Technical Report

UCAM-CL-TR-783
ISSN 1476-2986

Number 783

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Distributed Complex Event Detection for Pervasive Computing

Dan O'Keeffe

July 2010

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2009 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St. John's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Pervasive computing is a model of information processing that augments computers with sensing capabilities and distributes them into the environment. Many pervasive computing applications are *reactive* in nature, in that they perform actions in response to events (i.e. changes in state of the environment). However, these applications are typically interested in high-level *complex events*, in contrast to the low-level primitive events produced by sensors. The goal of this thesis is to support the detection of complex events by filtering, aggregating, and combining primitive events.

Supporting complex event detection in pervasive computing environments is a challenging problem. Sensors may have limited processing, storage, and communication capabilities. In addition, battery powered sensing devices have limited energy resources. Since they are embedded in the environment, recharging may be difficult or impossible. To prolong the lifetime of the system, it is vital that these energy resources are used efficiently.

Further complications arise due to the distributed nature of pervasive computing systems. The lack of a global clock can make it impossible to order events from different sources. Events may be delayed or lost en route to their destination, making it difficult to perform timely and accurate complex event detection. Finally, pervasive computing systems may be large, both geographically and in terms of the number of sensors. Architectures to support pervasive computing applications should therefore be highly scalable.

We make several contributions in this dissertation. Firstly, we present a flexible language for specifying complex event patterns. The language provides developers with a variety of parameters to control the detection process, and is designed for use in an open distributed environment. Secondly, we provide the ability for applications to specify a variety of detection policies. These policies allow the system to determine the best way of handling lost and delayed events. Of particular interest is our 'no false-positive' detection policy. This allows a reduction in detection latency while ensuring that only correct events are generated for applications sensitive to false positives. Finally, we show how complex event detector placement can be optimized over a federated event-based middleware. In many cases, detector distribution can reduce unnecessary communication with resource constrained sensors.

To my family

Acknowledgements

To begin with, I would like to thank Jean Bacon, my supervisor, for all her encouragement and advice during the course of my PhD. In reaching the finish line, her knowledge and understanding (and patience!) have been invaluable. I am also indebted to Ken Moody, my PhD advisor, for the guidance he has given me throughout the time of my PhD.

During my PhD I was fortunate enough to be a member of the Opera Group, and my research owes a debt to the many interesting discussions, talks, and ideas I was exposed to as a result. In particular, I would like to thank Salman Taherian, Peter Pietzuch, Eiko Yoneki, Dave Eyers, Jat Singh, Lauri Pesonen, Sriram Srinivasan, Pedro Brandao, Dave Evans, Dave Ingram, Luis Vargas, and Samuel Kounev for all their help over the years. I would like to also thank Salman, Dave Eyers, Ram, Jat, Pedro, and Eiko for proofreading chapters of my thesis.

I would like to thank the Cambridge European Trust, the Computer Laboratory, and Marconi for funding my research. I am also grateful to my college tutor Helen Watson and all the other staff at St John's College for their support and assistance.

Finally, I would like to thank my family, and especially my parents, for their support and encouragement. Without them it would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Pervasive Computing

Advances in micro-electro-mechanical systems (MEMS) continue to reduce the size and cost of sensor devices. Nevertheless, sensors capable of monitoring an ever-wider variety of real-world phenomena are becoming available. Together with Moore's law, these trends enable the deployment of small, cheap, computational devices, augmented with sensing and communication capabilities, into the environment.

The ability to embed such sensors in the environment has resulted in the vision of *pervasive computing*, where software applications are aware of environmental conditions relevant to their operation, and tailor the functionality they provide as these conditions change. Thus pervasive computing extends existing applications with the ability to infer, from data generated by sensors, contextual information about users and the environment. Moreover, it promises a plethora of novel applications previously thought impossible due to the difficulty of gathering and disseminating accurate sensor data to interested parties.

1.1.1 Challenges

Many problems remain to be addressed before the vision of pervasive computing can become a reality. We now highlight some of the main challenges.

Interpretation of Sensor Data Pervasive computing applications may involve vast numbers of sensing devices. These devices can potentially produce an enormous volume of raw sensor data. This sensor data has the potential to overwhelm both the resources available in large-scale pervasive computing systems, and the ability of users to respond to it. In particular, raw sensor data tends to be very low-level, and must be further processed, analysed, and transformed into higher-level information in order for it to be meaningful to applications and users.

Scalability Large numbers of users may be interested in the data produced by sensors, and the task of scalably delivering relevant sensor information to users is a challenging one. Failures of sensing devices may be common, and disseminating data to users is subject to the difficulties inherent to all large scale distributed systems, such as node failures and network partitions. In addition, sensors often communicate wirelessly, further increasing the potential for failure and unreliable communications.

Heterogeneity Another challenge faced in building pervasive computing systems is heterogeneity. Typically, the devices that form a pervasive computing system vary widely in their capabilities. Devices may be limited with respect to a number of criteria, including processing, storage, bandwidth, reliability, and energy resources. In addition, pervasive computing applications can differ considerably with respect to the type of interpretation they need to perform over raw sensor data.

1.1.2 Scope

This thesis is primarily concerned with sensor-driven pervasive computing applications operating over an architecture consisting of relatively static wireless sensor networks (WSNs) connected via a fixed backhaul infrastructure. Although this architecture can handle low to moderate levels of mobility (e.g. of client devices), it is not designed for use in highly mobile environments. Nevertheless, some of the contributions this thesis makes to complex event language design and reliable detection may transfer to more mobile environments.

1.2 Distributed Complex Event Detection

Pervasive computing applications are typically data-driven, with consumers interested only in the information produced by sensors, which we refer to as *events*, and not their identity. A promising approach to supporting applications that wish to use sensor data is to build them on top of an *event-based* middleware [Pie04]. This enables a loose-coupling between the producers of events and the entities that consume them. This loose-coupling helps to improve scalability, as producers and consumers do not have to be explicitly aware of each other. An event-based middleware acts as an intermediary between the two, delivering the events generated by producers to consumers. This style of communication is typically referred to as publish/subscribe, since consumers *subscribe* to events *published* by producers.

However, in order to increase the value of sensor data, event-based middleware must convert low-level data produced by *primitive* publishers into higher-level *composite* or *complex* events. One way for middleware to achieve this is to provide a language for consumers to specify patterns of events. The middleware can then detect these high-level patterns based on the low-level data produced by sensors.

1.3 Research Statement

In this thesis we argue that detection of complex event patterns by an event-based middleware is a useful tool for supporting pervasive computing applications, but that several problems remain to be addressed in augmenting current middlewares with such a feature.

We argue that traditional approaches to performing complex event detection must be adapted to take into account the characteristics of large-scale, distributed and possibly unreliable pervasive computing systems. Existing languages are not flexible enough to cope with the wide variety of detection semantics required for pervasive computing. In addition, many of the languages proposed for complex event detection are unrealistic in their assumptions of reliability and totally ordered input event streams. A more sophisticated approach to detection is necessary.

Furthermore, the placement of pattern detectors in the network is an important issue when trying to distribute complex event detection in an efficient fashion. Placement must incorporate requirements of producers, consumers and network administrators such as low-latency, efficient use of limited bandwidth, processing, and storage resources, and constraints on the energy available to wireless sensors.

This thesis contains several contributions that aim to address these issues:

A Complex Event Language Our first contribution is a complex event detection language designed for pervasive computing applications. The language relies on a time model based on uncertainty intervals to cope with the lack of a global clock in distributed systems. It is designed to be easily decomposable in order to enable distributed placement of detectors. This is particularly useful when event sources are energy-constrained wireless sensors, as it reduces unnecessary communication. It provides a selection of operators for composing events, and also allows conditions to be specified globally (i.e. across operators) over the attributes of events. The semantics of detection can be configured extensively through the use of a variety of detection parameters. Although these parameters add complexity to the language, we believe they are necessary due to the wide variety of detection semantics we found were required by different pervasive computing applications.

Reliable Detection Our second contribution is an analysis of how complex event detection is affected by lost events, delayed events, and events with overlapping uncertainty interval timestamps. In accordance with the results of this analysis, we extend our language so that users can express, via a selection of *detection policies*, their preferred means of handling these problems. Of particular interest is a detection policy for applications sensitive to false positives. We present an implementation and evaluation of such a policy for our language. This policy results in more accurate detection in comparison to a best-effort approach, and is more efficient than a guaranteed detection policy.

Detector Placement Our last contribution is with respect to the efficient placement of composite event detectors. We propose several detector placement strategies for an event-based middleware called Hermes. Hermes forms the centerpiece of a multi-domain architecture for pervasive computing. It leverages peer-to-peer techniques to provide scalable publish/subscribe communication between a network of event brokers. However, optimizing detector placement over Hermes is a challenge due to the nature of its routing algorithm.

Our placement strategies divide into two classes: static and dynamic. Static placement is simple to implement but unlike dynamic placement, it does not migrate detectors between brokers in order to optimize overall network usage. Our dynamic strategy augments event brokers with network latency coordinates, so as to enable them to reason about where to place detectors. We evaluate the performance of these strategies in comparison to a reference strategy from the literature. The results show that the dynamic placement strategy performs well when reuse of complex events is high. Furthermore, the dynamic strategy uses a similar amount of network resources to the best of the static placement strategies. Dynamic placement is thus preferable when flexible placement of detectors is required for other reasons, such as *cross-domain* complex event detection. Cross-domain complex event detection is useful when other domains consist of energy-constrained event sources such as wireless sensor networks.

1.4 Thesis Outline

In chapter two, we give a description of the background to the thesis, and place our contributions in the context of related work. The discussion begins with an overview of pervasive computing and wireless sensor networks, including a description of their main characteristics and design principles. We then introduce the concept of middleware. This introduction summarizes several classes of middleware relevant to pervasive computing, including publish/subscribe systems, data stream management systems, middleware for context-aware applications, and middleware for wireless sensor networks. The background chapter concludes with a discussion of complex event languages. The discussion examines several representative complex event languages from the literature, and contrasts complex event detection with related concepts such as continuous queries and production systems.

In chapter three, we describe a new complex event language targeted towards pervasive computing applications. The chapter begins with a description of some motivating application scenarios. We then describe the main features of our language, including its time model, syntax, processing model and data structures. The chapter concludes with a pseudo-code description of our implementation.

In chapter four, we discuss how different reliability guarantees provided by the underlying event-based middleware affect our ability to perform complex event detection. The discussion examines several common errors that arise in distributed systems, and describes with the aid of an example how they affect complex event detection. The discussion then shows how applications can use detection policies to cope with these errors. Finally, it looks in detail at a policy

that prevents false positive detections, and evaluates the performance of an implementation of this policy.

In chapter five, we describe some of the issues involved in placing complex event detectors over a large-scale event-based middleware called Hermes. The discussion begins by giving several motivations for distributed detection of complex events. It then gives a brief overview of Hermes before describing several detector placement strategies. It concludes with an evaluation of these strategies in comparison to a reference strategy from the literature.

Finally, in chapter six, we summarize the contributions made, highlight remaining challenges and suggest future work.

Chapter 2

Background and Related Work

This chapter describes background material relevant to the remainder of the thesis. It begins with a brief overview of the origins of pervasive computing and the driving forces behind its emergence. This is followed by an introduction to wireless sensor networks and a discussion of their importance to pervasive computing.

We then give an introduction to *middleware*, and discuss its ability to support large-scale pervasive computing applications. Our discussion includes a survey of several types of middleware relevant to pervasive computing, including middleware for context-aware computing, stream processing, and wireless sensor networks.

Finally, we give an overview of research areas related to *complex events*. We compare complex events to several related concepts, and examine how they can be used to simplify the development of pervasive computing applications.

2.1 Pervasive Computing

The vision of pervasive computing promises applications capable of transparently adapting themselves to cope with changes in the user's environment. In contrast to traditional desktop computing, users need not be aware of any interaction with a pervasive computing system. Neither should they have to learn custom protocols in order to engage with new computing services. Instead, pervasive computing attempts to automatically sense information about the environment relevant to a user, sometimes referred to as *context*. It then delivers this context information directly to the user, or automatically performs some action to help the user complete a task.

During the mainframe era of computing, one device was shared amongst many. The mainframe era was supplanted by the PC era, with a device for every person. The pervasive computing era entails a continuation of this trend, with vast numbers of computing devices, many of which will be embedded in the environment. Originally proposed by Weiser under the moniker of Ubiquitous Computing [Wei93], it is closely related to several other research areas, including Sentient Computing [Hop00], Ambient Intelligence [DBS+01], and Context-Aware Computing [SAW94]. All these concepts share a common vision of a computing environment with many devices per user.

Several forces are driving the emergence of pervasive computing as an important field. At the hardware level, Moore's law has ensured that the cost of computation continues to fall, allowing for small, low-cost processors that are nonetheless quite powerful. The advent of Micro-Electro-Mechanical systems (MEMS) has made it possible to equip these devices with tiny sensors and actuators. Additionally, wireless networking technology reduces the need to deploy expensive communication infrastructure, further increasing device flexibility. At the software level, users are starting to exploit these technologies by creating more autonomous applications capable of detecting and responding to changes in the environment.

A taste of the diversity of pervasive computing applications can be gleaned from Table 2.1. Here we list several popular application sectors for pervasive computing. Smart Home applications tend to be relatively small scale. Some examples include automated control of lighting and heating, tracking and finding lost objects, and multimedia applications (e.g. music that follows a user between rooms). Health care environments have also benefited from the emergence of pervasive computing. In-hospital automated monitoring of patients helps to simplify the work of doctors and nurses. Remote monitoring of elderly or other at risk patients can promises to free up even further hospital resources. Pervasive computing has also been applied in emergency response scenarios to aid rescue workers. Finally, pervasive computing applications for the transport sector include the provision of context-aware route guidance to users, monitoring of transport systems for congestion, and vehicle-to-vehicle communication systems that help with accident avoidance.

Sector	Applications
Smart Home [EG01; MR03]	Smart Lighting and Heating, Lost Object Finder, Multimedia
Health Care [Bar04; KKH+08]	Remote Patient Monitoring, Active Hospital, Emergency Response
Transportation [GK02; FM09]	Real-Time Route Suggestion, Traffic Monitoring, Accident Avoidance

Table 2.1: Sample Pervasive Computing Environments

Some of the characteristics of pervasive computing systems are listed in Table 2.2.

A crucial aspect of pervasive computing is *context-awareness*. Changes in the user's context,

Characteristic	Description
Distributed	Dynamic conglomerations of networked devices provide services to users.
Large-Scale	Potentially vast numbers of users and machines.
Intelligent Environments	Intelligence given to non-computational elements of environment.
Global, Ad-Hoc Interaction	Users can access services anytime, anywhere.

Table 2.2: Characteristics of Pervasive Computing

such as the current location or activity, must be detected automatically by pervasive computing applications, and used to tailor the service provided to the user. Context awareness is usually aided by the deployment of sensor technology in the environment. Initial attempts to provide

such functionality relied primarily on location context (e.g. [WHFaG92]). Furthermore, the sensing technology employed was targeted at small indoor domains, required expensive supporting infrastructure, and did not concern itself with issues such as energy usage of devices. Since then, the sophistication of sensing technologies, and the variety of contexts they are capable of detecting, has increased considerably [BKZD04].

One particularly important class of sensing technology used to provide context awareness is that of Wireless Sensor Networks (WSNs). WSNs have several unique characteristics that impact the design of systems to support pervasive computing. We now give a more in depth introduction to WSNs, including a discussion of their main characteristics, design principles, and applications.

2.1.1 Wireless Sensor Networks

WSNs consist of potentially large quantities of sensor devices equipped with wireless radios and limited computational, storage, and energy resources. They also tend to be small and inexpensive. Examples include Crossbows' Mica Mote and Mica2Dot (an inch-wide version of the Mica Mote) [CBo09], as well as the prototype Smart Dust node[WLLP01], 5 millimeters in size, developed at Berkeley. However, many WSNs contain several more powerful devices, known as *base stations*, scattered throughout the network or deployed at the edge. Base stations can provide a gateway to other networks such as the internet, in addition to performing more heavyweight computations than motes.

Nodes in a WSN are usually unreliable due to power failures and other physical damage caused by environmental factors. Thus data is often read from multiple sensors in order to increase reliability, and also to improve accuracy by fusing correlated readings. As sensor nodes are power constrained, overcoming the scarcity of energy is a crucial design requirement at all levels. Nodes achieve this by sleeping when not in use, and by performing in-network processing to reduce the amount of communication, since the energy cost of communication is much greater than that of computation[1].

WSNs have a wide variety of application areas including disaster response, pollution detection, structural fault detection [KKP99], habitat monitoring [CEE+01; MPS+02], battlefield scenarios, smart energy [RAF+02], and health monitoring [HMCP04]. Xu [Xu02], and Romer and Mattern [RM04b], provide surveys of WSN applications. Some of the important characteristics of sensor networks are given in Table 2.3. Given these characteristics, several design rules have been proposed in the literature, some of which we list in Table 2.4. For a more in-depth introduction to WSNs, the interested reader is directed to [ASSC02; KW03; ESS02], which give a good overview of the area. Romer and Mattern give further analysis of the design space of Sensor Networks [RM04b].

[1] To the order of roughly 1000 execution cycles per bit transmitted

Characteristic	Description
Small devices	Cubic millimeter Smart Dust motes in the near future
Limited Power	Communication much more expensive than processing
Limited Resources	Resource contention between applications
Heterogeneous Nodes	Different sensing, processing, and storage capabilities
Frequent Failures	Empty batteries, Environmental influences
Dynamic Systems	Node mobility, Node failure, Environmental obstructions
Wireless Communication	Lossy links, Broadcast typically used
Dense Deployment	Typically many other nodes within range
Lack of access	Standalone operation, Difficult to reconfigure manually
High Data Volumes	In-network processing required
Environmental Interaction	Sensing and Actuating

Table 2.3: WSN Characteristics

Principle	Description
Data-centric communication	Provide information services to users as opposed to just connecting different parties e.g. query a geographic location as opposed to a particular node
Localised algorithms	Group sensor nodes for scalability & reliability Reduce long range communication Lack of RAM to store global state (motes)
Adaptive Fidelity algorithms	Tradeoff between QoS & resource usage May need to change routing algorithms & node configurations dynamically
Application knowledge in nodes	More efficient use of resources through increased data aggregation & fusion Tradeoff with generality of system
Modular software	Can't afford to have unused functionality Enables easier reprogramming of nodes
Lightweight middleware	Limited resources

Table 2.4: WSN Design Principles

2.2 Middleware

If the vision of pervasive computing is to become a reality, *middleware* technology will likely play a vital role.The term middleware was originally coined to describe software that simplifies interaction between applications and legacy systems. However, the most common usage of the term today is in reference to a layer of software, sitting between applications and the network layer on several networked hosts, which helps to address programming issues related to distribution and heterogeneity [Ber96]. In theory, interaction between distributed application components can be programmed directly by developers using primitives provided by the network layer. However in practice, application developers usually require a higher level programming model that shields them from much of the complexity of the underlying system.

