*)*sum(x*z)-sum(z)* sum(x))/
 ((Math.sqrt(count(*)*sum(z*z)-sum(z)* sum(z)))
 * (Math.sqrt(count(*)*sum(x*x)-sum(x)* sum(x))))
from MergedStream.win:length(64)
group by deviceid output every 32 events

select (((count(*)/64) * Math.log(count(*)/64)) * count(*))
as EntropyPitch
from MergedStream.win:length(64) group by pitch, deviceid
output every 32 events

select (((count(*)/64) * Math.log(count(*)/64)) * count(*))
 as EntropyRoll,
 deviceid from MergedStream.win:length(64)
 group by roll,deviceid output every 32 events

select (((count(*)/64) * Math.log(count(*)/64)) * count(*))
as EntropyX
from MergedStream.win:length(64)
group by x,deviceid output every 32 events

select (((count(*)/64) * Math.log(count(*)/64)) * count(*))
 as EntropyY
from MergedStream.win:length(64)
 group by y,deviceid output every 32 events

select (((count(*)/64) * Math.log(count(*)/64)) * count(*))
 as EntropyZ from MergedStream.win:length(64)
group by z,deviceid output every 32 events
```

Figure 8.14: Compute 21 features from acceleration logs

8.5.2 Inter-EPN Parallelization of Time Limited Search

The second use case evaluation §8.4.2 is based on the computed events (activities) emerging from the accelerometers. The computed low-level activities (cutting, chopping, stirring, etc.,) from the accelerometer events are used for situated prompting as described in §3.6. The situated prompting varies based on the activities performed or the context of the events emerging from the accelerometers embedded inside the device. The methodology to evaluate the situated prompting is summarised in Table 3.4. On the arrival of new event, the system checks the succeeding event occurrence within the specified time limit is searched. Alerts are raised based on a successful search. During every event arrival, succeeding windows are checked for event occurrence starting from the current time of computation. When the window size is long, it will be hard to replicate the events in the same computing node. The events of each window will need a partition. The computation of events with reference to the current instance of time plays a significant role in situated prompting. The computation of the alerts to trigger the decision support system will require events from multiple windows, back referenced to minutes or hours of events starting with the current time. In data intensive applications, time limited search requires an extensive state sharing mechanism to distribute the events among multiple computing nodes. During parallelization, the events from a single EPN are distributed in multiple computing nodes, with a possibility of the distribution of events belonging to a single window in multiple machines.

It is clear that the performance of the EPNs is limited by the hardware capabilities. In a system with a single EPN, a user sends a query to implement the application logic and the events are mapped against the queries to send the appropriate prompt. The use case has few defining characteristics:

1. The EPN continuously emits events maintaining the rules stated in the use case.
2. The EPN by definition operates on the incoming data and stores only a minimal amount of event in the memory.
3. The EPN responds to produce alerts stated in the rules every millisecond, seconds or hours as dictated by the application logic.

Raise an alert whenever a specific type B occurs within 5 min from the occurrence of event A Raise an alert whenever a specific type C occurs within 5 min from the

occurrence of event B Raise an alert whenever a specific type D occurs within 5 min from the occurrence of event C

Consider event type A occurs at time δ, event type B occurs within time δ+t1, event type C occurs within time δ+t1+t2. There is a sequence of event occurrence where each stage is progressed through alerts raised by the successful condition. Successful occurrence of an event prompts an audio signal with instructions in the kitchen prompting the user to progress with the next action. The table to prepare a dish has an approximate 10 stages. The number of stages can be increased as per the application requirement. However, the core algorithm or analysis of the event resembles §8.4.2. The whole promiting scenario can be built based on n number of time limited search. If a certain attribute is found to be erroneous, temporal retrofitting needs to be done based on the merged data from the succeeding time stamps. The recent history and forthcoming events are all necessary to compute this.

Consider the event type B at time t occurs after the event type A. There are number of occurrences when both events needs to be correlated for a period of time. This leads to the correlation of events in terms of the time and identity. The data model is assumed to compute the prompting and is stored as a template available to all computing nodes. Under a parallelization, the algorithms to compute the time limited search are listed below:

1. Each EPN is provided with the search function with the list of event sequences.
2. T_x is the timestamp of the observed event at a given instance of time.
3. Occurrence of the event initiates the search for the next event type. The algorithm in §8.4.2 is used for the search
4. The first computing node which observes the event acts as the querying node and broadcasts the message about its status to all other nodes in the system.
5. Other nodes in the system start recording the events specific to the search. On successful event occurrence, the querying node computes the aggregation.

Efficient processing of the occurrence of events in relation to the time is the key issue. Granularity is an important aspect of the analysis to enable the exploration, understanding and subsequent raising of the alerts after the occurrence of the event. To analyse the events, length of window of a given dataset is considered to be finite. When a given event does not match, that event will eventually leave the window

for that particular partition (attribute id, in this example) and be deleted from the memory. When a match occurs, the partition and remaining active matches for the same partition are removed.

8.5.3 Inter-EPN Parallelization Experiments Outcome

Both the use case 1 and 2 were designed as EPNs and instantiated in multiple computing nodes. The outcomes were propagated to the final stage. Under stateful operation as illustrated in rule (3.3), every EPN requires the events for 'n' number of windows so that the earlier events affect the processing of events in the later stages. The stateless operations depend on the current occurrence and the events can be moved to disk or back up systems after the completion of processing. Parallelization of the stateless EPNs can be achieved using distribution and aggregation of events in any number of computing nodes. However, to achieve the inter-EPN parallelization in stateful operations, distribution and aggregation is pursued using the algorithms described in 8.4. The semantic integrity of the results related to EPNs executed on one computing node and multiple computing nodes were checked and evaluated carefully.

The EPN prototype is implemented using java SE SDK 1.6 and experiments were conducted using the OpenJDK run time environment. The prototype for the pattern matching in the EPN is executed using an open source event processing engine called ESPER. The prototype consists of the pattern matching to identify the unique occurance of the event along with the statistical computations written in CQL (continuous query language). In order to replicate the real world setting, the experiments were conducted in six computing nodes. Three nodes were deployed with the EPNs and the other three were used as event producers. The implementation uses Java NIO sockets for communication between the event producers and the EPNs. The broadcasting of the messages across the EPNs was done using the message queues.

The distributed event processing results are illustrated in Figure 8.15. The first set of experiments was started using two computing nodes. After the semantic verification of the results, the experiments were conducted in the machines with three computing nodes. In both experiments, the distributed event processing shows a decreased processing time compared to the centralised event processing. The results of the single node event processing were verified with multiple node results and found to be same. The arrival rate of events was increased from 1000 to 6000 events per second, from multiple data sources.

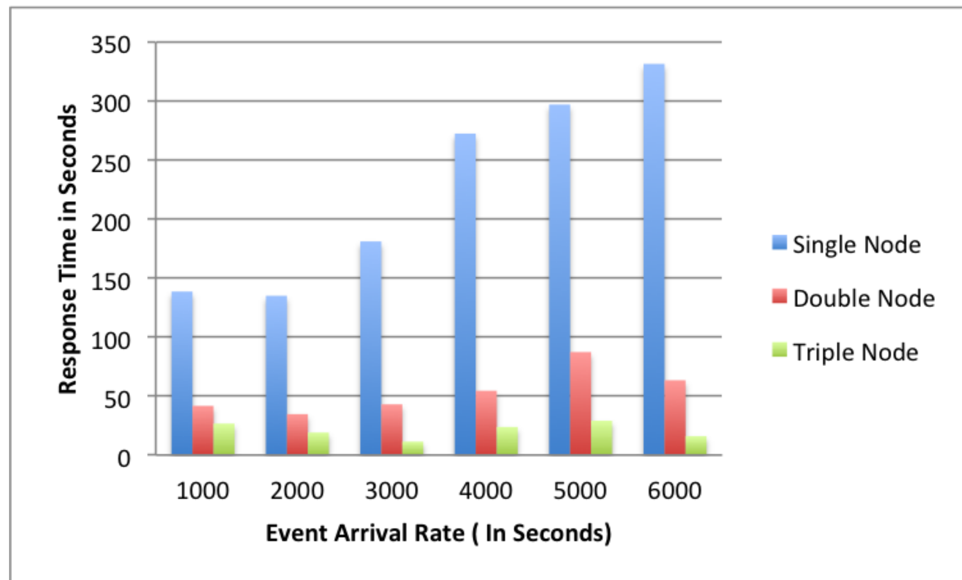


FIGURE 8.15: DEP performance results (three nodes)

A few main observations from the experiment results as illustrated in figure 8.15 are listed below:

1. Under high incoming arrival rate, the time taken by a single node is approximately five times higher than the three nodes as illustrated in 8.15.