For the purpose of this thesis, we adopt the definition of middleware suggested by Pietzuch [Pie04]:

Definition 2.2 (Middleware) *A* middleware *is a software layer present on every node of a distributed system that uses operating system functions to provide a homogeneous high-level interface to applications for many aspects of a distributed computing environment, such as communication, naming, concurrency, synchronisation, replication, persistence, and access control.*

As might be expected from this definition, any assessment of a particular middleware technology must consider the nature of the distributed computing environment in which it is to operate. Early middleware targeted applications distributed over local area networks (LAN). Their programming model was usually based on the concept of *synchronous remote procedure calls* (RPCs). In contrast to local procedure calls, RPCs send input parameters to a remote host. The remote host then executes the required computation and returns the result. Meanwhile, the requester blocks waiting for a response. The RPC programming model masks the distributed nature of the system, since local and remote procedure calls appear identical to a programmer.

Unfortunately, the RPC programming model is not suitable for large scale distributed systems such as those prevalent in pervasive computing. Its main drawback is its reliance on *synchronous* communication. Masking the distributed nature of an underlying system using synchronous RPCs is possible so long as communication latency is low. And indeed this assumption usually holds for LANs. However, upper bounds on response times are difficult to ensure for wide area networks (WANs). Hence in WANs, synchronous remote calls can take much longer than local ones. As a result, performance of the application degrades.

A more realistic approach to programming distributed applications over WANs is taken by *event-based* middleware. In contrast to middleware that provide RPCs, event-based middleware relies on *asynchronous* communication. An event-based programming model enables developers to structure their applications so that no blocking occurs waiting for a response. This allows other tasks to be performed in the intervening period, which can result in substantial performance improvements.

An event based programming model is also useful for applications that must react in a timely

fashion to changes in the state of a remote host. Under an RPC model, a local host must continuously poll the remote host to check for changes. In order to receive timely notifications, the local host must increase the polling rate. Unfortunately this is an inefficient use of bandwidth. A more timely and efficient solution is for the remote host to store a *callback* for the local host. When a change in state occurs, the remote host notifies asynchronously all hosts for which it stores a callback.

For some event-based middleware, clients must communicate with each other directly. However this solution does not scale well, since in many cases communication endpoints are relatively lightweight (e.g. mobile phones), and do not have enough resources to store large amounts of routing state. An alternative solution is to connect endpoints via an *event mediator*, as in Figure 2.1. Instead of clients storing routing information about each other, they offload it to the mediator, which acts as an intermediary and forwards events between them. Thus the scalability of such a middleware depends on the scalability of its mediator. For this reason, large scale systems usually implement the mediator in a distributed fashion.

Many event-based middleware, including some that allow clients to communicate indirectly via a mediator, offer only point-to-point communication. However, support for one-to-many and many-to-many communication is vital for pervasive computing applications. Event-based middleware that simulates such support using point-to-point communication tends to suffer from scalability issues. In particular, they forego efficiencies that middleware with dedicated one-to-many or many-to-many routing algorithms achieves by setting up shared routing paths between endpoints with overlapping interests.

Figure 2.1: Event Mediator

In the following sections we review several examples of the above types of middleware in more detail. We pay particular attention to their suitability for pervasive computing. We also examine several middleware that have been designed specifically for pervasive computing, or for related areas such as context-aware computing and WSNs.

2.2.1 Synchronous Request/Reply Middleware

Several early middleware employed a synchronous request/reply communication mechanism. Two of the most popular synchronous request/reply middleware are CORBA and Java RMI.

CORBA The Common Object Request Broker Architecture (CORBA) is a synchronous request/reply middleware designed for heterogeneous distributed systems by the Object Management Group (OMG) [OMG08]. Objects in CORBA are hosted by object request brokers (ORBS), and communication between a client and an object is relies on RPCs. CORBA defines a standard interface definition language (IDL) for objects and an inter-orb communication protocol (IIOP) for communication between brokers. The IDL allows objects to be written in any language so long as their interface can be mapped to it. This increases interoperability, since clients and remote objects need not be written in the same language.

In addition to the basic architecture, CORBA also provides a suite of middleware *services*. This includes services for naming, concurrency, transactions and events amongst others. Of particular interest to us are the Event Service and the Notification Service. The event service attempts to provide a publish/subscribe style of communication (see Section 2.2.3), but suffers from a number of drawbacks. These include a lack of support for filtering of events based on their content, a heavyweight implementation of events as CORBA objects, and an inefficient implementation of asynchronous communication on top of CORBA's synchronous method invocation. The Notification service remedies two of these problems by providing structured events and the ability to specify filters over their content. However, it still performs asynchronous communication of events using CORBAs synchronous RPCs.

Java RMI The Java RMI middleware is a synchronous request/reply middleware from Sun based on the concept of *remote method invocation* (RMI) [Sun06]. RMI is an object oriented version of remote procedure calls that allows an object in one Java Virtual Machine (JVM) to communicate with an object in a remote JVM. In order to achieve this a local object invokes one of the methods in the remote objects interface. Arguments to the method call are marshalled at the client side by a *stub* for the remote object. A stub acts as a proxy for the remote object at the client. Stubs are compiled by a programmer from the interface of a remote object, and allow for type checking of the remote method invocation. A stub forwards marshalled arguments of an invocation to the remote object's server, where they are received by a *skeleton*. The skeleton is responsible for unmarshalling invocations received from clients, passing them up to the remote object, marshalling responses, and sending them back to the clients. Skeletons are created by programmers at compile time and enable static type checking of object implementations.

Java RMI has several disadvantages from the perspective of pervasive computing. Firstly, interoperability is a problem, since all clients have to be implemented in Java. This is not feasible for large scale pervasive computing environments given their heterogeneous nature. Furthermore the synchronous nature of RMI introduces a tight coupling between client and server, reducing scalability.

2.2.2 Message Oriented Middleware

Message-oriented middleware are an alternative middleware paradigm to synchronous request/reply [BCSS99]. They provide support for asynchronous communication through the use of *message*

queues. Two of the more popular message-oriented middleware are the Java Messaging Service and IBM's MessageQueue.

JMS The Java Messaging Service (JMS) is an asynchronous messaging API for the Java platform [Sun02]. It allows clients to communicate with each other by sending asynchronous messages. Clients communicate indirectly with each other via a *JMS Server*. One-to-one communication is supported by Message Queues stored at the server. These receive a message from a sender, and promise to deliver it to a receiver. Message Queues are usually capable of storing messages persistently, allowing them to provide reliable delivery guarantees when required. Clients may also send messages to Topics, which provide a publish/subscribe interface and allow for many-to-many communication. Subscribers submit subscriptions to JMS for a particular topic. Publishers send events to the topic for their type at the server, which forwards them to all its subscribers.

JMS messages consist of headers, properties and a body. The body may include Java objects. Subscriptions can include filtering expressions over message properties, allowing for a form of content based filtering. The disadvantages of JMS are that the service itself is not distributed, since the specification only defines an interface to the broker network, and not its implementation. Furthermore, as with Java RMI, the requirement that clients of JMS must be written in Java limits its interoperability.

IBM WebSphere MQ IBM Websphere MQ is a message oriented middleware similar in many respects to JMS [IBM09]. The main abstraction is that of a message queues, which receives messages from senders and forwards them to receivers. Unlike JMS, it is language independent, and bindings of its API to several programming languages have been created. However, it does not attempt to perform type checking. Furthermore, it does not provide support for scalable many-to-many communication, as multi-hop routing and filtering of messages are not addressed. IBM have attempted to address this problem as part of the Gryphon messaging middleware discussed in Section 2.2.3.3.

2.2.3 Publish/Subscribe Systems

Publish/Subscribe systems provide an anonymous, many-to-many communication service that allows for the creation of highly scalable applications [EFGK03]. Publish/subscribe communication involves delivery of asynchronous messages called *events*. Publishers are producers of events and subscribers are consumers of events. Publishers and subscribers need not be aware of each other. Instead, subscribers specify the nature of the events they are interested in to the publish/subscribe system. It ensures that any event generated by publishers that match these specifications are delivered to subscribers. This form of communication is sometimes labeled data-centric, as opposed to node- or address-centric, since endpoints remain ignorant of each others' identity.

The scalability of publish/subscribe arises from the loosely-coupled communication it allows. Eugster et al. characterize this decoupling along three dimensions: time, space, and flow [EFGK03]. Time decoupling ensures publishers and subscribers do not have to be active at the same time for messages to be sent between them. When an event is received by the publish/subscribe system from a publisher, it delivers it to all subscribers, even if the publisher subsequently fails. Similarly, once a subscriber creates a subscription for an event, the publish/subscribe system can buffer any matching events while the subscriber recovers from a failure. Space decoupling refers to the fact that publishers and subscribers know neither the identity nor the total number of clients with whom they are communicating. This anonymity reduces the complexity of communication endpoints. Furthermore, it allows them to transparently join and leave the system. Flow decoupling refers to the asynchronous interface publish/subscribe systems provide to clients. In particular, subscribers do not have to block waiting for events to be delivered, or continuously poll the publish/subscribe system. Instead publish/subscribe systems notify subscribers when an event has occurred via an asynchronous callback. This allows subscribers to perform other tasks while waiting for events to be delivered, and reduces event delivery latency without requiring a high polling rate.

Publish/subscribe systems are not a panacea however. For some applications, synchronous communication may be more natural, or endpoints may need to know each others' identity. Nevertheless, publish/subscribe communication is a particularly good match for data-centric pervasive computing applications. In the following sections, we survey several representative publish/subscribe systems from the literature. We classify them into three different types: topic-based, content-based, and type-based publish/subscribe.

2.2.3.1 Topic-Based Publish/Subscribe

Early publish/subscribe systems provided a *topic-based* communication service. Topic-based publish/subscribe systems connect message producers to consumers via the indirection of a *topic*. Each topic has an associate *topic name*. Events also have a topic name, and producers send their events to the corresponding topic. Consumers create subscriptions for topics of interest to them. A topic-based publish/subscribe system ensures that consumers receive all events sent to topics matching their subscriptions.

Topic-based publish/subscribe bears many similarities to group communication [Pow96], from which it emerged. In particular, a topic name may be viewed as a group name. One of the first topic-based publish/subscribe systems is the Information Bus [OPSS93]. It requires producers to broadcast event messages to all clients, who then determine locally whether they are interested in the topic. However the Information Bus was designed for use in LANs, and its reliance on broadcast communication is not feasible at larger scales.

The main problem with topic-based publish/subscribe is that it does not allow for more complex filtering of events based on their content, which can be a major scalability problem as

event rates increase. However for smaller scale scenarios with lower event rates it provides a useful communication service.

2.2.3.2 Type-Based Publish/Subscribe

Type-based publish/subscribe systems enforce a close integration of publish/subscribe communication with object-oriented programming language concepts. In particular, they attempt to preserve encapsulation of event data, while still allowing for efficient distributed content-based filtering. This encapsulation is achieved for example by providing accessor methods for each attribute of an event. In contrast, other forms of publish/subscribe tend to expose the representation of events to clients, violating object-oriented programming principles.

One such system is that of Eugster et al. [EGD01], who extend the Java programming language with in-built support for type-based publish/subscribe communication. The extensions require use of a pre-compiler (similar to RMI), and consist of explicit constructs for publishing and subscribing to types. Events, implemented as objects and referred to as *obvents*, contain both data representing the content of an event and methods for accessing the data. Subscriptions may also contain content-based filters over the attributes of an event. Integration of publish/subscribe into the programming language allows for static type checking of events and subscriptions, and preserves encapsulation of event attributes. In addition, it allows subscriptions for an event type to implicitly match events whose type is a sub-type of that defined in the subscription. However, the fact that obvents contain code as well as data makes interoperability more difficult to achieve.

Eugster also suggests a library-based approach to the integration of publish/subscribe with Java [Eug07]. It uses parametric polymorphism [Sun05] and behavioural reflection to perform static type checking of events and subscriptions. This approach avoids explicit type casts, pre-generated typed proxies à la RMI, and pre-compilation of language extensions. It also allows for static compilation of filter code. This enables the matching process to access event attributes without using reflection. However this solution is still somewhat error-prone in Java, since runtime information about events is lost due to type-erasure. This requires programmers to redundantly specify type information in order to set up routing paths before events are generated. Eugster also discusses several other difficulties relating to the limitations of behavioural reflection in Java [Eug07].

2.2.3.3 Content-Based Publish/Subscribe

For many applications, topic-based subscriptions are too coarse-grained. In addition to the events they actually want, topic-based subscribers may receive many events that are irrelevant. Content-based publish/subscribe systems allow clients to define additional filters over event content as part of their subscriptions (or advertisements). This greatly enhances the expressiveness of publish/subscribe systems and, if implemented properly, increases scalability by reducing

unnecessary communication. However, storage and transmission of subscriptions and advertisements is more expensive for content-based systems. We now review several content-based systems from the literature. For additional information concerning routing in content-based publish/subscribe systems, we direct interested readers to the survey of Baldoni et al [BV06].

CEA The Cambridge Event Architecture (CEA) is an event-based middleware centered around a *publish-register-notify* paradigm [BBHM95; BMB+00]. During the publish stage, event publishers advertise a definition for the events they produce at a name trading service. Subscribers use this service to discover events of interest to them. Subscribers then register a subscription with publishers of these events. Publishers expose an interface containing an asynchronous registration method to receive subscription requests. The publisher interface also allows for regular synchronous communication. Finally, during the notification stage, publishers send their events to subscribers with matching registrations. The subscriber exposes an interface containing an asynchronous callback to receive these notifications. Event mediators with both publisher and subscriber interfaces are also supported by the CEA. These decouple communication and allow clients to offload the burden of storing routing information.

CEA type checks events statically in accordance with its goal of integrating publish/subscribe with existing middleware technology. CEA initially only allowed for equality filters over the content of events, but was later extended to support predicate filters over key/value pairs. However, it only type checks subscriptions dynamically at runtime. CEA also supports specification of *composite* event patterns in order to detect combinations of events [Hay96].

COBEA [MB98] is an CEA-based extension of CORBA with support for asynchronous events. COBEA clients use the CORBA IDL to specify typed events. These are transmitted as parameters in method calls in order to avoid using heavyweight CORBA objects. COBEA also allows for untyped events using the `any` datatype of IDL. The IDL compiler performs type checking of typed clients. Static type checking ensures that no run-time overhead is incurred from type checking, and allows for bugs to be caught early in the development cycle. However it does introduce a tight coupling between publishers and subscribers. Importantly, the subscription language of COBEA supports content based filtering of events.

The main disadvantage of CEA and its derivatives is the lack of support for distribution of content filters across multi-hop event mediators. This is critical for improving the scalability of a content-based publish/subscribe system.

Siena The *Scalable Internet Event Notification Architecture* (SIENA) is a large-scale content-based publish/subscribe system designed for scalability by Carzaniga et al [CRW01]. Publishers and subscribers communicate with each other in SIENA via an event mediator. The mediator is implemented in a distributed fashion over a multi-hop *overlay network*. Each node or *broker* in the overlay is responsible for hosting clients (i.e. publishers and subscribers), and for storing routing state that enables the efficient delivery of notifications. Routing state may include both advertisements from publishers indicating the events they intend to generate, and subscriptions from subscribers indicating the events they are interested in.

The choice of topology for the overlay network has a major effect on the scalability of event routing. Three different classes of topology are possible in SIENA (depicted in Figure 2.2), each with their own routing algorithm. The first, and simplest, is a hierarchical topology. Under this topology, a single broker is nominated as the root of a tree that connects all brokers. Brokers forward subscriptions and events detected by their children to their parent in the tree. They also forward events that a particular child detects to the rest of their children. This means that the only messages parents send to their children are events. Note however that advertisements are unnecessary with a hierarchical topology.

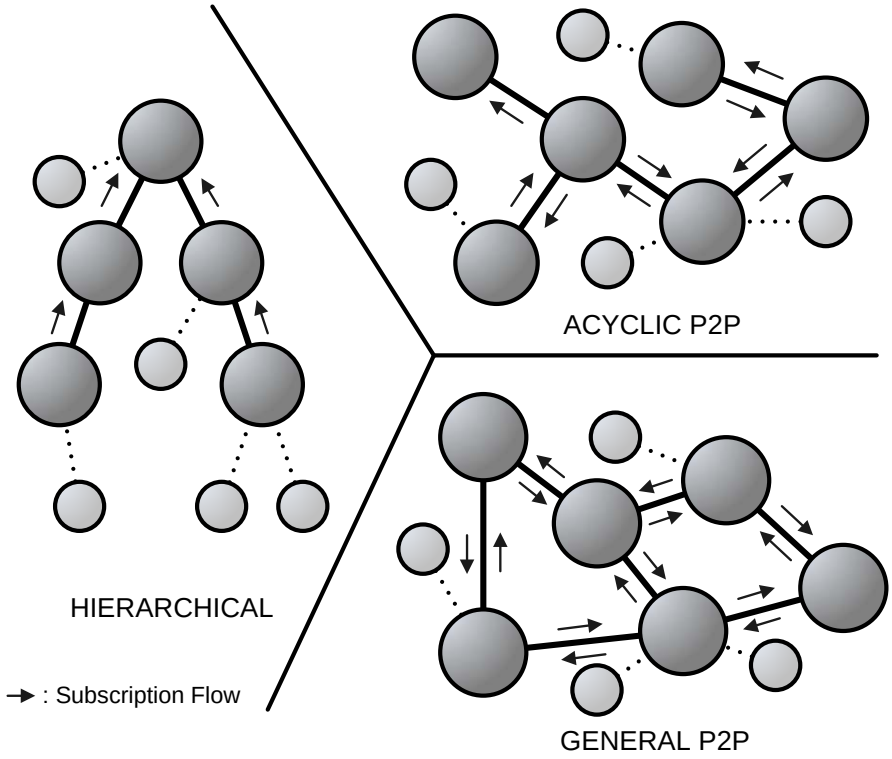

Figure 2.2: SIENA Routing Topologies

The second possibility for the overlay topology is an acyclic peer-to-peer topology. This topology connects all the brokers in the overlay via a minimum spanning tree. SIENA floods advertisements to all brokers over the spanning tree, creating an advertisement tree rooted at the publisher-hosting broker. Subscriptions follow the reverse path of matching advertisements towards the publisher-hosting broker. Finally, events generated by publishers follow the reverse path of subscriptions. This architecture assumes that events are more frequent than subscriptions, and that subscriptions are in turn more frequent than advertisements.