2. Under lower arrival rate, the processing times are very similar. So maintaining the minimum number of computing nodes will be ideal for the inter-EPN parallelization.
3. Using the stated algorithms implemented in the inter-EPN parallelization, the semantic verification of the results were tested in terms of the alerts issued from the system. The experiments demonstrated a similar query result. For example, the processing of three non-consecutive events delivered the same situated prompt when executed in the single computing node or multiple computing nodes.
4. In the instances of multiple EPN receiving 2000 events and the single EPN receiving 2000 events, the response time variation is observed. Even though the event arrival is similar for both cases, the response time difference varies due to the queuing of events. The single EPN has a much higher events queue compared to the multiple EPNs.

8.5.4 Observations

During the Inter-EPN parallelization, the utilization of more hardware or computing nodes sounds attractive in terms of response time or costs of deployment. However, there is a *hidden serialisation* cost. Sometimes an alert cannot be raised until two consecutive windows are processed at distinct hosts. With the overhead of stream splitting, conjugating, and communication delays, this synchronisation can add delays for a few alerts. One should at least be aware that there are other QoS metrics that need to be looked at closely if a value based judgement is to be made. These observations are valid even for *deterministic* queries in the conventional sense of its use. The commodity servers or virtual machines in the cloud decrease the capital costs (CAPEX) for any application. However, the overheads involved in such deployment should be analyzed in greater detail.

1. Read operations: The Inter-EPN parallelization, the utilization of more hardware or computing nodes sounds attractive in terms of response time or costs of deployment. One should at least be aware that it involves partitioning and replicating the streaming data. For instance, N is the read operation and S the number of computing nodes spread across the cloud to process the events, the load due to read operations are N/S. Each server needs to process fractions of read operations. If 1000 events are sent every second in 32 servers, each server processes 31.25 events/second instead of processing 1000 events/second in one computing node.

2. Write Operations: Each server in the Inter-EPN parallelization needs to deal with the writes of both the streaming data and their replicates. The writes are not homogeneously distributed; each computing node needs to be provisioned for increased number of writes mediated by the algorithms, which are designed specific to the use case. However, in most of the event distribution scenarios the read operations are more frequently compared to the write leading to the reduced traffic by reads. The replication of the data needs to be designed in a manner to reduce the excessive write operation during event distribution.

3. Memory: Memory requirements are the most limiting factor in the centralized or single computing node based event processing system. In order to achieve improved response time, it is common to minimize disk I/O in favor of memory I/O. Solutions such as memcache, denormalisation and SQL caching are a few options designed to address the memory issues and reserved for future work.

4. Network Traffic: The bandwidth requirements for the replication of the data during Inter-EPN parallelization would require careful design to replicate EPNs on many computing nodes compared to one server in centralized event processing.

5. Load Balance: Although different EPNs will require varying amounts of processing time, idle EPN can be assigned to the next batch of the incoming stream and all EPNs need to be given optimum work load based on the response time. The optimum load for the EPN can be predicted using the response time optimisation and reconfiguration algorithm discussed in Chapters 7 and 6.

8.6 Conclusion

To conclude this chapter, Inter-EPN parallelization can lead to performance improvements only when it is carefully engineered by taking into account several factors such as tuple arrival rates and the parameters specified in the queries. In event processing, the logical sequence of the events involves customizable selection policies, parameters and aggregates appearing over a long period of time. If attention is not paid to underlying issues, its relative advantages (over centralised processing) may not be significant in addition to potential for loss of accuracy. Algorithms were designed for the few complex event processing scenarios and the parallelization of the EPNs was evaluated to demonstrate the semantic integrity of the event processing.

Chapter 9

Challenges and Future Work

9.1 Introduction

Virtualization in cloud computing is the key enabler to drive the ability of hardware to scale according to the arrival of events. The cost effective, on demand approach in the virtualization helps to reduce the capital expenditure. The ability to scale the resources based on demand improves the operational efficiency and agility of the system. Under virtualization, a single piece of hardware will have the capability of running multiple independent resources in terms of operating systems with distinct management capabilities allowing the utilization of the underlying platform resources. The provisioning of the VMware resources is dynamically configured based on the thresholds configured by the service providers. Underlying hardware usage and the networking device state is dependent on the CPU utilization, application state and networking I/O. The optimization of the resources to be provisioned or released is designed based on the acceptable limits of the underlying server. This chapter discusses the opportunities of extending the work on the response time prediction, reconfiguration and inter-EPN parellelization in challenging real time and batch processing applications using cloud computing.