The final topology, general peer-to-peer, allows for redundant links in the overlay. Redundant links improve reliability with respect to broker failures. Furthermore they allow for more flexibility in the configuration of brokers. However routing over a general overlay is more com-

plex, since the routing algorithm has to handle cycles in the topology. SIENA uses a simplified distance vector routing protocol that extends each flooded message with the source's identifier and the distance traveled so far. At each hop, a broker examines its routing tables to see whether the message has arrived on the shortest path from the source. If not, it is dropped.

An important part of the SIENA routing algorithms is their ability to *cover* and *merge* advertisements and subscriptions so as to reduce network traffic and routing state storage requirements. One subscription covers another if their filters are equal, if the former has no filter, or if the former's filter is more general than that of the latter. When a broker is deciding whether to forward a subscription along a link, it first examines its routing tables to see whether it has already forwarded a covering subscription. If it has, it drops the new subscription. When two subscriptions overlap but neither of them covers the other, a broker may decide to merge them. Merging converts subscriptions with overlapping filters into a single joint subscription. Instead of forwarding both subscriptions to the next broker, a broker can forward a single merged subscription, reducing network traffic and routing state requirements downstream, at the cost of some extra computation. In a similar fashion, brokers may decide to cover and merge advertisements.

The scalability of SIENA arises from its combination of a distributed event mediator, content-based filtering of events, and covering and merging of filtering state. However, the overlay network topology must be configured statically by an administrator. This becomes unrealistic at large scales, where changes in the topology are common. Furthermore, in the worst case, SIENA floods advertisements to every broker in the network. With respect to the individual routing topologies, several other scalability problems arise. For the hierarchical topology, the root of the hierarchy is a single point of failure, and may prove to be a bottleneck. For the acyclic peer-to-peer topology, failure of a single link or broker partitions the network. For the general peer-to-peer, redundant links improve reliability, and allow for more flexibility in the configuration of connections between brokers. However, the topology must still be configured statically by an administrator at deployment time. In addition, as observed by Li et al [LMJ07], cycles in the overlay result in either redundant advertisements, subscriptions or publications. Finally, events in SIENA are unstructured lists of attribute value pairs. Without type checking, the likelihood of programming errors increases, reducing usability.

Padres Padres is a content based publish/subscribe middleware from the University of Toronto [LHJ05; LJ05]. Its original event dissemination algorithm is based on SIENA's routing algorithm for acyclic peer-to-peer topologies. Li et al. subsequently extended this algorithm to allow for event dissemination over general peer-to-peer topologies [LMJ07]. To prevent cycles, the algorithm assigns a unique *tree ID* to each advertisement predicate. This has the added benefit of reducing the number of times matching must be performed for an event. Furthermore it allows extra flexibility when forwarding events since a broker can sometimes choose a different path if one is overloaded. This also increases the flexibility of complex event detection, since alternate routing paths allow for a wider choice of detector placements. However, the algorithm still requires flooding of advertisements. In addition, the increase in routing flexibility comes at

the cost of a reduced ability to perform covering and merging of advertisements generated by different publishers, further increasing state requirements.

Gryphon Gryphon is a large-scale information flow architecture from IBM [BKS+99]. It supports messaging using an information-flow graph (IFG) model. Sources in the information flow graph produce events, and sinks receive them. Intermediate nodes transform events en route and may be stateless (e.g. filtering) or stateful (e.g. aggregation). The system maps nodes in the information flow graph to physical brokers in an overlay network. An administrator creates the overlay network statically at deployment time. For reliability, Gryphon groups brokers together to form *virtual broker cells*. It connects virtual broker cells to each other using *link bundles*, where link bundles may consist of several redundant physical links. The redundancy afforded by virtual brokers and link bundles allows Gryphon to cope with failures to a degree, although reconfigurations of the overlay network beyond this are complex. Gryphon provides several other important services, such as exactly-once delivery [BSB+02], durable subscriptions [BZA03], and relational subscriptions [JS03a].

Hermes Hermes is a large scale publish/subscribe middleware from Pietzuch that employs peer-to-peer technology to achieve scalability [Pie04]. It uses the Pastry distributed hash table (DHT) [RD01] to implement an event mediator as an overlay network of event brokers. Events in Hermes are typed, and are dynamically type checked by brokers at runtime. Before a client can publish (or subscribe to) an event, it must create and add a type schema for it to the middleware. Hermes stores this type schema at the *rendezvous* broker for the type. The rendezvous for an event type is found by hashing its name and searching for the broker whose Pastry ID is closest to the hash value. This spreads the burden of acting as a rendezvous node evenly. Hermes provides two event dissemination algorithms, type-based routing and type- and attribute-based routing. The former is analogous to topic based publish/subscribe, while the latter allows for content based filtering of events.

Routing of advertisements is the same for both algorithms. Having added the schema for an event to the system, publishers send advertisements for an event towards the same rendezvous broker. At each hop, intermediate brokers store routing state about the advertisements they have received and forwarded. Advertisements are only forwarded if no covering advertisement exists. Routing of advertisements stops at the rendezvous.

Type-based and type- and attribute-based routing differ in their handling of subscriptions and event notifications. Under type-based routing, subscribers forward their subscriptions towards the rendezvous. When an intermediate broker receives a subscription, it forwards it towards the rendezvous so long as it is not covered by an existing subscription. In addition, if it has received a matching advertisement from a broker other than the sender of the subscription, it forwards a copy of the subscription along the reverse path of the advertisement. When a subscription reaches the rendezvous, forwarding, including reverse-path forwarding, terminates. Event notifications follow the advertisement of their publisher towards the rendezvous. They also follow the reverse path of any subscriptions they encounter en route.

Type-based routing maintains relatively little routing state at brokers. This can be an advantage, but means brokers cannot filter events based on their content. In addition, every event publication must go to the rendezvous, even if no subscriber is interested in it. This could easily overload the rendezvous, particularly since no filtering of events is performed. Furthermore, if publishers are energy-constrained sensors, as is common for pervasive computing applications, it is a waste of their resources to publish events that no-one is interested in.

Type- and attribute-based routing takes an alternative approach to routing of subscriptions and event notifications. Subscriptions may contain filters over the content of events. Similar to type-based routing, they are sent by a subscriber towards the rendezvous broker. Each intermediate broker evaluates covering relationships between its existing routing state and every subscription it receives. It forwards uncovered subscriptions to the rendezvous, and along the reverse path of any matching advertisements. It may also choose to merge overlapping subscriptions.

In contrast to type-based subscriptions, the rendezvous may continue to forward some of the type- and attribute-based subscriptions it receives. Instead of always dropping a subscription, the rendezvous looks to see whether there are any matching advertisements for it, and if so, forwards it along their reverse path. In this way, subscriptions reach all brokers hosting publishers with matching advertisements, so long as they are not covered by existing subscriptions. In contrast to type-based routing, event notifications only follow the reverse path of subscriptions, and do not always reach the rendezvous. Furthermore, the publisher hosting broker does not need to forward events unnecessarily, since it always knows the current matching subscriptions for its publishers.

Hermes, and in particular type- and attribute-based Hermes, is highly scalable. This is due in large part to its use of the Pastry DHT to maintain an overlay network. In contrast to SIENA, addition and removal of brokers to the Pastry DHT is simple, since it automatically reorganises in response to changes in the underlying topology. Furthermore, it allows for fast and localized recovery from link and broker failures. For example when a link between two brokers fails, the source broker automatically reroutes any subscription or routing state traveling in the direction of a rendezvous broker along that link towards the new nearest neighbour to the rendezvous. Apart from its scalable routing algorithm, Hermes is a fully fledged event based middleware. It uses open standards for messaging and expressing filters (XML and XQuery [W3C07]) to enable interoperability. It also provides several higher level services, such as composite event detection and security. On the downside, it is difficult to provide strong reliable delivery guarantees in Hermes. We discuss Hermes further in Chapter 5 in relation to the distributed placement of complex event detectors.

2.2.4 Data Stream Management Systems

A related area to publish/subscribe systems is that of data stream management systems (DSMSs). DSMSs arose from work done in the continuous query community to improve the scalability of

centralized continuous query systems. Similar to publish/subscribe systems, DSMSs provide a data-centric programming interface. Their modus operandi is to compile continuous queries into a graph and then map this query graph efficiently to an overlay network. Operators in the query graph may be stateless, such as simple filters, or stateful, such as aggregation or sliding window join operators. This model is similar to the information flow graph model we discussed earlier for Gryphon (Section 2.2.3.3).

Some of the work done by the DSMS community in terms of distribution includes finding efficient placements for operators, dynamic load balancing, and coping with lost or delayed messages. Operators in a graph may have multiple inputs and outputs. However, in contrast to most large-scale content based publish/subscribe systems, the routing algorithms of most DSMSs are not designed to cope with large numbers of overlapping publishers and subscribers. We now give an overview of some of the more prominent distributed DSMSs from the literature.

Aurora* and Medusa Aurora* and Medusa [CBB+03] are distributed implementations of Aurora [CcC+02; ACc+03] (see Section 2.3.2). They attempt to provide support for large scale streaming applications requiring communication and cooperation across multiple administrative domains. Aurora* focuses on intra-domain communication, while Medusa handles inter-domain communication and cooperation. Each domain consists of one or more event brokers. Within a domain, Aurora* stores information about the locations of all producers, consumers, and query graph operators in a catalog. The implementation of the catalog may be centralized or distributed, but it must be accessible to all brokers in the domain. New queries can reuse existing operators by looking up their location in the catalog.

In many cases, query graphs may involve entities from multiple administrative domains. Medusa enables the creation and deployment of streams across domain boundaries. It also allows for the negotiation of contracts that specify payment agreements between domains that wish to trade resources. Information about the location of producers, consumers, and query graph fragments is available in Medusa through a distributed catalog. In contrast to the intra-domain catalog, whose implementation is left unspecified, the inter-domain catalog is implemented using a distributed hash table. Each entity has a globally unique identifier, and the hash of this identifier is used as a lookup index for information about the entity in the DHT.

Hourglass The Hourglass project at Harvard [SPL+04] attempts to deliver data from sensor networks to interested parties using a stream abstraction. Hourglass manages multiple streams and tries to find commonalities between them in order to reduce bandwidth consumption while providing reasonable latency. A key component of Hourglass is the Relaxation stream operator placement system of Pietzuch et al [PLS+06]. It attempts to place query graph operators efficiently by modeling the physical overlay network as a *latency space*. This space assigns each broker an approximate network coordinate, allowing it to estimate its distance to other brokers in the network. A spring relaxation algorithm enables operators in the query graph to dynamically

modify their position in a decentralized fashion in response to changing network conditions. Relaxation also suggests a multiplex operator for handling large numbers of subscribers to an operator, but details of how this might work are not given.

2.2.5 Middleware for Context-Aware Applications

An important characteristic of pervasive computing applications is *context-awareness*. Context-awareness allows applications to tailor the service they provide to users in response to changes in the environment. Early pervasive computing systems required applications to incorporate context information in an ad-hoc manner. Subsequently, several researchers recognised the need for context-aware middleware to simplify application development. Key issues for context-aware middleware include scalability, adaptivity to environmental changes, and the discovery, representation, and aggregation of contextual information. We now give an overview of several such middleware from the literature.

The Context Toolkit Dey et al. [ADB+99] discuss several requirements for dealing with context, and emphasize the need for context that supports and enhances a users ability to perform *tasks*. They present a software infrastructure called the Context Toolkit to meet these requirements [DAS99]. The Context Toolkit architecture contains three main types of component: context widgets, context servers, and context interpreters. Context widgets are used to abstract over different context sources. This helps to increase reuse by decoupling context information from the device that produces it. Context servers are responsible for aggregating all context relating to a single object in one place. Finally, context interpreters convert low-level context into high-level context. These components are executed independently from applications, but are fixed at design time. The Context Toolkit does not support publish/subscribe communication, and distribution of the various components is not discussed by the authors. The Context Toolkit lacks scalability since widgets and servers must store information about all their subscribers.

Solar Chen et al. [CLK04; CK05] designed the Solar middleware for the collection, aggregation and dissemination of context information. The middleware is built around the concept of a Context Fusion Network (CFN). Applications specify operators (both system and user-defined) over streams of context information. Solar merges this operator tree into a system-wide CFN in order to reuse common operators and thus improve scalability. Users define operators explicitly, in contrast to operators defined implicitly via a complex event language, and operators can therefore presumably contain arbitrary code.

Similar to the Hermes middleware described in Section 2.2.3.3, Solar relies on an overlay network implemented using the Pastry DHT. However in contrast to Hermes, subscribers send subscriptions to publishers independently, and not based on their advertised type. Brokers cooperate to forward events using SCRIBE [RKCD01], an application-level multicast algorithm.

This becomes less efficient than Hermes type- and attribute-based routing when there are many publishers with the same type. It is also less data-centric, since subscribers need to know the identity of individual publishers. Furthermore it does not take into account the needs of energy constrained wireless sensor networks since all events generated by a publisher must go to the root of the multicast tree, even if there are no active subscribers.

The same researchers suggest a policy-controlled data dissemination service called PACK [CK05], which allows Solar to cope with conditions of high load. Overloaded event queues can drop events and summarise them into digests. Users define policies for each of their subscriptions. These policies control both the order in which events are dropped by queues and the aggregation functions used to generate digests. In the worst case, overloaded brokers who cannot meet policy requirements simply drop all events. However, they also create a summary of dropped events using the aggregation functions defined in policies, or the COUNT aggregation function if none are specified.

SCI The Strathclyde Context Infrastructure (SCI) of Glassey et al. [GSR+03] is a middleware framework for managing context. The central abstraction is that of a range, which hosts the entities that produce, manage, and consume contextual information in an area (physical or logical). Ranges are organized into an overlay network for scalability and robustness. Each range contains a context server (CS) that stores information about entities in that range and enables communication with other ranges. The CS can also use a set of specialist services called Context Utilities to help manage its range (e.g. Location Service, Event Mediator Query Resolver). SCI uses the idea of a dynamic composition graph similar to the Context Fusion Network of Chen et al. [CLK04] to deliver information from sources to sinks. SCI does not attempt to optimize operator placement however, and its context query language does not support complex event expressions.

Nexus Lehmann et al. [LBBN04] address the problem of modeling context information, concentrating on the interoperability between different world models, and between applications and resources. They outline a system for managing small to medium domains, and a global architecture called Nexus for federating between several such systems. Nexus relies on a global *Augmented World Model* ontology for this federation, a concept similar to the Pervasive Computing Standard Ontology of Chen et al [CFJ03].

In addition, Nexus provides two complementary services that address the issue of event management for mobile users [Bau04]. A notification service is responsible for delivery of events to users. An observation service supports the detection of several common state-based predicates, similar to complex event expressions. These can be parametrised by users as desired. Users may also stipulate a confidence level to control the number of false positives and negatives. The observation service leverages the notification service for delivery of the events it detects. Nexus uses the Pastry DHT to store information about the locations of event sources and sinks. However Nexus routes events directly between communication endpoints. This raises scalability

issues for large numbers of subscribers. Furthermore, the observation service does not attempt to optimize placement of detectors.

CIS The Contextual Information Service (CIS) of Jin et al. provides a distributed database interface for accessing context information [JS03b]. It allows users to specify quality of service requirements including accuracy and timeliness as part of a query. CIS uses this information to quench the information produced by sensors when it is not needed by subscribers. This helps to reduce unnecessary communication, and may also prolong the lifetime of sensors if they are energy constrained. The CIS interface is conceptually similar to the database abstraction proposed for WSNs (see Section 2.2.6.2), although the CIS implementation is targeted towards more resourceful environments. CIS is not particularly concerned with scalability, and has only limited support for event routing.

Abstract Events Katsiri et al. argue in [KBM04] that current event models are limited for use in a variant of pervasive computing called Sentient Computing [Hop00]. These limitations arise from the incomplete mapping from abstract top-level knowledge to low-level concrete data produced by sensors. They propose the concept of *abstract events* to model notifications about transparent changes in distributed state. They express abstract events using temporal first-order logic (TFOL). An abstract event detection service takes in TFOL abstract event definitions and creates detectors to notify the user on their occurrence. Thus, abstract events allow users to specify high-level events that correspond closely to real world concepts. However their detection algorithm is based on the Rete algorithm (see Section 2.3.3), which scales poorly in terms of memory usage, and is unsuitable for deployment on resource-constrained devices. Furthermore, the Turing-complete nature of TFOL makes it hard to reason about. A closed-world assumption, as required to support negation and quantification, makes abstract event detection difficult in open distributed environments.

Sentient Objects Biegel and Cahill [BC04] propose the Sentient Object programming abstraction to support event-based situation analysis in sentient environments. Each sentient object interacts with sensors and actuators using an event-based communication infrastructure to handle mobility. A Sentient Object provides components for fusing sensor data using Bayesian networks, storage of context for efficient reasoning, and an inference engine for determining what course of action to take on the occurrence of an event within a context. Application developers can program Sentient Objects at a high level using graphical tools. A context acts as a dynamic filter over input events streams, in that only events of interest in a particular context must be delivered to the object. However Sentient Objects themselves are implemented in a centralised fashion.

2.2.6 WSN Middleware

WSNs are an important component of many pervasive computing applications. The literature contains a variety of middleware and programming models targeted towards WSNs. These range from low-level platform-centric programming models that target individual WSN nodes, to network-level macro-programming abstractions that treat a WSN as a single entity.

In this thesis, our interest in WSNs relates primarily to how they can be integrated into larger-scale, possibly global, pervasive computing infrastructures. Applications that utilize such an architecture may require the services of several distinct WSNs. These WSNs may be part of separate administrative domains, and are likely to have different characteristics in terms of sensor technology, routing topologies, and scale.

One of the key design principles of WSNs is that sensor data should be processed in-network to reduce unnecessary communication. Therefore pervasive computing applications that utilise WSNs may wish to perform some of their processing within the WSN domain. Ideally, developers could define the processing necessary for such applications in a data-centric, domain-independent programming model. However, for efficiency reasons, applications may require more fine-grained control over WSN data processing than this would allow. As an example, collaborative object tracking in WSNs usually requires low level control over communication between WSN nodes.

This control could be provided by a more sophisticated pervasive computing programming model that incorporates support for WSN data processing. Alternatively, developers could implement their applications using two separate programming models; the first programming model would support in-network data processing for the WSN, and the second would support further processing within the rest of the pervasive computing system as before. In between these alternatives lies a spectrum of application development models, where varying amounts of WSN data processing are defined using the pervasive computing system's programming model and then pushed into WSNs.

We now give an overview of several classes of popular WSN middleware and programming models. For a more detailed discussion of middleware issues in sensor networks, several surveys can be found in the literature (e.g. that of Yu et al. [YKP04] or that of Roemer et al [RKM02]). A good survey of common programming models for sensor networks is that of Sugihara and Gupta [SG08]. They divide sensor network programming models into node-level, group-level, and network-level programming models. Readers are referred to their survey for more details on node-level programming models. We concern ourselves mainly with the latter two, since node-level programming is too low-level for developers of large-scale multi-domain pervasive computing applications.