9.2 Virtualisation

The core objective in virtualisation is to provide performance isolation, scheduling priority, memory demand, network access and disk access for all the concurrently running resources spawned in a single piece of hardware. The enabling factor for

cloud computing is the virtualization techniques provided by hypervisors such as XEN [Barham et al.], VM Ware ESXI, Hyper-V and KVM. The hypervisors allow multiple operating systems and applications to run simultaneously on the hardware. To offer linear scalability to the applications, VMware provides the following tasks:

- Resource utilization
- Scalability
- High availability

9.2.1 Scalability

Technologies to manage scalability in cloud computing as a part of a software defined environment (SDE) is an emerging area of research. Resource-aware and workload management solutions enable dynamic allocation of virtual machine infrastructure in accordance to the incoming event streams or proportionately adapt to the arrival rate of the streams. Design of automated resource management can be built on the reconfiguration algorithms in section §7.2 focusing on major issues such as

- Cost of the cloud computing utilization
- Policy driven resource scheduling

9.2.1.1 Cost of cloud computing utilization

Spot instances are one of the emerging popular services in cloud computing. Spot instances are the excess computing nodes or instances of the elastic compute cloud facility, which are available in real time through bidding. Innovative applications can be built based on the reconfiguration algorithms. At regular intervals, an application should register internal demand for computing resources and search for the cheapest available compute resource in real time. Increasing the percentage of application execution in the cheapest available computing node would drastically reduce the expense of the event processing. Quantitative research projects that require hundreds of terabytes of data need many hours of compute power. Highly prevalent options are to use on-demand instances in cloud computing. However, to utilize spot instances, custom-built algorithms to optimally bid and consume cost-effective resources are essential. The extension of cost centric reconfiguration algorithms can be utilized in

the batch processing scenarios as well to map multi-hour jobs in to smaller sub-jobs, queue them based on the memory and I/O requirements, monitor the spot price of the compute resource and run the outstanding jobs in the most cost effective way using cloud computing. Cloud computing has enabled shrinking of the multiple years' work to a few hours, however, the automated hire of the spot instances could drastically reduce the cost of the resource utilization.

9.2.1.2 Policy driven scheduling

Automated resource management should choose the appropriate servers on which to place the event processing networks based on the policies designed by the application. For example, in an indoor localization of the elderly people inside a care home, the personal data about the individual sensors used by the elderly people reside in a private server. The encrypted private information on the health conditions or the identity is abstracted within the private access residing in a private cloud. The movement or history of the activity logs of elderly residents is a stream of events from multiple sensors, which can be processed in a public cloud. The criteria on the utilization frequency for each virtual machine in terms of the context of events, privacy requirements, cost, security criteria and the data management policy could act as a decision-making criteria for the management of the resources in the automated manner. For example, the computing nodes can be made available to hire as determined by the algorithm's security and privacy policy. The configuration scheduler in Section §7.2 can be extended by incorporating additional features related to the security and privacy of the underlying applications.