2.2.6.1 Group Level

Group level programming models take a more application centric view than that of platform-centric node level programming models. As their name suggests, they provide primitives to simplify creation and manipulation of groups of nodes in order to support collaborative in-network processing. Sugihara and Gupta [SG08] divide groups into *neighbourhood* groups and *logical*

groups, and classify group-level programming models according to this distinction. Neighbourhood groups are defined based on proximity to a node or location. Logical groups are defined based on some attribute of a sensor node or its data. Members of logical groups do not have to be physically located near to each other. Groups can be further distinguished by whether their membership criteria are data-centric or involve node or network-centric attributes such as topological information. One advantage of neighbourhood groups, and in particular one-hop groups, is that WSNs that communicate using wireless broadcast can support them very efficiently.

Abstract Regions The Abstract Regions programming model of Welsh and Mainland [WM04] provides support for constructing groups of sensor nodes, and for performing operations on those groups. Groups come in a variety of forms, including k-hop neighbour groups, k-nearest neighbour groups, groups within a spatial distance, and global spanning trees. Furthermore, Abstract Regions provides several group operators, including operators for enumerating the members of a group, sharing data between members, and aggregating member data values. Groups also have a QoS interface that enables applications to trade-off quality of service with resource usage. Abstract Regions provides a useful building block for the implementation of higher level constructs and languages. However, it mainly focuses on providing primitives to simplify the collection of sensor data from groups of nodes, and not on providing support for interpretation of this sensor data. Thus it is still relatively low level in comparison to the more data-centric programming abstractions we will discuss in Section 2.2.6.2, as it requires programmers to think in detail about the nature of the underlying network topology.

Hood Simlar to Abstract Regions, Hood is a group-level programming model from Whitehouse et al. [WSBC04] based on the notion of a *neighbourhood*. Nodes in a neighbourhood share data with each other according to a push policy. In contrast to Abstract Regions, Hood only supports one-hop neighbourhoods. Extensions that support multi-hop neighbourhoods are suggested by the authors as future work, but interestingly they also mention that none of the applications they surveyed required this. However this may be due to the difficulty of implementing such groups without programming support.

EnviroTrack EnviroTrack [ABC+04] is a logical-group programming abstraction from Abdelzaher et al. targeted towards object-tracking applications. It forms groups based on physical events that occur in the environment, and allows these groups to be named using *context labels*. Groups can then migrate in an agent-like fashion as further events occur. Thus in contrast to Abstract Regions and Hood, it provides management support for *mobile* groups. However, in order to prevent redundant groups, they enforce a restriction on the relationship between a nodes sensing radius and its communication radius.

PIECES The PIECES [LCL+03] group-level programming model for sensor networks of Liu et al. provides a state-centric programming abstraction for collaborative and signal information processing (CSIP) applications. This abstraction supports the definition of groups in terms of both their structure and the roles which different members of the group should assume. They show how their model can be used to implement a multi-target tracking application. In comparison to EnviroTrack, their implementation is capable of handling cases where different objects cross tracks.

2.2.6.2 Network Level

In contrast to node or group level programming models, network level programming models allow programmers to define programs for the network as a whole. They attempt to abstract over lower level communication concerns, and allow the programmer to concentrate on high level tasks like data processing. They can be broadly classified into two types (as done by Sugihara and Gupta or Gummadi et al. [SG08; GGG05]), those that take a data-centric view and those that take a more network-centric view.

Distributed Database abstraction Some of the most popular research implementations designed for programming and collecting data from sensor networks take a distributed database view. Hellerstein et al. [HHM03] present an overview of the motivation for this abstraction and some of the research challenges concerning WSNs being faced by the database community. The main problem with taking a distributed database view is that important data may not always be resident in a node, and custom collaboration may have to be performed in order to achieve some tasks (such as object tracking).

Woo et al. [WMG04] give an interesting overview of system architecture directions for query processing in sensor networks. They emphasize that the strictly defined layers of traditional network stacks may have to be sacrificed in wireless sensor networks in order to save energy. This portends architectures where lower layers suggest possible configurations to higher layers, which in turn supply routing hints to lower layers in order to optimise the network for a particular application's needs. This may result in a situation where the tightest (i.e. least complex) API of all the layers in the stack is the interface presented to the user by the application layer.

TAG Madden et al. developed a generic aggregation service for ad hoc networks of TinyOS motes called TAG [MFHH02]. It treats the network as a distributed database, and provides a simple declarative interface based on a subset of standard database query languages for data collection and aggregation. Aggregation queries are distributed and executed in the network by TAG in order to reduce the total amount of communication, and hence save power. The authors present a taxonomy of aggregates that distinguishes aggregation functions along different dimensions, enabling them to make generic routing decisions based on a function's classification. Some common sensor tasks such as object tracking are inherently localized and node-centric

however, and difficult to express declaratively.

One drawback of TAG is that data acquisition is performed at a user specified sample rate. Hence there is no way for data to be acquired asynchronously in response to external events. Madden et al. subsequently [MFHH03] proposed the idea of an acquisitional query processor (ACQP) for sensor networks. ACQP provides much of the functionality of TAG, but in addition provides support for other forms of data acquisition such as asynchronous events. It also addresses issues of query optimization, dissemination, and data quality. As with TAG, power optimization is central to the design of ACQP. However, only primitive, locally generated events are supported.

COUGAR COUGAR [YG02; BGS01] is a data collection service for WSNS from Gehrke et al. that provides similar functionality to TAG. Queries are disseminated by COUGAR using a query optimiser at the WSN gateway. However unlike TAG, it assumes the gateway has complete knowledge of the network topology. Furthermore, adapting a distributed query to take into account changing network conditions is not possible.

Bonfils and Bonnet [BB03] attempt to address this latter problem. They model their solution on the task assignment problem, i.e. they try to find a mapping of query operators to sensor nodes that minimizes the amount of data transferred over the network. Under their solution, operators are initially mapped arbitrarily, but gradually improve their positions in a decentralized fashion using a *spring relaxation* algorithm. To find better placements, each operator maintains a set of *tentative* operators on neighbouring nodes. Tentative operators continuously compare the efficiency of their placement to that of the active operator. Active operators migrate to a node hosting a tentative operator if it will result in a more efficient placement. This algorithm is similar to the Relaxation placement algorithm for wired networks proposed by Pietzuch et al. as part of the Hourglass project (see Section 2.2.4) [PLS+06]. However, instead of tentative operators, the Relaxation algorithm uses a latency space to find better placements.

DSWare The Data Service Middleware (DSWare) of Li et al. [LSS03] extends the sensor database abstraction with a real time event detection service. DSWare also provides Data Storage, Data Subscription, Group Management, Data Caching and Scheduling. Li et al. distinguish between primitive and compound events, and allow a confidence value to be associated with events for improving the detection reliability. An SQL-like language is used for registering and canceling events.

MiLan MiLan [HMCP04] is a middleware for sensor networks from Heinzelman et al. that efficiently manages sensor network resources in order to meet application QoS requirements. Applications submit QoS requirements as state-based variable graphs, and Milan proactively adapts the network configuration to meet these requirements. A network abstraction layer allows tight coupling with the underlying network using plug-ins. Network management and adaptation is left to the plug-in. It supports concurrent applications by allowing policies to be

specified to deal with resource contention, but it does not allow for dynamic application updates.

Macro-Programming Similar to the distributed database abstraction, *macroprogramming* systems allow programmers to treat the network as a whole. However, they typically provide network-centric programming abstractions instead of the data-centric interface of distributed database systems.

Regiment Regiment [NMW07] is a macroprogramming language from Newton et al. that builds on abstractions from functional programming. It allows programmers to create and manipulate streams and regions of sensor data. Regions may be defined based on a variety of grouping mechanisms, including location based regions or k-hop neighbours. Programs in Regiment compile to an intermediate Token Machine language (TML), since the gap between Regiment and low-level sensor node programming languages is too high. Programs in Regiment are succinct, but it is not a panacea, and the authors observe it is not best suited to applications that require frequent dynamic reprogramming (in comparison to distributed database style query interfaces).

Kairos Kairos [GGG05] is a language independent macroprogramming model from Gummadi et al. that allows users to specify global application behaviour in a centralized fashion. Kairos provides primitives for addressing and manipulating nodes and lists of nodes, for retrieving the one-hop neighbours of a node, and for accessing local and remote variables. Kairos is implemented as an extension to existing programming languages. Programs are first preprocessed into annotated source code, and then compiled into a node specific binary. Kairos provides a synchronous programming interface. However, the Kairos runtime uses asynchronous request/reply to access remote variables, and returns cached values of a previously accessed variable immediately. Overall, Kairos has a more restricted set of data types and operations than Regiment, but its language-independent nature may be advantageous.

2.3 Complex Events

One of the key observations made by researchers regarding pervasive computing is that applications are not usually concerned with individual sensors. Rather, they are interested in the information these sensors produce. Therefore, system support for context acquisition in pervasive computing applications tends to rely on *data-centric* routing. Instead of explicitly naming relevant data sources by their network address, application developers prefer to specify the information they are interested in declaratively. This leaves it up to the system to find relevant data sources and deliver their data to the application. Since applications are not aware of which information sources they communicate with, and vice versa, data-centric routing allows for *loosely-coupled* communication between information sources and users. As we described in previous sections, loose-coupling makes it easier to create highly scalable systems. This is crucial

given the scalability requirements of pervasive computing.

Hence it is important that application developers have a mechanism for quickly and accurately specifying sensor data of interest. For data modeled as primitive events, this will typically take the form of a filter language. However, in many cases, the raw data produced by sensors requires additional processing to be of practical use. Application developers thus require a means to specify complex *patterns* of event occurrences.

Researchers have proposed a wide variety of techniques to help application developers specify high-level complex events. In this thesis we investigate the use of a dedicated *complex event language* designed for use in pervasive computing applications. Complex or composite event languages arose as part of work done by the Active Database community. They allow users to detect patterns of events through the specification of complex event operators. For the purposes of communication, complex events can be treated in the much the same way as primitive events. In terms of modeling of real world occurrences, they extend the expressiveness of the event abstraction, and make it easier for users to detect high level patterns of interest.

Most complex event languages can be viewed as examples of *domain-specific-languages* (DSLs). While typically not providing the functionality of a general purpose programming language, the availability of a DSL for a particular domain has several advantages. Programs written in DSLs are usually clearer since they are closer to the semantics of the domain. Furthermore, they make it easier to write programs targeted towards their domain, and also make it simpler to find mistakes in programs. In some cases domain specific languages can be understandable to non-programmers familiar with the domain. Together, these advantages can help to greatly improve developer productivity.

The benefits of DSLs must be weighed against their costs. Creating a new programming language is a difficult task. In addition, developers must take the time to learn a new language. To ease the task of language design, and to help smooth the learning curve for developers, building on concepts from existing languages is usually a good strategy. However, complex event languages for active databases were primarily designed for use in centralized environments. Extending them to work for open distributed environments such as pervasive computing architectures raises several challenging problems. In the following sections we review several complex event languages from the literature, in addition to several other relevant approaches to pattern specification and detection.

2.3.1 Complex Event Languages

Having discussed the role of middleware in pervasive computing systems, we now turn our attention to *complex events*. Complex events can be used to represent combinations or patterns of raw sensor data. Specifying and detecting such patterns is of particular importance to programmers of pervasive computing applications, since individual sensor readings are often hard to interpret reliably.

Programmers usually express complex event patterns within a *complex event language*. Complex events languages were first suggested by the Active Database Community. Their original

motivation was to enable database programmers to specify rules that trigger an action to take in response to an event occurrence. Events of interest were primarily database related, such as the update of a particular tuple or the commit of a transaction. The rules defined by programmers to control active behaviour are known as Event-Condition-Action (ECA) rules. Complex event languages allow the detection of complex events during the event part of the rule. When the system detects an event, it checks to see whether the rule has an associated condition[1]. If so it evaluates it with respect to the state of the database. Finally, if the condition holds, the rule fires, triggering its associated action. An action usually generates a new event, although for many languages it may contain arbitrary code.

Complex event languages provide a wide range of complex event *operators* for detecting event patterns. Popular operators detect sequences (i.e. ordered occurrences of events), conjunctions (i.e. unordered occurrences of events), and negation (i.e. the non-occurrence of an event). Composition of operators enables the output of one operator to be used as the input to another. Languages usually differ in the precise operators provided. Complex event languages must also cope with the fact that there may be several different ways to detect a composite event occurrence for a given set of input events. A language's semantics must be clear on which events are used, and what is done with them after detection. Several different formal models for complex event detection have been suggested, including finite-state automata, petri nets, and detection trees.

Although originally designed for centralized scenarios, several researchers subsequently proposed complex event languages for use in distributed event-driven applications, such as network monitoring [MSS97] or distributed debugging [Sch96]. However, accurate and efficient complex event detection is more difficult to achieve in distributed systems. Unlike centralized active databases, where events are usually timestamped with a local logical clock, the lack of a global clock for distributed systems can make it impossible to determine the true ordering of a set of events. Furthermore, messages may be lost, delayed, or arrive out of order. These characteristics limit the timeliness and precision achievable by distributed complex event detection.

Distributed complex event detection for pervasive computing applications introduces even further challenges. In particular, many of the event sources for pervasive computing applications are energy-constrained wireless sensors. A key characteristic of these sensors is that it takes much more energy for them to communicate than to compute. Hence a popular design principle is to push any computation that may reduce unnecessary computation onto sensor devices. However, the heterogeneity of sensors makes this a difficult task, since detection of all or part of an expression can only be performed on sensor devices with sufficient computational and memory resources to handle it. Thus non-trivial system support is required for decomposition and placement of complex event detectors. Furthermore, sensors may be deployed redundantly for reliability reasons. Hence language constructs to filter and aggregate duplicate readings from multiple sensors are necessary. Finally, applications may require detection of a complex event expression at several different geographic locations. The language should allow for partitioning

[1] Rules without a condition are known as Event-Action (EA) rules.

of complex event detection so that it may be performed separately for each region.

We now give a brief introduction to several representative complex event languages. A more comprehensive overview is postponed to the discussion of related work in Chapter 3, where we give a detailed comparison of a variety of languages from the literature to a new language we propose for pervasive computing applications.

Ode Ode [GJS92; GJ92] is a regular-expression like language from Gehani et al. where composite events are detected using finite state automata. It supports a variety of composition operators. Some of these are listed in Table 2.5. Ode supports parametrised detection, where expressions may contain conditions that reference attributes of different input events. Conditions can span operands of a single operator, as well as operands of different operators. This helps to improve detection efficiency, since filtering can be performed earlier in the detection process. The alternative is to ignore the condition initially, generate all possible matching instances for the operator, and then post-filter instances whose attributes don't satisfy the condition. However this is much less efficient, and may also affect the correctness of event detection. An example of an Ode expression involving an implicit condition across event attributes (from [GJ92]) is:

```
immediate_rehire(X) = sequence(fire(X), hire(X,Y,Z))
```

Here `fire(X)` and `hire(X,Y,Z)` are primitive input events with schemas `fire(name)` and `hire(name, age, sex)`. The expression checks for a sequence of `fire` and `hire` events. Note that the variable name `X` is used by both operands of the sequence operator. It acts as an implicit constraint, and ensures the expression only matches sequences of `fire` and `hire` events for the same person.

An advantage of Ode is its formal definition based on finite state automata, since they are both familiar and efficient. However Ode does not allow for extensive control over how input events are reused, and the time model, designed for centralized active databases, is unsuitable for distributed systems.

Snoop Snoop [CM94; CKAK94] is an expressive complex event detection language from Chakravarthy et al. with extensive temporal support. Snoop relies on a tree-based detection model, where a tree is derived from the structure of event expressions. Nodes of the tree correspond to Snoop's composite event operators. Snoop provides a variety of operators, a selection of which we list in Table 2.5. An example of a Snoop expression that detects a sequence of `fire` and `hire` events is:

```
immediate_rehire = fire;hire
```

Here the sequence operator is represented by a semicolon (`;`), and `fire` and `hire` are primitive events whose schemas we defined as part of our description of Ode. Note that this expression does not ensure `fire` and `hire` events refer to the same person.

Snoop introduces the notion of parameter contexts, (also known as consumption policies), to control the constituent events of a complex event in the case of ambiguity. One popular consumption policy in Snoop is the *recent* consumption policy. Under this policy, if several `fire` events occur before a `hire` event, only the most recent `fire` event is retained, and thus only a single instance of `immediate_rehire` occurs. Consumption policies are an important innovation, since different applications may require different detection semantics when ambiguous detection is possible. Earlier languages implicitly defined a single semantics, limiting their scope. On the other hand, allowing the configuration of detection using consumption policies increases the complexity of the language.

Some disadvantages of Snoop are that it does not allow the specification of conditions taking into account the attributes of events, and was not designed with or distributed systems in mind (although attempts to define a distributed detection semantics were made subsequently [YC99]). In addition, consumption policies in Snoop can only be specified globally, instead of for individual operands.

Amit The Amit language of Adi and Etzion allows the specification and detection of complex events, which they refer to as *situations* [AE04; AE02]. A situation is defined in XML by a single operator. Input events relevant to an operand of the operator, called candidates, are stored by Amit in a candidate list for the operand. In addition to an operator, each situation has an associated *lifespan*, which acts as a context for its detection.

Table 2.5 gives a selection of Amit operators. An example of a lifespan definition (taken from [AE04]) is:

```
lifespan = "example 7"
initiator = event: "option-quote"
                   where: "symbol = IBM and stock exchange = NYSE"
                   correlate: "add"
initiator = event: "stock-quote"
                   where: "symbol = IBM and stock exchange = NYSE"
                   correlate: "ignore"
terminator = event: "detected situation"
                   where: "symbol = IBM and stock exchange = NYSE"
                   termination type: "discard"
                   quantifier: "each"
terminator = expiration interval: "60 minutes"
                   termination type: "discard"
```

A situation definition can reference this lifespan if it wishes to limit the input events its operator receives. Instances of the lifespan are *initiated* by the occurrence of IBM `option-quote` and `stock-quote` events from the New York Stock Exchange (NYSE). Lifespans are *terminated* by an event of type `detected situation` or after a time interval of 60 minutes. Several parameters (`correlate`, `termination type`, `quantifier`) allow for more fine-grained control over lifespan initiation and termination.

A restricted version of Amit called SENSID was subsequently used to investigate composite event detection on resource constrained WSN nodes [Kra05]. This potentially allows for expressions defined in Amit to be transparently decomposed and analysed to determine whether parts of them can be detected by SENSID. Distribution of these expressions can help prolong the lifetime of a WSN by reducing unnecessary communication. Some disadvantages of Amit are that it does not allow nested operators within a single complex event, and its time model assumes a globally synchronised clock.