In an automated scenario, the deployments may require more than one environment such as private cloud, public cloud or both. The configuration scheduler can be defined with a directory of event processing channels where multiple networks can be interconnected. A network interface can be provided to connect with individual computing nodes in the system. In example 9.1, machine A and B can communicate with each other, machines C and D can communicate with each other but machine A cannot communicate with machines C or D unless it goes through B which has an interface in both networks. The Network X.X.X.0 could be a private network operating internally within an enterprise and the Network Y.Y.Y could be a group of computing nodes operating in the public cloud. The configuration scheduler in this case maintains the directory of available computing nodes in both the private and the public cloud. During the scaling up of the network, the highest preference can be given to the computing nodes available in the same network and security policy. This can help

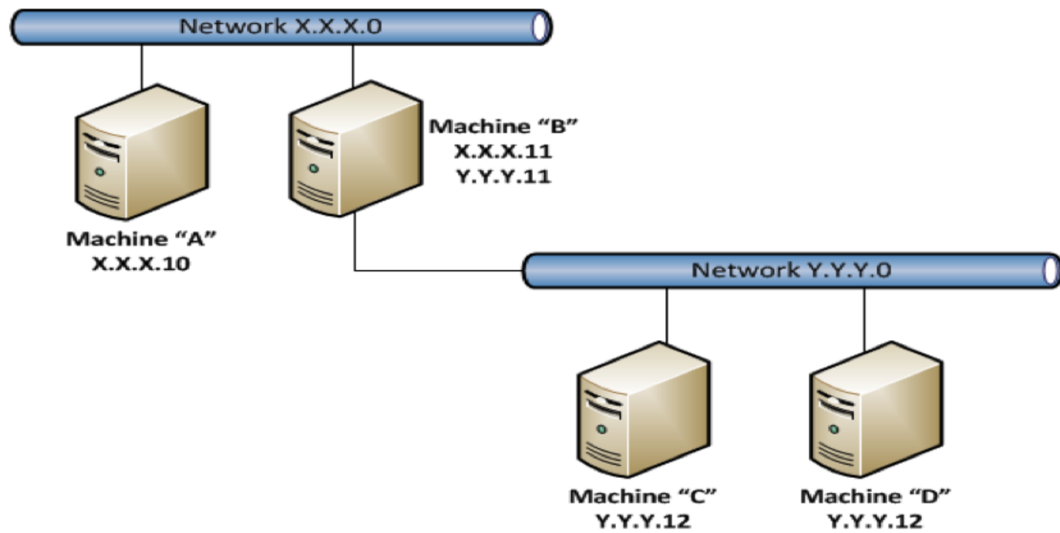


FIGURE 9.1: Event Processing Channels of communication

the system to utilize the computing nodes optimally within the internal environment and once sufficient resources are unavailable, external hiring could be done based on a pay-as-you-go basis. As part of creating the network of computing nodes, multiple virtual networks can be created based on the requirements. All subsequent networks can be set up dynamically based on the incoming event stream traffic. As a part of automation, live migration of the virtual machines or images of EPNs hosted in the virtual machines can be ported as in Clark et al. (2005). Live migration with the automated reconfiguration could reduce the service time and improve the overall system performance.

9.3 Resource Utilisation

The applications running inside one virtual machine are expected to be independent of the co-located applications running inside another virtual machine. The performance isolation [Koh et al. (2007), Nathuji et al. (2010)] and CPU fair sharing [Kazempour et al. (2010), Liu et al. (2010)] in a virtualized environment address the resource contention issues. However, the real time streaming events complexities in terms of performance isolation and resource contention in terms of dynamism exhibited by the system in terms of event producers, event consumers, state management and context. The assumption of the performance isolation in the virtualization fails under high I/O requests and heavy CPU intensive jobs. The resource utilisation policies can be extended targeting specifically towards each hypervisor and constraints associated

with it. This will lead to some generic policy formulation to handle the resource utilisation.

9.4 High Availability

The configuration scheduler in this research is built on a single server, which is meant to be prone to crashes and intrusions. An approach specified in FORTRESS [Clarke and Ezhilchelvan (2010)] for adding resilience capability along with the primary back up system could act as an extension to this research.

9.5 Oscillatory behaviour of EPN

Oscillatory behaviour of the EPN can be caused by many factors such as bursty workload, instable resource allocation for the computing nodes, the workload in the shared hardware, network, input or output fluctuations in the host environment and many other reasons. This work is carried out based on the assumptions that stable CPU and memory allocation is provided to reconfigure the EPNs across multiple nodes. There is plenty of scope to improve the oscillatory behaviour of the EPN movement.

9.6 Conclusion

Engineering aspects of event processing are commonly focused around timeliness and scalability. This research focuses on the response centric scalability along with the parallelization of the event processing networks. This research could provide a solid foundation to understand the challenges and basics behind scalable and responsive event processing. A few areas of future work were discussed in this chapter such as scalability, availability and resource utilisation. However, given the era of *Big Data*, progression can be made in terms of extending this research in to a big data platform for event processing.

The novel approach described here deploys a collection of EPNs on cloud platforms in such a manner that the response times can be ensured to meet specified targets even when event arrival rates fluctuate over time. The novelty of the approach lies in

predicting response times based only on prior calibration and periodic measurements of event arrival rates and end-to-end latencies of EPNs; there is no need for intrusive, low-level measurements at operator-level within EPNs.

Experiments confirm that prediction is reasonably accurate and the heuristic for selecting appropriate configurations is effective. Periodic reporting of arrival rates and response times by EPNs is the only overhead imposed. This small but inevitable running overhead and the overhead of executing a heuristic selection algorithm make the approach a highly scalable one in managing a large number of EPNs with response time constraints on a cloud platform.