	Operators			
	Sequence	Conjunction	Negation	Counting
Ode	sequence(A,B)	A∧B	A	-
Snoop	A;B	A,B	(A)[B,C]	Any(m, E1,...,En)
Amit	sequence(E1,...,En)	conjunction(E1,...,En)	not(E)	nth(n, E1,...En)

Table 2.5: Sample Complex Event Operators

2.3.2 Continuous Query Languages

Continuous query languages arose to provide expressive and efficient support for queries over *streaming* data tuples. They have much in common with composite event languages given that events also arrive in streams. However, continuous query languages address several aspects of data stream processing that complex event languages ignore. In particular, they usually provide extensive support for the specification of *windows* over input data streams. Windows are usually bounded temporally or by a maximum number of tuples. Tuples that fall outside a window are discarded. However, unlike many complex event languages, continuous query languages do not allow input tuples that cause the generation of an output tuple to be consumed.

Apart from language differences, much of the research into system support for continuous query languages has had a different focus to complex event languages. Several systems provide support for load-balancing, scheduling, and quality of service (QoS) guarantees. Many of these issues are important for pervasive computing applications, but are not fully addressed by complex event detection systems. Fortunately, many of the ideas are transferable.

Of particular interest is research related to the handling of distributed processing of continuous queries. There is a high likelihood of sensor failures and communication problems in pervasive computing environments. Therefore knowledge about a particular query's ability to

tolerate imperfect data streams is extremely useful to have. Several continuous query systems in the literature allow users to stipulate how queries should handle lost, delayed, and out of order inputs. Much of this work was originally proposed in order to enable continuous query systems to shed tuples under conditions of high load while still respecting query semantics.

We now describe some representative continuous query systems from the literature. We provide a more extensive survey in Chapters 3 and 4, where we draw comparisons with our own work.

CQL Arasu, Babu, and Widom [ABW03] provide a precise semantics for continuous query languages as well as an instantiation of the semantics called CQL. Their language provides three classes of operator – stream to relation, relation to relation, and relation to stream. Stream to stream operators can be built by composing these operators together. This enables their semantics to exploit well understood relational semantics.

An example of a query in CQL (taken from [ABW03]) that detects congestion of highways is:

```
Select segNo, dir, hwy
From SegSpeedStr [Range 5 Minutes]
Group By segNo, dir, hwy
Having Avg(speed) < 40
```

Here `SegSpeedStr` is an input stream with schema `SegSpeedStr(vehicleId,speed,segNo,dir,hwy)`. Each tuple in the stream captures the current speed, direction, and location (i.e. segment of highway) of a particular vehicle. The `GroupBy` clause of the query splits the input stream into separate partitions for each highway segment. Each partition stores a five minute sliding window of tuples (i.e. `[Range 5 minutes]`). At every time instant, the query evaluates the average speed of all tuples in each partition, and outputs a congestion tuple for partitions with an average speed greater than forty (i.e. `Having Avg(speed) < 40`).

The Stream query processor [MWA+02] of Motwani et al. is a stream processing system that uses CQL and attempts to perform both static and dynamic approximation of query results in cases of high load or limited resources. This is achieved through intelligent load shedding (e.g. by reducing window sizes or dropping tuples). CQL employs a heartbeat mechanism to deal with delayed events in distributed systems. However, as with most continuous query languages, it relies on a global clock for timestamping.

Aurora Aurora is a data stream management system for monitoring applications from Carney et al [CcC+02; ACc+03]. Queries in Aurora are specified directly as an operator graph. Aurora provides several operators that can be composed to form queries. An example of an expression that applies its aggregation operator to a stream of stock updates (taken from [ACc+03]) is:

```
Aggregate [Avg (Price),
           Assuming Order (On Time, GroupBy StockId),
           Size 1 hour,
           Advance 1 hour]
```

Here the input to the query is a stream of stock events with schema `StockEvent(Time, StockId, Price)`. The `Assuming Order` clause ensures tuples are ordered based on their `Time` attribute, and grouped into separate partitions according to their `StockId`. The `Size 1 hour` and `Advance 1 hour` clauses define a *hopping* window of length one hour and step one hour over these partitions. Thus in contrast to the sliding window of our example CQL query, a hopping window generates output once every hour, and not every time a new tuple arrives.

Aurora allows the specification of *QoS graphs* to control how events should be shed under conditions of high load, and *slack parameters* to help deal with out of order input events. However, like CQL, it does not deal with issues related to distributed time.

2.3.3 Production Systems

Production systems are processing engines that support the specification and detection of condition-action (CA) rules. CA rules result in a more declarative programming model than the ECA rules provided by active databases. Production systems consist of two main parts, a set of facts stored as elements of a working memory (WM) and a set of CA rules. The condition part of a CA rule specifies a pattern over the elements of working memory, whose truth must be monitored by the production system. When a change to WM causes a pattern to hold, the rule associated with it fires, and the corresponding action executes.

In many cases, the conditions of multiple rules are simultaneously matched by a change to working memory. Such a set of rules is termed a *conflict set*. A variety of strategies have been employed to handle conflict sets. Some systems allow the user to assign a priority to rules. Others try to fire the most specific rule, where a rule is deemed more specific than another rule if the other's condition is necessary but not sufficient for its own condition to match.

Production system languages vary in complexity. Some rule languages can have the expressiveness of first order logic; such languages thus allow for cyclic rules, where the action of a triggered rule causes updates that affect working memory elements monitored by its own condition. Programmers must therefore specify rules carefully in order to ensure they are *confluent*, i.e. will terminate eventually. One disadvantage of production systems is that in their general form they are computationally expensive, making it infeasible to employ them on resource constrained devices. Furthermore, determining termination properties of expressions in a Turing-complete rule language is not always possible. Thus for many pervasive computing architectures, acyclic rule sets make it easier and safer to perform distributed detection.

We now give a more detailed overview of two well known production system pattern-matching algorithms from the literature. A broader survey of production system languages can be found

in our discussion of related work in Chapter 3.

Rete The Rete algorithm, originally proposed by Forgy [For82], is a fast algorithm for matching patterns to working memory elements. It was designed for use in the OPS5 production system. The algorithm compiles rules into a *rete network*, also known as a discrimination or token network. Figure 2.3 gives an example of such a network. A rete network consists of nodes of two main types: α nodes and β nodes. Tokens, representing the existence or non-existence of working memory elements, flow through the network. The α nodes contain conditions over a single working memory element. In order to pass an α node, a token's attributes must match the node's condition. β nodes implement joining operators over the tokens output by α nodes. They may have an associated *join* condition. Whenever a new token is added to one of their inputs, the β node evaluates it with respect to the join condition and any tokens on its other input. If they match, then the β node outputs a new combined token. Top level nodes in the network insert the tokens they generate into a conflict set, where a conflict resolution strategy is employed to determine which rule to fire.

The Rete algorithm provides very fast matching for many rule sets. However, it does so at the potential cost of large amounts of memory [WM03]. Rete is an *eager* matching algorithm since it stores intermediate results in the nodes of its network. However in many cases this is a wasted effort, since later changes to working memory may invalidate these results. Furthermore, if many rules are matched simultaneously, and all added to the conflict set, the rule chosen by the conflict resolution strategy may invalidate other potential rule firings in the conflict set. In this case, eager matching of rules leads to further wasted effort and extra memory requirements in order to store the conflict set.

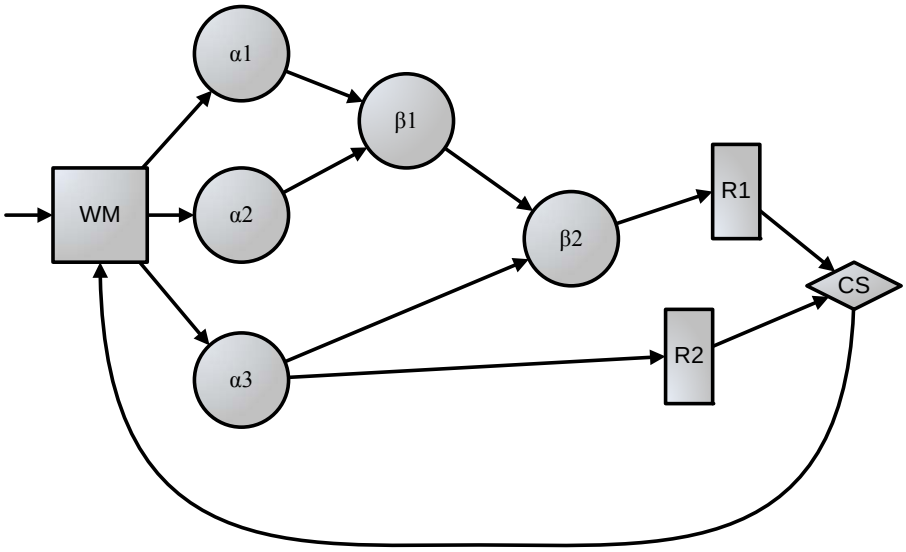

Figure 2.3: A RETE Network

LEAPS Lazy Evaluation for Active Production Systems (LEAPS) [MB90; Bat94] is an alternative production system pattern matching algorithm from Miranker et al. Unlike Rete, LEAPS does not store intermediate matched join conditions for rules in β nodes. This reduces pattern matching memory requirements. Moreover, LEAPS does not attempt to compute a complete conflict set in response to each change in working memory. Instead, it triggers the first rule it can match. This avoids wasting computation and memory resources matching conflict set elements that never cause a rule to be fired. The authors claim that this enables LEAPS to scale to much larger rule sets for many types of application.

The implementation of LEAPS is more complicated than that of the Rete algorithm. Instead of a discrimination network, the main data structure involved is a stack. The addition or removal of an element from working memory pushes a handle to that element onto the stack. Elements also acquire a timestamp indicating the time at which they were added or removed. The matching algorithm examines each rule in turn to see whether the top element of the stack matches part of the rule's condition. If so, it tries to match the rest of the condition against the relevant elements of working memory. If its condition can be matched, the rule fires, and any changes it causes to working memory are added to the stack. If not, the algorithm picks another rule and tries to match it with the top element of the stack and working memory. If no rules are fired, the stack is popped and matching continues using the new top element of the stack. Matching terminates when the stack is empty, indicating a fix point has been reached.

The disadvantage of the LEAPS approach is that the choice of conflict resolution strategy is somewhat limited. In particular, applications that require a conflict resolution strategy that must enumerate all matches in the conflict set are not possible.

2.4 Conclusion

This chapter has outlined several areas of research relevant to distributed complex event detection for pervasive computing. We began by introducing the concept of pervasive computing, including the key characteristics of pervasive computing systems and applications. We also introduced wireless sensor networks, and described how they might affect the design of pervasive computing systems.

Having discussed pervasive computing, we introduced the concept of middleware, including its origins and the current state of the art. We focused in particular on event-based middleware, and how its scalability and data-centric nature make it suitable for pervasive computing applications. We then discussed publish/subscribe systems, a particular type of event-based middleware that allows for highly scalable communication. We also discussed several other relevant classes of middleware, such as data stream management systems, middleware for context-aware applications, and middleware for WSNs.

Following our overview of middleware, we discussed the importance of complex event detection for pervasive computing. We examined how several complex event languages from the literature can be used to specify high-level patterns of events. Finally, we discussed some alternative ways to specify patterns, and how they compare to complex event languages.

Having analysed a variety of middleware and complex event detection languages in the context of pervasive computing, we highlight several issues with existing work. In particular:

- As event publishers may include energy-constrained wireless sensors, support for distribution of complex event detection is vital. Complex event expressions should be easy to decompose, and complex event services should automatically place detectors in suitable locations.
- Since many sensor devices have low quality clocks, the detection model should include support for errors arising from timing uncertainties.
- Language constructs should enable fine-grained control over the resource usage of complex event expressions. This may entail support for the consumption policies of Snoop, the selection paramters of Amit, and the windowing constructs found in continuous query languages. However, any such constructs must interact in a sensible and composable way with conditional expressions across the attributes of input events.
- Given the unreliable nature of many pervasive computing systems, failures that affect complex event detection must be handled in such a way as to minimize their impact on both detection efficiency and correctness. Where these goals conflict, it should be possible for the application to influence any tradeoffs made.

Chapter 3

A Complex Event Language

3.1 Introduction

Pervasive computing is concerned with enhancing applications using information about their environment. It is increasingly common for pervasive computing applications to rely on distributed systems known as Wireless Sensor Networks (WSNs). WSNs consist of large numbers of sensing devices equipped with wireless radios. The key advantages of WSNs over wired solutions are their low cost and ease of deployment.

Many WSN devices have limited computational and storage capabilities and are referred to as *motes*. However, WSNs may also contain more powerful devices, known as *base stations*, scattered throughout the network or deployed at the edge. Base stations can provide a gateway to other networks such as the internet, in addition to performing more heavyweight computations than motes.

Many of the pervasive computing applications built using WSNs are inherently *data centric*. Rather than prescribing the individual devices from which sensor data should be obtained, application programmers wish to specify the data they are interested in, and have an underlying system find and deliver it to them. In addition, programmers often wish to aggregate and combine raw sensor data into more meaningful information. Supporting these requirements in the context of WSNs is especially challenging given their distributed nature, their heterogeneity (in terms of the resources available at different devices), and their limited energy reserves.

Several research projects have attempted to address this problem. One popular abstraction is to treat the WSN as a distributed database, and provide an SQL-like interface to application programmers, with continuous query execution semantics [MFHH03; YG02]. One of the main observations from this work was that by distributing computation in the network significant reductions in communications could be achieved, thus saving energy. These computations were usually in the form of simple filters over readings, or summaries of several readings using SQL-like aggregation functions.

Other researchers observed that many WSN applications are reactive or event-based in nature. Instead of continuously sampling sensor data and sending readings, users may wish to specify *complex events* of interest, and have sensors detect these events locally. Data is only

transmitted when an event occurs. Distributed complex event detection is thus another way to save energy by pushing computation into the network.

These two approaches, complex event detection and continuous queries, have much in common. Indeed, two recent projects have attempted to combine them [JAC04; Riz05]. However, neither of these projects was explicitly targeted towards WSNs. In particular, we believe the constructs they provide for configuring the detection semantics of a complex event (e.g. consumption policies) are not fine-grained enough, and it is not clear how they might interact with conditional expressions across event content. Furthermore, both languages ignore difficulties raised by the lack of a global clock in distributed systems.

In this chapter we attempt to address the shortcomings of previous approaches. Our main contributions are:

- A complex event detection language for pervasive computing applications. Our language is designed to allow easy decomposition and distribution of simple expressions to increase detection efficiency, while still enabling the specification of more sophisticated expressions when needed.
- A discussion of how problems inherent to open distributed systems such as the lack of a global clock affect complex event detection. We use uncertainty intervals to bound the timing imprecision of events, and argue that conflicts between events need to be handled in an application-specific manner. Techniques for enabling applications to handle conflicts are discussed further in Chapter 4.
- A variety of parameters to enable fine-grained control over the constituent events of a complex event. We argue that these parameters are necessary due to the wide variety of detection semantics found in different pervasive computing applications. Increased control over the detection process allows programmers to create expressions deployable on resource-constrained devices.

3.2 Application Scenarios

In this section we outline two motivating scenarios for our work.

3.2.1 Transport Monitoring

The first scenario is in the field of transport monitoring. Solutions here can be broadly divided into those that use information from probe vehicles, and those that use statically deployed sensors in the road. We are primarily interested in the latter, since they do not require individual vehicles to be instrumented with sensor technology.

Many cities currently have deployed induction loop sensors for detecting vehicle presence. However these can be very expensive to deploy, due to the cost of wiring them to power sources. An alternative approach, outlined in [Kna00], is based on deploying battery powered wireless sensors. These sensors can then communicate with a local base station for forwarding data.

The effectiveness of this approach relies on the battery lifetime of the sensors, as the cost of roadworks to replace them is high. Since the biggest drain on power for these sensors is communication, it is extremely important to eliminate unnecessary communication between sensors and the base station. One common way to do this is to push application-specific computation such as data filtering, aggregation, and correlation onto sensor nodes wherever possible.

Our work is motivated by the need to integrate such sensor networks into a large scale infrastructure to support transport information monitoring. For example, the goal of the TIME project [BM07b; BBE+08], is to provide a middleware to enable new sensor technologies to be easily deployed and the information they produce to be shared efficiently with interested participants. As part of this project, we wish to examine how logic specified as part of a distributed complex event processing service (e.g. [PSB04]) provided by the transport monitoring middleware can be safely distributed into resource-constrained wireless sensor networks where appropriate.

3.2.2 Remote Health Monitoring

In a similar vein, one of the goals of the CareGrid project [BM07a] was to provide a secure and privacy preserving infrastructure for remote patient monitoring. At risk patients can be given sensors to wear that monitor a particular aspect of their physical condition. These sensors can then communicate with the hospital when certain patterns of interest occur. Once again, it is important to maximize the battery life of devices attached to patients so that they do not have to recharge them frequently. In the future, sensor devices may even be surgically implanted into patients, further motivating the need to prolong battery lifetimes.

As with transport monitoring, parties interested in the monitoring data may be numerous and widely dispersed. In this paper we focus on applications that access this information using a publish/subscribe communication paradigm. Figure 3.1 illustrates the basic architecture.

3.3 Language

Our language for specifying complex events draws on ideas from several research communities (see Related Work in Section 3.5). Our main motivation was to adapt these languages to support pervasive computing applications in open distributed environments. This section begins with an overview of both our event model and the principal language constructs. We then describe the main steps involved in event detection, and discuss in more detail how they can be controlled by complex event expressions.

3.3.1 Overview

Information about the environment is communicated in our system using *events*. Each event instance has an associated type, which specifies a schema for the data contained in that event.

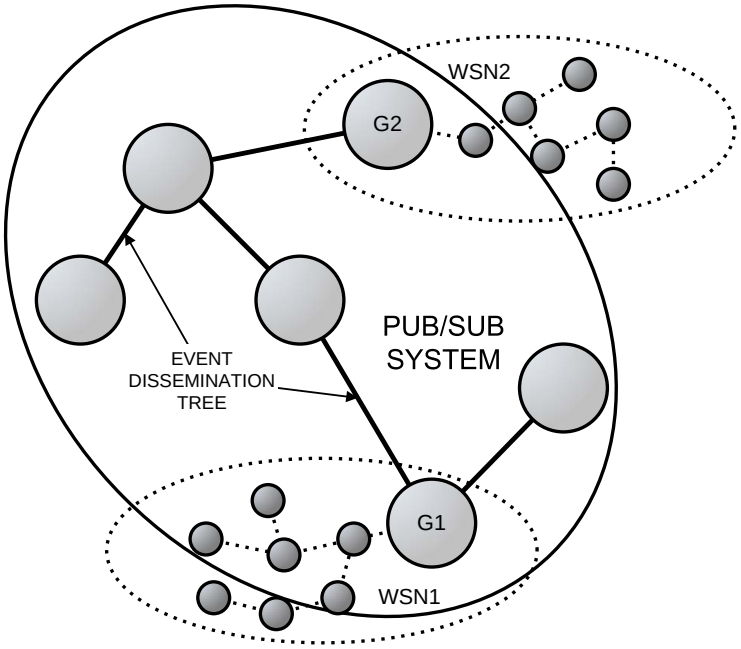

Figure 3.1: WSNs connected via Gateways to a Publish/Subscribe System

Thus at any moment in time, a set of event types are defined in our system. An event type e may be either a primitive event type ($e \in PE$) or a complex event type ($e \in CE$). Primitive events are the lowest level input to the system. Complex events are defined by expressions in our complex event detection language. A complex event takes as input primitive events or other complex events. However, a complex event type cannot use itself as input, either directly or indirectly[1]. The event type namespace thus forms a directed acyclic graph.

Each event instance, whether primitive or complex, has both a start and end timestamp. A start and end timestamp that are the same denote an instantaneous event occurrence. A start and end timestamp with a gap between them denote a durative event occurrence. To deal with imprecision introduced by the lack of a global clock in distributed systems, the start and end timestamps of both primitive and complex events are modeled as *uncertainty intervals*, in the manner of [LCB99]. However, we extend uncertainty intervals timestamps with an optional clock identifier parameter. If specified, a clock identifier should be globally unique. In our system, two uncertainty intervals are concurrent if and only if they span the same time interval and have the same clock identifier. Otherwise, treatment of conflicting uncertainty interval timestamps is application dependent; we allow applications to define detection policies that control the handling of uncertainty intervals that intersect but are not concurrent. Detection policies may for example raise an exception, or perform best effort detection based on endpoints or midpoints of intervals. Note however that we do still allow applications to access uncertainty interval timestamps in expressions.

In our detection model, the input to a complex event detector is a totally-ordered sequence

[1]Enforcement of this constraint is discussed in Chapter 5.

of *stages*, where a stage is a set of events with concurrent end timestamps. In addition to primitive and complex events, a stage may also consist of internal timer events. We define the *stabilization time* of a complex event expression as the timestamp of the current stage. In a distributed system, the stabilization time will generally lag behind the actual physical time due to network delays. Our handling of distribution issues such as delayed event delivery and uncertainty interval timestamps is discussed in more detail in Chapter 4.

The start timestamp of a complex event is determined by the earliest start timestamp of its constituent events. The end timestamp of a complex event is determined by the stabilization time at which it is detected. Thus the end timestamp equals the timestamp of the most recent stage, which usually corresponds to the timestamp of the most recent constituent primitive or complex events. However internal timer events can also trigger complex event detection (e.g. due to event expiry). Thus the end timestamp may also be derived from an internal timer event.

Complex events are defined in our language using an SQL-based syntax, a high-level view of which is given below[1]:

```
CreateCE <ce-name>
Select <select-expression>
[Init <context-initiators>]
Event <complex-event-pattern>
[Where <conditional-expression>]
[Term <context-terminators>]
[GKeys <global-keys>]
[LKeys <local-keys>]
```

Note that the following discussion is just a brief overview, and a detailed description of our language begins in the next section. The `CreateCE` clause defines a unique type name for the complex event expression. The `Select` clause is similar to that of SQL. It enables data contained in the parameters of constituent events to be mapped to the parameters of newly created complex event instances (Section 3.3.5). In addition, in cases where multiple events are created by the detection process, it can be used to define aggregation functions over their parameters. The `Event` clause is the core part of a complex event expression. In it, users can define nested composite event expressions using various temporal, joining, negation and counting operators (Section 3.3.4.2). It also allows fine-grained control over the detection process when necessary using operand specific event selection and consumption parameters. This is necessary to support efficiently the wide variety of event detection semantics found in pervasive computing applications.

The remaining clauses, delimited by square brackets, are all optional. The `Where` clause allows the definition of conditional expressions across the operators in the `Event` clause (Section 3.3.4.2). It is similar to the `Where` clause of SQL. A common requirement in complex event applications is to define a temporal context or window during which a complex event expression

[1] A complete definition of the syntax can be found in Appendix A

should be detected. This is supported directly in our language using the `Init` and `Term` clauses (Section 3.3.3). They define the events that initiate and terminate such a context. Finally, the `GKeys` and `LKeys` clauses allow the definition of *global* and *local* keys across the input events to a complex event expression (Section 3.3.2). Keys split detection into several different partitions, where events in each partition have one attribute with the same value for each key. Global keys specify an attribute of each event that is used by the complex event expression (including initiators and terminators). Local keys must only specify an attribute for each event in the event clause. Partitions can be detected independently, making them simple to decompose for distributed detection. In pervasive computing applications, a common use of keys is to partition the detection process based on location or identity attributes of constituent events.

We present an example of a complex event definition below. As we have not introduced our language in detail, we do not expect the reader to understand all the features it uses just yet. However the example does give an idea of the style of the language. This complex event is based on a scenario described by Chakravarthy et al [JAC04]. A highway is divided up into a number of segments, each with its own `linkId`. Cars, identified by their registration number (`regNumber`), send periodic updates containing their current speed and the link on which they are located. Between the times of 8 am and 6 pm the expression detects any speed limit violation event that is one of the top ten violations observed so far. Separate top tens are maintained for groups of link segments defined by `HighwayRegion1`, `HighwayRegion2`, and `HighwayRegion3` of a highway, and for any violations outside the these highway regions[1]. The expression could be useful for police officers that wish to prioritize the deployment of a speed monitoring squad. The schema of `CarEvent` can be found in Appendix B, together with the input event schemas for all examples presented in the remainder of this chapter.

```
CreateCE Top10SpeedLimitViolations
Select CarEvent.speed as speed, RegionKey.labels as region
Init At8am
Event Unless(CarEvent<stages 1, "speed > 60">,
             AtLeast(10, CarEvent as Others<"speed > 60">))
Where Others.speed > CarEvent.speed
Term At6pm
LKey *.linkId as RegionKey (In R1 as HighwayRegion1, R2 as HighwayRegion2,
                            R3 as HighwayRegion3, Rest as OtherHighwayRegions)
```

Newly stabilized events are processed in several phases by a detector for a complex event expression. In the next sections (i.e. Sections 3.3.2 to 3.3.5), we describe the language features above in more detail, and how they control event detection. Our exposition of these language features is structured according to the order in which they process input events.

A complex event expression with global and local keys divides input event streams into separate partitions, and may divert each partition to a separate detector. Hence we first discuss

[1]For simplicity, we use the notation `Ri` here as a placeholder for a region. In an actual expression, the link identifiers in each region must be specified explicitly.

global and local keys and the detection partitions they cause (Section 3.3.2). Within each partition, detectors examine events to see whether they terminate a detection context. A detection context is essentially a temporal window within which the remainder of a complex event is detected. We thus follow our discussion of detection partitions with a discussion of detection contexts (Section 3.3.3). Note detectors examine events to determine whether they *initiate* a new context only when they do not terminate an active context and are not otherwise used for event detection. However since initiation and termination are both relevant to the concept of detection contexts, we discuss them together.

The remainder of our discussion is concerned with what might be viewed as the "traditional" part of the event processing model (Section 3.3.4). A detector adds events to the appropriate operands of the tree of operators defined in the `Event` clause. Events output by an operator bubble up the tree to become input events for operators higher in the tree. Events output by the top level operator constitute detections of the complex event expression. However, before the detector outputs them, it first evaluates the `Select` clause to determine what data contained in the constituent events should be retained in the final output. Note that as we mentioned above, during the final phase of event processing the detector checks whether any unused input events initiate a new detection context (i.e. if they did not already terminate a context and were not added to an operand of an operator). However for clarity of exposition we discuss initiation and termination of detection contexts together.

3.3.2 Detection Partitions

As we described in the overview of our language, each complex event expression may have optional lists of *global and local keys* associated with it. These keys effectively split the complex event into several *detection partitions*. Partitions define disjoint subsets of the input events of a complex event expression. We perform detection independently within each partition. This simplifies distribution of complex event detectors. It also reduces the complexity of event specification, since interactions between events in different partitions do not need to be considered in the `Event` or `Where` clauses.

A key definition *key-def* is a pair:

$$(key, ranges)$$

The *key* element of a definition is simply a list of attribute identifiers of the form *event-name.attr-name*. All attribute identifiers specified in a key must have the same type.

Given a complex event, a *local key* specifies a single attribute from every leaf operand in the `Event` clause of the expression. *Global* keys must in addition specify an attribute for each initiator and terminator defined in the `Init` and `Term` clauses. The attributes specified for initiator and terminator events by a global key must also have the same type as the attributes of operands in the `Event` clause. If the `Event` clause contains nested operators, then both local and global keys must contain an entry for every initiator and terminator of all inner operators. Thus for an inner operator (i.e. any operator that is not the root of the operator tree) a local

key must contain a reference to the same events as a global key. Multiple local or global keys may be defined over different attributes of the events in an expression. In many cases, several events in a key may have the same attribute name. As a shorthand notation, we allow all events with a particular attribute name in an expression to be referred to using `*.` *attribute-name*.

The set of global keys thus defines an m-tuple of attributes for each event in the expression. Similarly, the local keys define an n-tuple for every event, excluding initiators and terminators. At run-time, the m-tuple defined by the global keys results in a set of *global partitions*. Within each global partition, the n-tuple defined by the local keys results in a set of *local partitions*. If no global keys are defined, then there is a single, universal, global partition. Similarly, if no local keys are defined then each global partition contains only a single local partition. Note the m-tuples and n-tuples of different operands may refer to different attribute names.

To clarify the difference between global and local keys, consider the example expressions `GloballyPartitionedViolations` and `LocallyPartitionedViolations` below:

```
CreateCE GloballyPartitionedViolations
Select CarEvent.speed, CarEvent.linkId
Init PoliceOnDuty
Event Unless(CarEvent<stages 1, "speed > 60">,
             AtLeast(10, CarEvent as Others<"speed > 60">))
Where Others.speed > CarEvent.speed
Term PoliceOffDuty
GKey *.linkId

CreateCE LocallyPartitionedViolations
Select CarEvent.speed, CarEvent.linkId
Init PoliceOnDuty
Event Unless(CarEvent<stages 1, "speed > 60">,
             AtLeast(10, CarEvent as Others<"speed > 60">))
Where Others.speed > CarEvent.speed
Term PoliceOffDuty
LKey *.linkId
```

These expressions are variations of the `Top10SpeedLimitViolations` event we defined in Section 3.3.1. As with `Top10SpeedLimitViolations`, they both generate an event whenever they receive a car event whose speed is within the top ten previously observed speeds. However, instead of monitoring speeds for several predefined regions, these expressions maintain a separate top ten list for each distinct link. With respect to the difference between local and global partitions, a more important distinction is that `LocallyPartitionedViolations` and `GloballyPartitionedViolations` events may only occur within a detection context (see Section 3.3.3) initiated by a `PoliceOnDuty` event and terminated by a `PoliceOffDuty` event. Since `GloballyPartitionedViolations` has a global key, it only monitors a link for speeding events

when a police car is on duty *and located at that link*. In contrast, `LocallyPartitionedViolations` has a local key, and thus will monitor a link for speeding events so long as there is a police car on duty *at any link*.

Key definitions may contain an optional `In` sub-clause. When this is left unspecified, every event instance within a partition will have the same values for its keyed attributes as the corresponding attributes of every other event in the key. For example consider the expression:

```
CreateCE Top10SpeedLimitViolations
Select CarEvent.speed as speed, CarEvent.linkId as linkId
Init At8am
Event Unless(CarEvent<stages 1, "speed > 60">,
             AtLeast(10, CarEvent as Others<"speed > 60">))
Where Others.speed > CarEvent.speed
Term At6pm
LKey *.linkId
```

This example is similar to the introductory example we gave in Section 3.3.1. However, instead of detecting the top ten speeding violations within each region, it detects the top ten speeding violations separately for every highway segment.

In contrast, the `In` sub-clause of a key allows the user to specify explicitly a finite set of partitions. This enables partitions to contains groups of event values, instead of a single value. Each element of the `In` sub-clause must define a subset of the type-space of the attributes in its associated key. If defined, these subsets are stored in the *ranges* parameter of the key definition.

A keyword `Rest` may also be specified as an element of the `In` sub-clause, which defines a partition containing any portion of the attribute type-space not covered by the other elements. Consider for example the `In` sub-clause definition `In R1, R2, R3, Rest` (similar to the one we used in the introductory example of Section 3.3.1). Here, `R1`, `R2`, and `R3` define geographic regions. If we add this to the end of the global key expression in the previous example, four global partitions will be created. Events in the same partition need no longer have equal values for their `linkId` attribute. Instead they must simply fall within the region of the partition. The `Rest` partition contains events that occur outside any of the other regions. Labels may be attached to each of these regions using the keyword `as`. In addition, keys can be named locally using `as`. This enables the name of the partition in which an event is detected to be referenced within the scope of the complex event definition.

3.3.3 Detection Contexts

For every detection partition defined by its keys, a complex event is detected within a particular *detection context*. A context defines a time interval during which a complex event should be detected. Each context definition consists of a set of *initiators* and a set of *terminators*. Initiators and terminators define when the context becomes active and inactive respectively. An initiator

can be an event or the keyword `Start`, indicating the time at which the complex event expression is created. Terminators can be either events or a time duration after the occurrence of the initiator. An event initiator definition is a tuple:

$$(type, as, threshold, icc)$$

Similarly, an event terminator definition is a tuple:

$$(type, as, threshold, cross\text{-}threshold, tcc)$$

Apart from the type-name of the event, there are several optional parameters that affect the semantics of an initiator or terminator. A label *as* may be defined for the initiator (terminator) in order to distinguish between initiators, terminators, and operands of the event pattern with the same type. For example, the initiator clause `Init HeartRateEvent as InitHR` assigns the label `InitHR` to an initiator event of type `HeartRateEvent`. A label is only visible within the scope of a complex event expression. The *threshold* parameter allows a condition to be specified to filter initiators and terminators based on the values of their attributes. An initiator threshold may refer only to the attributes of the initiator. Similarly, the threshold of a terminator may refer only to the attributes of the terminator. In addition, we allow users to specify a *cross-threshold* for a terminator that may refer to the attributes of both the initiators and the terminator. Allowing a terminator of a context to be parametrised by the initiator of the context is quite useful for many applications.

If multiple initiators occur before the occurrence of a terminator, the default semantics is to ignore them. However, this can be changed by setting the *icc* (initiator correlation code) parameter of an initiator to `add`. This allows multiple detection context *instances* to be open in parallel. Each context instance evaluates the enclosed composite event expression independently of other open contexts instances. When multiple contexts exist, an additional *tcc* (terminator correlation code) parameter determines which contexts are closed and consumed by a terminator. It may have a value of `new` (terminate the most recent context that matches the terminator), `old` (terminate the oldest context that matches the terminator), or `each` (terminate all contexts if any matches the terminator).

Events may happen concurrently, and the same event may match more than one initiator, terminator, or event-clause operand. We must define a clear evaluation order in these cases to provide an unambiguous detection semantics. At each timestep, a set of event instances relevant to a complex event expression may be stabilized. These event instances are firstly evaluated as potential terminators for any open contexts. Next, open contexts *collect* any events that may act as constituents of their composite event expressions. This process is described in more detail in the next section (Section 3.3.4.1). Finally, events are evaluated to determine whether they initiate any new contexts. A simple way to remember this evaluation order is that contexts are closed intervals, and an event can act in only one role out of initiator, terminator or constituent event with respect to a single context *instance*.

An event instance may match more than one initiator (terminator) during the evaluation

process. In this case, the event is evaluated against the initiators (terminators) of the `Init` (`Term`) clause in left-to-right order. Finally if multiple events occur concurrently, they are all evaluated against a single initiator (terminator) before being evaluated against the next initiator (terminator). Thus in general, the initiator (terminator) that opens (closes) a new context instance will be a set of events that matched the initiator (terminator).

An example of a detection context with an *icc* parameter of `add` is given below:

```
CreateCE MaxHeartRateChange
Select ((Max(InitHR.value Union Set(EventHR.value))
         - Min(InitHR.value)) / Min(InitHR.value))
         as max-pct-increase
Init HeartRateEvent as InitHR<add>
Event HeartRateEvent as EventHR
Term 24hrs <old 1>
```

Over a sliding twenty four hour period, this complex event continuously tracks, *with respect to each heartrate event*, the maximum percentage increase in heartrate. Note that `InitHR` may be a *set* of heart rate events. This kind of trend analysis is one of the main motivations for allowing multiple detection contexts to be active at the same time.

An example of a detection context with a threshold condition across the initiator and terminator is given below:

```
CreateCE CarAvgSpeedPerLink
Select ValidCarEvent.regNumber as regNumber,
       ValidCarEvent.linkId as linkId,
       Avg(LinkInit.speed Union ValidCarEvent.speed) as avgSpeed
Init ValidCarEvent as LinkInit
Event Or((ValidCarEvent)<deferred>, Not(ValidCarEvent))
Term ColocatedError, ValidCarEvent as
     LinkTerm <Max(LinkInit.linkId) != Max(LinkTerm.linkId)>
GKey *.regNumber
```

This complex event is based on a similar scenario to that used in our introductory example in Section 3.3.1. A highway is divided up into a number of segments, each with its own `linkId`. Cars, identified by their registration number (`regNumber`), send periodic updates containing their current speed and the link on which they are located. The expression computes the average speed over each link for each car. A context is terminated when a car moves onto a new link, as specified in the terminator cross-threshold condition (`Max(LinkInit.linkId) != Max(LinkTerm.linkId)`). Since the detection mode is `deferred`, the detector outputs the average speed of a car over a link when the context is terminated by the car moving onto a new

link. The global key ensures that detection is performed separately for each car[1].

`ColocatedError` and `ValidCarEvent` are defined as:

```
CreateCE ColocatedError
Select A.regNumber as regNumber
Event Concurrent(CarEvent as A<stages 1>,
                 CarEvent as B<stages 1>)
Where A.linkId != B.linkId
LKey *.regNumber

CreateCE ValidCarEvent
Select A.regNumber as regNumber,
       A.speed as speed,
       A.linkId as linkId
Event Unless(CarEvent as A <stages 1>,
             CarEvent as B <stages 1>)
Where A.linkId != B.linkId AND end-ts(A) = end-ts(B)
LKey *.regNumber
```

These expressions are used to clean the input when a car gives position readings indicating it is on two or more different links at the same time. This results in the termination of any open context, with the conflicting readings being ignored. A `ValidCarEvent` is any position report that does not give conflicting positions for the car. This could be extended easily to filter out concurrent reports giving different speeds.

3.3.4 Event Patterns

The main part of a complex event expression is the composite event pattern described using the `Event` and `Where` clauses. The event clause allows the specification of a set of operators that may be composed to detect composite event patterns. The `Where` clause allows the specification of a condition across the operands (i.e. constituent events) of these operators.