Centralized event processing systems have a single node bottleneck for maintaining key performance indicators. To decrease the per-node tuple processing time and to increase the overall throughput, each node of a distributed event processing system requires a scalable, responsive monitoring and redeployment framework. Any system with a static number of computing nodes might experience under provisioning (i.e. the overall computing power does not handle the performance requirements) or over provisioning (i.e. the number of allocated nodes operating below their full capacity meeting the target performance metrics). The event based system was designed and built to address the under provisioning or over provisioning of computing resources using cloud computing. Event processing networks were designed as directed acyclic graphs. The challenge is to build a response centric EPN placement, across multiple computing nodes. The following objectives were addressed:

1. The response time of each event processing network under varying event flows was theoretically modeled and predicted using M/G/1 queuing theory.

2. Generic reconfiguration algorithms were designed and tested to place the EPN in a particular computing node. The predicted response time of each event-processing network was used to reconfigure the EPN across multiple computing nodes in the cloud.

3. The EPNs were dynamically increased, decreased or moved across multiple computing nodes using the predicted response time.

4. In certain cases, where the data is distributed across multiple EPNs, the resulting tuples require aggregation. This was studied using inter-EPN parallelization. The results from the inter-EPN parallelization were tested for the equivalent outcome from the centralized execution. The inter-EPN parallelization was analyzed using a wide variety of streaming data.

5. The response time of the inter-EPN parallelization was evaluated between the centralized and distributed placement maintaining the semantic integrity of the results.

In this research a novel response time prediction for the event processing networks is designed, implemented and evaluated. The inter-EPN parallelization strategy implemented through split and process principles, maintains the semantic integrity and avoids computing bottlenecks.

Bibliography

. *Workflow Management: Models, Methods, and Systems.* MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-01189-1.

Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. Springer, 2004.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010. ISSN 0001-0782.

Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. *SIGMOD Rec.*, 2000.

B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Data Engineering, 2004. Proceedings. 20th International Conference on*, 2004.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *IN PODS*, pages 1–16, 2002.

Shivnath Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, 2005.

Magdalena Balazinska, Hari Balakrishnan, and Mike Stonebraker. Contract-based load management in federated distributed systems. In *In NSDI Symposium*, 2004.

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*

Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 2003.

Andrey Brito, Andre Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof Fetzer. Scalable and low-latency data processing with stream mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, 2011.

Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. VLDB Endowment, 2003.

Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03. ACM, 2003.

Sirish Chandrasekaran and Michael J. Franklin. Streaming queries over streaming data. In *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB Endowment, 2002.

Jianjun Chen, David J. Dewitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *In SIGMOD*, pages 379–390, 2000a.

Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. *SIGMOD Rec.*, 2000b.

Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uur etintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003a.

Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, U?ur etintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *In CIDR*, 2003b.

Christopher Clark, Keir Fraser, Steven H, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, pages 273–286, 2005.

Dylan Clarke and Paul D. Ezhilchelvan. Assessing the attack resilience capabilities of a fortified primary-backup system. 2010.

Chuck Cranor, Theodore Johnson, and Oliver Spataschek. Gigascope: a stream database for network applications. In *In SIGMOD*, pages 647–651, 2003.

Gianpaolo Cugola and Alessandro Margara. Low latency complex event processing on parallel hardware. *J. Parallel Distrib. Comput.*, 2012.

Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. Towards expressive publish/subscribe systems. In *In Proc. EDBT*, pages 627–644, 2006.

Alan Demers, Johannes Gehrke, and Biswanath P. Cayuga: A general purpose event monitoring system. In *In CIDR*, pages 412–422, 2007.

Markus Dhring, Lars Karg, Eicke Godehardt, and Birgit Zimmermann. The convergence of workflows, business rules and complex events - defining a reference architecture and approaching realization challenges. In Joaquim Filipe and Jos Cordeiro, editors, *ICEIS (3)*, pages 338–343. SciTePress, 2010. ISBN 978-989-8425-06-5.

EsperTech. CEP Software. `http://www.espertech.com/esper/`. [Online; accessed 17-August-2015].

Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010. ISBN 1935182218, 9781935182214.

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 2003.

P.Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35:114–131.

Michaël Gabay and Sofia Zaourar. Variable size vector bin packing heuristics - Application to the machine reassignment problem. September 2013.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Erol Gelenbe and Isi Mitrani. *Analysis and synthesis of computer systems*, volume 4. World Scientific, 2010.

Google. Dremel: Interactive Analysis of Web-Scale Datasets . http://research.google.com/pubs/pub36632.html. [Online; accessed 17-August-2015].

Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008. ISSN 0146-4833.

Eric Griffis, Paul Martin, and James Cheney. Semantics and provenance for processing element composition in dispel workflows. In *Proceedings of the 8th Workshop on Workflows in Support of Large-Scale Science*, WORKS '13. ACM, 2013.

Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Trans. Parallel Distrib. Syst.*, 23(12):2351–2365, December 2012. ISSN 1045-9219.

Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, and Patrick Valduriez. Streamcloud: A large scale data streaming system. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*, Washington, DC, USA, 2010. IEEE Computer Society.

Daniel Gyllstrom, Eugene Wu, Hee jin Chae, Yanlei Diao, Patrick Stahlberg, and Gordon Anderson. Sase: Complex event processing over streams. In *In Proceedings of the Third Biennial Conference on Innovative Data Systems Research*, 2007.

Souleiman Hasan, Sean O'Riain, and Edward Curry. Approximate semantic matching of heterogeneous events. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12. ACM, 2012.

Thomas Heinze, Zbigniew Jerzak, André Martin, Lenar Yazdanov, and Christof Fetzer. Fault-tolerant complex event processing using customizable state machine-based operators (demo). In *Proceedings of the 15th International Conference on Extending Database Technology*, EDBT '12, pages 590–593, New York, NY, USA. ACM. ISBN 978-1-4503-0790-1.

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association, 2010.

Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. On the performance variability of production cloud services. In *Proceedings of the 2011 11th IEEE/ACM International*

Symposium on Cluster, Cloud and Grid Computing, CCGRID '11, pages 104–113, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4395-6.

Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. Aash: An asymmetry-aware scheduler for hypervisors. *SIGPLAN Not.*, 2010.

Younggyun Koh, Rob Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An analysis of performance interference effects in virtual environments. In *In Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS*, 2007.

L. T. Kou and G. Markowsky. Multidimensional bin packing algorithms. *IBM J. Res. Dev.*, 21(5):443–448, September 1977. ISSN 0018-8646.

Heiko Koziolek. Towards an architectural style for multi-tenant software applications. In *Software Engineering*, 2010.

Rouven Krebs, Christof Momm, and Samuel Kounev. In *Proceedings of the 8th ACM SIGSOFT International Conference on the Quality of Software Architectures (QoSA 2012)*. ACM Press, 2012.

Alok Kumbhare, Yogesh Simmhan, and Viktor K. Prasanna. Exploiting application dynamism and cloud elasticity for continuous dataflows. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 57:1–57:12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2378-9.

Geetika T. Lakshmanan, Ying Li, and Rob Strom. Placement of replicated tasks for distributed stream processing systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 128–139. ACM, 2010. ISBN 978-1-60558-927-5.

N. Lesh and M. Mitzenmacher. Bubblesearch: A simple heuristic for improving priority-based greedy algorithms. *Inf. Process. Lett.*, 97(4):161–169, February 2006. ISSN 0020-0190.

Ming Li, Mo Liu, Luping Ding, Elke A. Rundensteiner, and Murali Mani. Event stream processing with out-of-order data arrival. In *Proceedings of the 27th International Conference on Distributed Computing Systems Workshops*, ICDCSW '07, pages 67–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2838-4.

Ling Liu, Calton Pu, and Wei Tang. Continual queries for internet scale event-driven information delivery. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 11:610–628, 1999.

Zhaobin Liu, Wenyu Qu, Weijiang Liu, and Keqiu Li. Xen live migration with slowdown scheduling algorithm. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT), 2010 International Conference on*, pages 215–221, Dec 2010.

David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201727897.

David C. Luckham and Roy Schulte. Event processing glossary - version 1.1. Online Resource. `http://complexevents.com/2008/08/31/event-processing-glossary-version-11/`, 2008.

Yiduo Mei, Ling Liu, Xing Pu, and Sankaran Sivathanu. Performance measurements and analysis of network i/o applications in virtualized cloud.

Microsoft. CEP Software. `https://msdn.microsoft.com/en-us/sqlserver/ee476990.aspx`. [Online; accessed 17-August-2015].

Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system, 2003.

Arijit Mukherjee and Paul Watson. Adding dynamism to ogsa-dqp: Incorporating the dynasoar framework. In *In Distributed Query Processing, CoreGrid Workshop on Grid Middleware, EuroPar 2006*.

Vinod Muthusamy, Haifeng Liu, and Hans-Arno Jacobsen. Predictive publish/subscribe matching. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. ACM, 2010.

Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: Managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, 2010.

Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10. IEEE Computer Society, 2010.

Patrick Olivier, Guangyou Xu, Andrew Monk, and Jesse Hoey. Ambient kitchen: Designing situated services using a high fidelity prototyping environment. In *Proceedings of the 2Nd International Conference on PErvasive Technologies Related to Assistive Environments*, PETRA '09, pages 47:1–47:7, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-409-6.

Chris Olston, Jing Jiang, and Jennifer Widom. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03. ACM, 2003.

Oracle. CEP Software. `http://docs.oracle.com/cd/E11882_01/server.112/e17069/strms_over.htm#STRMS249`. [Online; accessed 17-August-2015].

A. Osman and H. Ammar. Dynamic load balancing strategies for parallel computers. In *Computers, International Symposium on Parallel and Distributed Computing (ISPDC*, 2002.

Cuong Pham and Patrick Olivier. Slice and dice: Recognizing food preparation activities using embedded accelerometers. In Manfred Tscheligi, Boris Ruyter, Panos Markopoulus, Reiner Wichert, Thomas Mirlacher, Alexander Meschterjakov, and Wolfgang Reitberger, editors, *Ambient Intelligence*, volume 5859 of *Lecture Notes in Computer Science*, pages 34–43. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-05407-5.

Eduardo Pinheiro, Ricardo Bianchini, Enrique V. Carrera, and Taliver Heath. Compilers and operating systems for low power. chapter Dynamic Cluster Reconfiguration for Power and Performance, pages 75–93. Kluwer Academic Publishers, Norwell, MA, USA, 2003. ISBN 1-4020-7573-1.

B. Satzger, W. Hummer, P. Leitner, and S. Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 348–355, July 2011.

SiDE. SiDE EPSRC Project. `http://www.side.ac.uk/`, note =.

Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. 1996.

Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *In USENIX*, pages 13–24, 1998a.

Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *In USENIX*, pages 13–24, 1998b.

Chunqiang Tang, Malgorzata Steinder, Michael Spreitzer, and Giovanni Pacifici. A scalable application placement controller for enterprise data centers. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 331–340. ACM, 2007. ISBN 978-1-59593-654-7.

Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003. ISBN 0-12-722442-4.

E. Thomaz, T. Plötz, I. Essa, and G. Abowd. Interactive techniques for labeling activities of daily living to assist machine learning. In *Proceedings of Workshop on Interactive Systems in Healthcare*, 2011.

R. Tolosana-Calasanz, J. Angel Baares, C. Pham, and O. Rana. End-to-end qos on shared clouds for highly dynamic, large-scale sensing data streams. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 904–911, May 2012.

Viet Tran, Ondrej Habala, Branislav Simo, and Ladislav Hluchy. Distributed data integration and mining. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*. ACM, 2011.

W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003. ISSN 0926-8782.

Anthony Velte and Toby Velte. *Microsoft Virtualization with Hyper-V*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 2010. ISBN 0071614036, 9780071614030.

Sai Wu, Vibhore Kumar, Kun-Lung Wu, and Beng Chin Ooi. Parallelizing stateful operators in a distributed stream processing system: How, should you and how much? In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12. ACM, 2012.

Yongluan Zhou, Beng Chin Ooi, Kian lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system, 2006a.

Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In Robert Meersman and Zahir Tari, editors, *OTM Conferences (1)*, volume 4275 of *Lecture Notes in Computer Science*, pages 54–71. Springer, 2006b. ISBN 3-540-48287-3.

Yunyue Zhu, Dennis Shasha, and Yunyue Zhu Dennis Shasha. Statstream: Statistical monitoring of thousands of data streams in real time. In *In VLDB*, pages 358–369, 2002.

Birgit Zimmermann and Markus Doehring. Patterns for flexible bpmn workflows. In *Proceedings of the 16th European Conference on Pattern Languages of Programs*, EuroPLoP '11, pages 7:1–7:9, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1302-5.