In contrast to the composition of whole complex event expressions, operator composition within an event pattern enables the conditional detection of high-level events from primitive events. Where no condition across operators is required, a choice between these two approaches may exist. When possible, creating a single complex event expression with a tree of operators in the `Event` clause is generally easier to code. However, it is usually easier to decompose and reuse multiple complex event expressions than a single complex event with an internal operator tree. In addition specifying an operator tree using multiple complex events allows more fine-grained control over detection. This is because the collection, selection, and consumption parameters

[1]Note that since the top level event is `Or`, termination of the context will still result in its deferred sub-contexts being evaluated.

we provide to enable this control may only be specified for the leaf operands of an operator tree (as discussed in the following sections).

Our design couples support for conditions across multiple operators with expressive configuration parameters for individual operands (e.g. windows, consumption). Previous work either does not allow conditions across multiple operators, or do not provide mechanisms that allow for fine-grained control over the detection semantics. The former precludes expressions such as the `Top10SpeedLimitViolations` complex event (Section 3.3.1). The latter hinders the creation of many of the complex events we introduce in the following sections (e.g. `BagContainerID in Section 3.3.4.3`).

Informally, the event pattern of a complex event is a tree of operators, where internal operators also have detection contexts. Thus an event pattern definition *ep* is a tuple:

$$(\textit{op-name}, \textit{operand-defs}, \textit{detect-mode}, \textit{params})$$

The *op-name* specifies one of the operators in our language (discussed in the following sections) The *operand-defs* element is a list of definitions for the operands of the operator. The *detect-mode* of an event pattern may be either `immediate`, `delayed`, or `deferred` (discussed in the following sections). Finally, *params* contains various operator-specific parameters (e.g. the value of n for counting operators).

An operand definition in *operand-defs* may be either a leaf operand definition or an internal operator definition. A leaf operand definition *leaf-def* is a tuple:

$$(\textit{type}, \textit{as}, \textit{threshold}, \textit{collect}, \textit{quant}, \textit{consume}, \textit{cond})$$

The *type* element is the name of a primitive event or another complex event. A label *as* may be defined to distinguish between operands with the same type. This is similar to the label for event initiators and terminators. The *threshold* element is the threshold condition defined for the operand. The *collect*, *quant*, and *consume* elements control event collection, detection, and consumption respectively for the operand (as discussed in the following sections). Parts of the `Where` clause that reference the operand are stored in *cond*.

An operand that is not a leaf must be an internal operator of the event pattern. An internal operator definition *internal-operator-def* is a tuple:

$$(\textit{sub-ctxt}, \textit{sub-ep}, \textit{sub-gkeys}, \textit{sub-lkeys})$$

Here, *sub-ctxt* defines the detection context within which the nested operator should be detected. It is a pair consisting of a list of initiators and a list of terminators. A sub-context may only become active if its parent context is active. The *sub-ep* element gives the definition of a nested event pattern. It is of the same form as *ep*. Thus an event pattern definition can be a nested data structure. The *sub-gkeys* and *sub-lkeys* elements represent any keys defined within internal event patterns.

Detection of an event pattern is performed in three phases: the *collection* phase, the *detection* phase, and the *consumption* phase. The collection phase is responsible for adding newly arriving

input events to operands, and for removing any events from leaf operands that are no longer relevant due to the advancement of windows over that operand. The detection phase tries to generate all events that match the event pattern using the input events stored in leaf operands. Finally, if the expression requires it, the consumption phase removes from leaf operands the constituent events of any events generated during the detection phase. We now describe the purpose of each phase, and how users can control them, in more detail.

3.3.4.1 Collection Phase

Every time a new event arrives at the detector for a complex event, and the event does not terminate a context, it must be passed to every leaf operand in the `Event` clause whose type is the same as the event instance. Each leaf operand maintains an ordered list of the event instances that may still contribute to a complex event detection. The task of the collection phase is to determine whether a particular event should be added to each list, and if so, whether any existing members of the list should be overwritten.

The collection of an event is done for each operand independently, and the same event may be added to more than one operand. For each operand, a threshold condition (similar to the threshold for context initiators) can be specified over its event attributes. Event instances that match the threshold are added to the operand's list.

To enable bounds to be put on the number of events that should be considered for pattern detection, we allow the specification of several optional collection parameters. The syntax of these parameters is summarized below:

```
[(rows | stages) n][range t]
```

The simplest of these is the `rows` n parameter. This defines an upper bound on the number of tuples the list may contain. When the addition of new events causes this bound to be exceeded, old events are removed until the size of the list equals the maximum number of rows. The ability to specify such upper bounds is especially important when detection is to take place on resource-constrained devices.

In the presence of concurrent events, the specification of a maximum list size using `rows` n may lead to some events that match the threshold condition never being used for detection. This can happen when a number of concurrent events exceeding the maximum list size are added. To prevent this, while still allowing the user some control over the size of operand lists, we allow the specification of a `stage` n parameter *instead* of `rows`. With respect to an operand list, a *stage* is defined as any time instant for which there is an event in the list with the same end timestamp. An operand list with a `stage` upper bound of n requires the number of different stages at which events in the list occurred to be at most n. This is similar to the notion of stage used in the Datalog$_{1S}$-based composite event language of Motakis et al [MZ95a].

The two parameters above essentially enable the specification of row or stage based *sliding windows* over each operand. In addition we provide the ability to specify temporal sliding windows over event operands. This is achieved by allowing an optional `range` t parameter to be

specified for each operand. When an event is added to the operand's list, a timer is started based on the expiry time for that operand, as defined by the time interval t. When the timer fires the event is removed from the list.

Note that the windows above are defined on the level of an operand, are of fixed size, and can only slide. In contrast, the temporal windows defined by detection contexts may be event bounded, and are defined over all the operators they contain.

3.3.4.2 Detection Phase

In the second phase of composite event processing, detection of the event pattern is performed. The event pattern consists of a tree of event operators, selection parameters for each leaf operand of these operators, and an optional condition across the operands of the tree (as specified by the `Where` clause). Nested operators may have their own detection context. However a nested context may only be active when all its parent contexts are active. The `Where` clause may not reference initiator or terminator events of a context at any level.

A list of operators we currently support together with a brief description of their semantics is given below. Some of the operators are similar to those proposed by Amit [AE04]. Unless otherwise specified, each operand E can be a leaf operand or an internal operator.

- `Sequence`$(E_1, .., E_k)$ - Detects a sequence of events $E_1 \ldots E_k$ such that $\forall i,\ 1 \leq i < k$, `end-ts`$(E_i) <$ `end-ts`(E_{i+1}).
- `And`$(E_1,..,E_k)$ - Detects a conjunction of events $E_1 \ldots E_k$ in any order.
- `Or`$(E_1,..,E_k)$ - Detects a disjunction of events $E_1 \ldots E_k$.
- `Concurrent`$(E_1,..,E_k)$ - Detects the occurrence of events $E_1 \ldots E_k$ such that $\forall i, 1 \leq i < k$, `end-ts`$(E_i) =$ `end-ts`(E_{i+1}).
- `Not`(E) - Detects the non-occurrence of the event E during the detection context enclosing the operator.
- `Unless`(E_1,E_2) - Detects the occurrence of E_1 so long as no E_2 has occurred, i.e. it can be read as E_1 unless E_2.
- `At`$(timePattern)$ - Triggers an event at any time matching $timePattern$. A time pattern is a string of the form $yyyy/mm/dd/hh : mm : ss.mmm$, indicating a point in time. Time patterns may contain wildcards (e.g. ****/**/**/12 : 00 : 00.000) triggers an `At` event at midday every day).
- `Every`$(timeInterval)$ - Periodically triggers the occurrence of an event using a period specified by $timeInterval$. The periodic timer starts at the time when the enclosing detection context is opened.

- `After(`E`[`$size$`,`acc`],`$timeInterval$`)` - Detects the occurrence of an event at a time interval $timeInterval$ after the occurrence of the event E. The optional `[`$size$`,`acc`]` parameter specifies how multiple events occurring within the $timeInterval$ should be handled. The syntax of the $size$ element is `(rows | stages) n`, and the syntax of acc (after correlation code) is `add | ignore`. The $size$ parameter limits the number of events that can be retained whose $timeInterval$ has yet to expire. If a newly arriving event exceeds this limit, and acc is `add`, it replaces the oldest event whose $timeInterval$ has not expired. If acc is `ignore` and the limit is exceeded, the event is discarded. Note that E must always be a leaf operand.

- `Nth(`n`,`E_1`,...,`E_k`)` - Counts the number of events $E_1 \ldots E_k$ that have occurred, and triggers an event when the total equals n. An optional weight parameter may be specified for each operand such that the number of instances of an operand is multiplied by its weight when counting.

- `AtLeast(`n`,`E_1`,...,`E_k`)` - This operator is similar to `Nth`, except it detects an event when the total is greater than or equal to n.

- `AtMost(`n`,`E_1`,...,`E_k`)` - This operator is similar to `Nth` and `AtLeast`, except it counts the number of events $E_1 \ldots E_k$ that have occurred within a detection context, and triggers an event when the total is less than n.

Two parameters provide additional control over the detection semantics of these operators. The first of these parameters, the *detection mode*, may have a value of `immediate`, `deferred`, or `delayed`. An `immediate` detection mode ensures event detection is performed when events are added to an operator's operands. In contrast, a `deferred` detection mode waits until the end of the detection context before executing the detection algorithm. Finally, a `delayed` detection mode is similar to the `immediate` detection mode in that event detection is performed as soon as events are added. However, detected events are stored until the end of the detection context before being delivered. This is useful in some cases where the events detected during a context are to be viewed together (e.g. as part of a summary of the events that occurred during the context).

There are a few restrictions over which detection modes can be defined for some operators. In particular, a `Not` operator may only be detected using a `deferred` detection mode, and temporal operators (i.e. `At`, `Every`, and `After`) may only be detected in `immediate` mode. Apart from `Not`, the default detection mode for all operators is `immediate`. Finally, there are some limitations to the detection modes that can be specified for nested operators. A `deferred` operator cannot be nested within another `deferred` or `delayed` operator, and a `delayed` operator cannot be nested at all (i.e. it must be at the top level of any operator tree of which it is a part). These restrictions on nesting of detection modes are a limitation of our current implementation.

The second parameter that affects event detection, the selection *quantifier*, is specified at the operand level. The quantifier tells the detection algorithm which events should take part

in the detection of a complex event, and can be used to detect a wide range of event patterns. Allowing quantifiers to be specified at the operand level enables fine-grained control over the event detection process. This is in contrast for example to the global consumption policies of Snoop [CM94; CKAK94]. The syntax of a quantifier is:

`all | ([strict] (new | old) (max` n `| +) [staged])`

A quantifier of `all` requires that all events in an operand's event list match every event to which they are compared from other operands event lists. Thus when an event from one operand's list is being matched against an operand with a quantifier of `all`, and it fails to match one of the events in that operand's list, detection backtracks. The alternative part of a quantifier contains several flags. The `(new | old)` part of a quantifier determines whether events are selected from newest to oldest or vice versa. The `(max` n `| +)` part of a quantifier specifies that up to n events, where $n \geq 1$, may be selected from this operand in the case of `max` n, or that as many events as possible should be selected in the case of `+`. A `strict` flag indicates that any events selected must occur consecutively from the start (in the case of `new`) or end (in the case of `old`) of the list. This is useful in placing hard bounds on the amount of processing required when conditions across operators are defined in the `Where` clause. Finally, the `staged` option allows n to refer to a number of stages (as discussed for the collection phase) instead of event instances. Examples of some valid quantifiers are `new max 5` or `strict old +`. The default selection quantifier is `new +`. In addition, we note that quantifiers may only be specified for leaf operands of the operator tree. Internal operands are always detected as if they used the default quantifier.

Having discussed the operators and optional control parameters of the `Event` clause, we now describe how the `Where` clause of a complex event expression may be used to specify conditional composite event patterns. The conditional expression in a `Where` clause may refer to the attributes of any of the operands of a composite event pattern, subject to some *scope restrictions* which we discuss at the end of this section. Conditions consist of the standard relational (`=`, `!=`, `<`, `>`, `<=`, `>=`) and logical (`AND`, `OR`, `NOT`) operators. Note however that the `NOT` operator used in conditions is a logical operator, and differs from the `Not` operator for an interval defined earlier.

There are some restrictions on the events that can be referred to in the `Where` clause. Firstly, the `Where` clause may not refer to the initiators and terminators of detection contexts. Secondly, we impose some *scoping constraints* on the use of conditions across some operators. These constraints only affect the negation operators (i.e. `Not` and `Unless`) and the counting operators (i.e. `Nth`, `AtMost`, and `AtLeast`). For the `Not` operator, no condition can be defined that contains a relational comparison between an operand internal to the `Not` operator and an operand that *follows* it in the composite event pattern. Intuitively, one operand follows another if it is evaluated after it by the detection algorithm. We define this concept more formally in Section 3.4. A similar restriction is enforced for the `Unless` operator, except that it only applies to the *second operand* (i.e. the negated operand). Finally, the same following restriction is enforced for each of the counting operators.

We provide several built-in functions that extend the expressive power of the `Where` condition. For example, the start and end timestamps of constituent events may be accessed using the functions `start-ts(E)` and `end-ts(E)` respectively. The ability to restrict event detection based on start timestamps raises the possibility of supporting the detection of composite events using a *durative* detection semantics [AC03; YB05]. For example, consider the case where we wish to detect an elderly patient who is bleeding after having taken a fall:

```
CreateCE PatientBleeding
Select Fall.patientId
Event Sequence(Fall<rows 1, range 5 mins, consume>,
               And(HeartRateIncrease as HRInc <max 1>,
                   BloodPressureDecrease as BPDec <max 1>) as
                      Bleeding)
Where start-ts(Bleeding) > end-ts(Fall) AND
       NOT(end-ts(HRInc) < start-ts(BPInc)
           OR end-ts(BPInc) < start-ts(HRInc))
```

Bleeding is indicated by a decrease in blood pressure (`BPDec`) and an increase in heart rate (`HRInc`) overlapping in time, but only if they both occur strictly after a `Fall` is detected (e.g. by an accelerometer). The `Where` clause contains appropriate restrictions over the timestamps of the different operands.

For complex event expressions involving durative events and the `Unless` operator, some additional issues may arise. Consider for example the detection (in `immediate` mode) of the event:

```
CreateCE BloodPressureProblem
Select BP.level
Event Unless(BloodPressure<level = high> as BP,
             SignificantHeartRateIncrease as SHRI)
Where start-ts(SHRI) <= end-ts(BP) AND
      end-ts(SHRI) >= start-ts(BP)
```

This expression signals a possible problem with a patient's blood pressure whenever a high blood pressure event occurs that was not caused by an increase in the patient's heart rate. However, the `SignificantHeartRateIncrease` (`SHRI`) event is durative. In some cases, an `SHRI` event may overlap with a `BloodPressure`(`BP`) event, but not occur until after it (i.e. `end-ts(SHRI) > end-ts(BP)`). This may result in a `BloodPressureProblem` event being signalled erroneously. In general, we do not even guarantee the start timestamps of events will increase monotonically. This is because if an operand contains several events, and the operator is detected conditionally, different constituent events may be selected every time the detection algorithm is executed.

To prevent these problems from arising, we must know the earliest possible start timestamp of any `SHRI` event that may be received in the future. We can obtain this information using the

function `sst(E)` (start stable-time). This enables us to extend the earlier condition to `Where ... AND sst(SHRI) > end-ts(BP)`. Note that the start stable-time is a property of the *operand* `SHRI`, and *not* of a particular event-instance. Ideally, detection should be performed using an *active* detection semantics (similar to the active expiration semantics suggested by Bai et al [BTW+06]), where lower level detectors can notify subscribers of the advancement of their start stable-time independently of any event occurrence. Currently however, our language only provides an *inactive* detection semantics. This means that a `BloodPressureProblem` event is only triggered in response to a later `BP` or `SHRI` event that advances `sst(SHRI)`.

One additional use of the `Where` clause is to allow the specification of *incremental* output. Consider the case where we wish to generate the event `And(A,B)`, where there are no restrictions on the size of either operand list (i.e. they both have a `row` parameter set to `unbounded`). Every time a new event is added to either operand, a cross-product is performed. In many cases, we would like to ensure that only events not detected during previous stages are generated. Repeated events can be prevented by specifying a condition `Where end-ts(A) = stable-time OR end-ts(B) = stable-time`. In this condition, `stable-time` is a built-in function that returns the current stable-time for the whole complex-event expression.

3.3.4.3 Consumption Phase

In the third phase, the consumption phase, events may be removed from operand lists in response to a complex event being triggered. This can be controlled through a consumption flag for each leaf operand of the `Event` clause. If an event is to be consumed, a value of `consume` is specified. Otherwise, a value of `!consume` should be specified. The default value is `!consume`[1].

Below, we give an example of a complex event expression in which events are consumed:

```
CreateCE BagContainerID
Select BagEvent.id as bagId,
       ContainerEvent.id as containerId
Event And(BagEvent<old rows max 4, consume>,
          ContainerEvent<old rows max 1, consume>)
LKey BagEvent.carouselId, ContainerEvent.trackId
```

In an airport, each gate has a baggage carousel and a container track. Bags move along the carousel until they meet automated baggage containers. Up to four bags are loaded into each container, which then moves along a track towards the appropriate gate. The identities of bags and containers are scanned in order before loading as they move along the carousel and track respectively. The complex event generates a record of the container of each bag. The `old rows max 4` selection quantifier over `BagEvent` and the `old rows max 1` selection quantifier over `ContainerEvent` indicate that bags are packed into containers in FIFO order, and that at most four bags may be loaded into a container. Containers that arrive when there are no

[1] This example is similar to the `PackageContainer` example in the next chapter

bags wait until at least one bag arrives before moving along the track. Finally, the local key partitions the detection by gate. This indicates that the `carouselId` attribute of `BagEvent` and the `trackId` attribute of `ContainerEvent` are equivalent for each gate.

It is important to note that a detector only *consumes* an input event when it contributes to the occurrence of an output complex event (assuming the consumption flag of the corresponding input operand is set to `consume`). Thus in the example above, the detector only consumes a `BagEvent` or `ContainerEvent` when they contribute to a new `BagContainerID` event. We emphasize however that consumption is not the only mechanism by which a detector may delete input events. In particular, detectors may delete events to windowed operands as their window advances. Similarly, a detector may delete events when a detection context terminates. In contrast to consumption driven event deletion, window and detection context driven event deletion may occur independently of whether any new output complex events are generated. As might be expected, the manner in which a particular detector deletes input events has direct implications for its resource usage. As we will discuss in Chapter 4, this is especially true when input event streams are unreliable.

3.3.5 Complex Event Mapping

The `Select` clause of a complex event controls how the information contained in events generated by the detection phase is mapped to newly created instances of the complex event. This includes the ability to specify aggregation functions over sets of events. It is similar to the `Select` clause of an SQL statement, and we thus refer to this stage as the *projection* phase. We refer to the expression defined in the `Select` clause as a *mapping expression*. There are two types of mapping expression, a *tuple* mapping expression and a *group* mapping expression. The former maps each instance of the event pattern to separate instances of the complex event. The latter maps all instances of the event pattern that are detected at a stage to a single complex event.

Formally, a mapping expression consists of a list of attribute definitions, where an attribute definition is a tuple:

$$(\textit{attr-name}, \textit{attr-expr})$$

Each *attr-name* defines the name of an attribute of the complex event[1]. An *attr-expr* specifies how the attribute value is computed from the parameters of constituent events.

In a tuple mapping expression, all *attr-expr* are *tuple-attr-expr*. A *tuple-attr-expr* can refer to an attribute of a constituent event using its attribute identifier (i.e. *event-name.attr-name*). It may also refer to these attribute identifiers within aggregation functions. We support the standard SQL functions of `Max`, `Min`, `Sum`, `Avg`, and `Count`, as well as a median aggregation function `Median`. In many cases, there will be a single constituent event instance per attribute identifier, making the aggregation functions redundant. However, if the attribute identifier refers to an operand of a counting operator, an event initiator, or an event terminator[2], there may be multiple constituent event instances. Thus a *tuple-attr-expr* may only refer to such an attribute

[1] Attribute names must be unique within a *map-expr*.

[2] Obviously, terminator events are not available to an event detected in `immediate` mode.

identifier within an aggregation function.

When the mapping expression is a grouped mapping expression, then all its *attr-expr* must be *group-attr-expr*. Since there may be multiple events within a group, a *group-attr-expr* may only refer to an attribute identifier within an aggregation function. A *group-attr-expr* must refer to each attribute identifier or event name using either the `Set` or `Bag` function, where `Set` removes any duplicate values. The aggregation function will be performed over the set (or bag) of attribute values extracted from all event instances within the group. Similarly, an event name may be referred to using `Set` or `Bag`, but must be aggregated using the `Count` aggregation function. If the user does not specify which function to use, the `Set` function is applied by default. Note that in the case where the attribute identifier refers to a counting operand, an event initiator or an event terminator, nested aggregation functions must be specified. The inner aggregation function aggregates the set of constituent event instance attribute values within each tuple. The outer aggregation function then aggregates this group of values. An example of a grouped mapping expression is given in the `Select` clause of the complex event expression below:

```
CreateCE SmoothedSpeeds
Select Avg(Set(CarEvent.speed)) as car-avg-speed,
       Median(Bag(BusEvent.speed)) as bus-med-speed
Event And(CarEvent<range 10 mins>, BusEvent<range 10 mins>)
Where end-ts(CarEvent) = stable-time OR
      end-ts(BusEvent) = stable-time
```

At most one `SmoothedSpeeds` event is produced per stage, and only distinct `CarEvent.speed` values are used to calculate `car-avg-speed`[1]. In contrast, multiple identical `BusEvent.speed` values may be used to calculate the `bus-med-speed`.

In contrast to the `Where` clause, the `Select` clause may refer to the initiators and terminators of detection contexts. In cases where a different event is responsible for initiating or terminating the context than the one referenced, a `Null` value is returned as the value for the corresponding attribute of the newly detected event[2]. A `Null` value may have an undesirable effect on the value returned by aggregation functions. By default, we ignore `Null` values in aggregation functions. We also allow the replacement of `Null` with a suitable value using the `Is-Null(E.attr, value)` function when necessary. This is similar to the `Is-Null` function of standard SQL. Thus the expression `Is-Null(CarEvent.speed, 20)` would replace any `Null` car speed values with `20`.

The detection phase can have a significant effect on the efficiency of a complex event's projection. For example, if we wish to maintain a running average over a window of events, then incrementally computing the average is generally more efficient. By analysing expressions during creation of a detector it is possible to determine when such optimizations are worthwhile. Currently we only perform this optimization for relatively simple cases (e.g. when no condition exists, all operands have unbounded windows, and events are not consumed). Although it is

[1] Note the choice of `Set` over `Bag` is irrelevant here since the aggregation function is `Avg`

[2] A `Null` value may also be returned if an operand of the `Or` operator is referenced.

transparent to the user, state is maintained as part of the projection's aggregation function and only new events are transferred to the projection phase.

Finally, we note that the `Select` clause is evaluated separately for each active detection context. Thus aggregation functions may only be applied to the output of a single context instance. If keys are defined, then the `Select` clause is applied separately to every context instance contained within each partition.

3.4 Implementation

Having introduced the various features of our language in the previous section, we now describe the algorithms and data structures used by our implementation. The pseudo-code description of our detection algorithm also serves as an operational semantics for the language. We use several conventions in our pseudo-code. Firstly, indentation is used to denote the beginning and end of block scopes. Secondly, all function calls are italicized. Thirdly, all function names that end in a question mark return a boolean value. Finally, loop constructs based on the universal quantifier ($\forall$) indicate that evaluation is order insensitive. Procedural loop constructs (i.e. `foreach` and `while`) indicate that iteration order is important.

3.4.1 Data Structures

We introduce firstly the data structures used by our implementation. A complex event data structure *ce-data* $\in$ *CE-DATA* holds information about the schema of a complex event, in addition to the dynamic information used to perform detection. A *ce-data* is a tuple:

$$(name, ctxt, ep, map\text{-}expr, gkeys, lkeys, gps) \tag{3.1}$$

The first element of a *ce-data* tuple, *name*, is the type name of the complex event, as specified in the `CreateCE` clause of its definition. Thus for all complex event data structures *ce-data* $\in$ *CE-DATA* there exists a unique complex event type $type \in CE \mid type.name = ce\text{-}data.name$. Conversely, for all complex event types $type \in CE$, there exists a unique complex event data structure $ce\text{-}data \in CE\text{-}DATA \mid ce\text{-}data.name = type.name$.

The schema for the detection context of *ce-data* is contained in *ctxt*. A *ctxt* is a tuple $(inits, terms)$, where *inits* is a sequence of initiator definitions and *terms* is a sequence of terminator definitions. The position of each initiator (terminator) definition in the sequence corresponds to its position in the `Init` (`Term`) clause of the complex event definition. Initiators and terminators correspond to the definitions given earlier in Section 3.3.3.

The *ep* element of a *ce-data* is the event pattern specified by the `Event` and `Where` clauses. The formal description of an event pattern definition was given in Section 3.3.4. The *map-expr* element of a *ce-data* data structure defines its mapping expression, as we discussed in Section

3.3.5. The *gkeys* and *lkeys* elements of a *ce-data* contain the lists of global and local keys defined for it, as we discussed in Section 3.3.2.

All elements of *ce-data* we have described so far relate to the definition or *schema* of a complex event. In contrast, the final element of a *ce-data* tuple, *gps*, maintains the run-time information used to perform event detection. The element *gps* is a list of *global partitions*. Each global partition *gp* is a tuple:

$$(values, lps)$$

The values that match this global partition are stored in *values*, with one value range per global key. If a global *key* definition has an empty *ranges* element, then its corresponding value element in *values* will always contain a single value. Only event instances with the same values for their keyed attributes match the partition. The set of local partitions contained in the global partition is stored in *lps*. Each local partition *lp* is a tuple:

$$(values, ctxts)$$

The values element of a local partition is similar to the values element of a global partition. The *ctxts* element is a list of *context instances*. A context instance is an important data structure in our detection algorithm. A local partition may contain multiple context instances if one of its initiators has an *icc* parameter of `add`. A context instance *ctxt-inst* is a tuple:

$$(init\text{-}events, term\text{-}events, buf\text{-}events, ep\text{-}inst)$$

The *init-events* element of a context instance contains the set of events that started the context by matching one of the initiators defined in *ce-data.ctxt*. Obviously, if this set contains multiple events they must all have a concurrent end timestamp.

The *term-events* element of *ctxt-inst* contains the set of (concurrent) events that terminate the context. Obviously, this set is only useful when the detection mode of *ctxt-inst* is `deferred` or `delayed`. In `immediate` mode, the set is empty. The element *buf-events* of *ctxt-inst* is only used when detection mode is `delayed`. It stores all the events that are detected during the lifetime of a context instance. These are then output by the detection algorithm when the context instance is terminated.

The final element of a *ctxt-inst* is *ep-inst*. An *ep-inst* is the event pattern instance of a context instance. An event pattern instance essentially mirrors the structure of the event pattern definition for *ce-data*. It stores a set of operand instances. An operand instance *operand-inst* may be a leaf operand instance *leaf-inst* or an internal operator instance *internal-operator-inst*. A *leaf-inst* is a list of event instances. Each event in *leaf-inst* has an event type equal to *leaf-def.type* and a set of attributes that match *leaf-def.threshold*, where *leaf-def* is the corresponding leaf operand definition stored in *ce-data.ep*. The events are ordered by their end timestamps. Events with concurrent end timestamps are ordered based on their start timestamps. Events with concurrent start and end timestamps are ordered deterministically but arbitrarily.

An internal operator instance also mirrors its definition. It consists of a sequence of internal global partitions *sub-gps*, where each *sub-gp* in *sub-gps* is a tuple of the form:

$$(values, sub\text{-}lps)$$

Note that in contrast to *gps*, *sub-gps* is a sequence, since the order in which internal partitions are evaluated may affect the output of complex event detection. All internal partitions are ordered deterministically based on the time at which they were created. The *values* attribute of a *sub-gp* has the same form as for a top level global partition *gp*. Each internal local partition *sub-lp* in the sequence of internal local partitions *sub-lps* is a tuple of the form:

$$(values, sub\text{-}ctxts)$$

Finally, each *sub-ctxt-inst* in the sequence *sub-ctxts* is a tuple of the form:

$$(init\text{-}events, term\text{-}events, ep\text{-}inst)$$

A *sub-ctxt-inst* is thus almost identical to a *ctxt-inst*. However, since sub-contexts can not have a delayed detection mode, there is no need for a *sub-ctxt-inst* to have a *buf-events* element.

3.4.2 Detection Algorithm

The locations of detectors for each complex event are transparent to users. To simplify the description of our detection algorithm, we assume that detectors are hosted on a single node. At a logical level, extending the algorithm to cope with detectors distributed across multiple nodes does not affect the algorithm. In practice, distribution introduces several difficulties. We discuss these further in later chapters.

At any moment in time, a set of complex event data structures *CE-DATA* exist in our system. From these, a function $get\text{-}ce\text{-}graph \in \mathbb{P}\, CE\text{-}DATA \mapsto (E,\ E\ X\ E)$ generates a directed acyclic graph *ce-graph* of the currently defined event types. The graph *ce-graph* is a tuple $(types, edges)$, such that $ce\text{-}graph.types \subseteq E$, and $ce\text{-}graph.edges \subset E\ X\ E$. An edge *edge* exists in *ce-graph.edges* if its source vertex is a direct input type of its sink vertex. The function $get\text{-}direct\text{-}input\text{-}events \in (E,\ E\ X\ E) \mapsto \mathbb{P}\, E$ returns the direct input event types of an event type according to a set of edges.

Our detection algorithm divides the event types in *ce-graph.types* into several *strata*. The stratum of an event type is a non-negative integer determined by the strata of its input event types. A primitive event type *pe* such that $pe \in PE \subset E$, where PE is the set of all primitive event types, has stratum zero. The stratum of a complex event type is one greater than the maximum stratum of all its input event types. Thus all input event types to a complex event type must come from lower strata, and at least one input event type must come from the stratum directly below it. More formally the function $stratum \in (E \mapsto N_0)$ is defined as:

$\forall e \in$ *ce-graph.types*

$$stratum(e) = \begin{cases} 0 & \text{iff } e \in PE \\ (max\text{-}stratum(get\text{-}direct\text{-}input\text{-}events(e, ce\text{-}graph.edges)) + 1) & \text{otherwise.} \end{cases}$$

Here the function *max-stratum* $\in (\mathbb{P}\, E \mapsto N_0)$ returns the maximum stratum of a set of event types.

The detection algorithm is invoked when new events become stable. We refer to a time instant t at which new events become stable as a stage. At each stage t, the detection algorithm processes the the strata in *ce-graph* in a bottom-up fashion. The events detected by lower level strata become part of the input event sets for higher level strata. Thus complex events at stratum 1 are detected at t based on the events of stratum 0 (i.e. the primitive events). Complex events at stratum 2 are detected at t using the primitive events occurring at t and the events generated by stratum 1. Detection for t completes when there are no more complex event strata to process. Event processing for each stage t is handled by a function ***process-stage*** $\in ((\mathbb{P}\, CE\text{-}DATA, \mathbb{P}\, PE\text{-}INST) \mapsto (\mathbb{P}\, CE\text{-}DATA, \mathbb{P}\, E\text{-}INST))$. Pseudo-code for this function is given below:

```
1 process-stage(ce-datas, primitive-events)
2   stage-events := primitive-events
3   updated-ce-datas := ∅
4
5   s := get-stratum-ce-datas(1, ce-datas)   /* s ⊆ ce-datas */
6
7   while (s ≠ ∅)
8      /* stratum-result = (updated-s, stratum-stage-events) |
9            updated-s ∈ ℙ CE-DATA ∧
10            stratum-stage-events ∈ ℙ CE-INST */
11
12      stratum-result := process-stratum(s, stage-events)
13      stage-events := stage-events ∪ stratum-result.stratum-stage-events
14      updated-ce-datas := updated-ce-datas ∪ stratum-result.updated-s
15
16      s := get-stratum-ce-datas(stratum(s) + 1, ce-datas)
17
18   return (updated-ce-datas, stage-events)
```

The ***process-stage*** algorithm takes as input the set of complex event data structures `ce-datas` whose elements' associated types are part of *ce-graph*, and the set of primitive event instances `primitive-events` that occurred at stage t. It initializes the set of events detected for t using `primitive-events` (line 2), and creates an initially empty set `updated-ce-datas` to store updated versions of each element of `ce-datas` (line 3). It then extracts the set of complex event

data structures in the first stratum from `ce-datas` (line 5). The algorithm then iterates through the strata (lines 7–16). It calls the *process-stratum* function (see below) to perform event processing for each stratum (line 12), and stores the result in a tuple `stratum-result`. The first element of `stratum-result` contains updated versions of each complex event data structure in the stratum, and the second element contains the set of complex event instances generated for the stratum. It adds the event instances detected for complex event types in the stratum to the events detected for lower strata (line 13), and also stores the updated versions of each complex event data structure in the stratum (line 14). It uses the accumulated event instances to perform detection for the next stratum. When there are no more strata to process, detection for stage t terminates, and a tuple containing the updated complex event data structures and the set of events generated for the stage is returned (line 18).

The detection algorithm performs event processing for a stratum using the *process-stratum* function, where *process-stratum* $\in ((\mathbb{P}\ \textit{CE-DATA}, \mathbb{P}\ \textit{E-INST}) \mapsto (\mathbb{P}\ \textit{CE-DATA}, \mathbb{P}\ \textit{CE-INST}))$. We present a pseudo-code description of *process-stratum* below:

```
1 process-stratum(s, in-events)
2   new-events := ∅
3
4   /* GPE ⊆ in-events, LPE ⊆ GPE (see text) */
5   ∀ ce-data ∈ s, ∀ GPE ∈ partition(ce-data.gkeys, in-events)
6      if (∃ gp ∈ ce-data.gps | matching-partition?(gp, GPE))
7         ∀ LPE ∈ partition(ce-data.lkeys, GPE)
8            if (∃ lp ∈ gp.lps | matching-partition?(lp, LPE))
9
10              foreach (term :  ce-data.ctxt.terms)
11                 terminate(LPE, term, lp.ctxts)
12
13                 ∀ ctxt-inst ∈ lp.ctxts | terminated?(ctxt-inst)
14                    if (ce-data.ep.detect-mode = delayed)
15                       new-events := new-events ∪ ctxt-inst.buf-events)
16                    else if (ce-data.ep.detect-mode = deferred)
17                       terminate-sub-ctxts(ctxt-inst, LPE, ce-data.ep)
18                       new-events := new-events ∪
19                             detect-consume-project(ctxt-inst, ce-data.ep)
20                    delete(ctxt-inst)
21
22              ∀ ctxt-inst ∈ lp.ctxts
23                 terminate-sub-ctxts(ctxt-inst, LPE, ce-data.ep)
24                 collect(ctxt-inst, LPE, ce-data.ep)
25                 if (new-stage?(ctxt-inst))
26                    new-events := new-events ∪
```

```
27                          detect-consume-project(ctxt-inst, ce-data.ep)
28
29              foreach (init :  ce-data.ctxt.inits)
30                 lp.ctxts := lp.ctxts ∪ initiate(LPE, init, lp.ctxts)
31              ∀ ctxt-inst ∈ lp.ctxts
32                 initiate-sub-ctxts(ctxt-inst, LPE, ce-data.ep)
33
34           else
35              lp := create-local-partition(ce-data.lkeys, LPE)
36              foreach (init :  ce-data.ctxt.inits)
37                 lp.ctxts := lp.ctxts ∪ initiate(LPE, init, lp.ctxts)
38              ∀ ctxt-inst ∈ lp.ctxts
39                 initiate-sub-ctxts(ctxt-inst, LPE, ce-data.ep)
40
41     else
42        gp := create-global-partition(ce-data.gkeys, GPE)
43        ∀ LPE ∈ partition(ce-data.lkeys, GPE)
44           lp := create-local-partition(ce-data.lkeys, LPE)
45           foreach (init :  ce-data.ctxt.inits)
46              lp.ctxts := lp.ctxts ∪ initiate(LPE, init, lp.ctxts)
47           ∀ ctxt-inst ∈ lp.ctxts
48              initiate-sub-ctxts(ctxt-inst, LPE, ce-data.ep)
49
50  return new-events
```

The algorithm processes each `ce-data` in `s` independently, as indicated by our use of the universal quantifier (line 5). The algorithm partitions the input events based on the global keys of `ce-data` (line 5). If no global keys exist, events are detected within a single universal partition. The resulting sets of globally partitioned event instances (GPEs) each match a different global partition. Thus each set of event instances GPE $\in \mathbb{P}$ E-INST. It firstly handles the case where a partition matching the GPE already exists (lines 6–39). For all global partitions, it uses the local keys of `ce-data` to further partition the GPE into sets of locally partitioned event instances (LPEs) (line 7). Thus each LPE $\in \mathbb{P}$ E-INST is a subset of exactly one GPE.

The next part of our algorithm handles the case where a local partition already exists for an LPE (lines 8–32). Within each local partition, there may be one or more existing `ctxt-inst`. The detection algorithm firstly attempts to terminate these `ctxt-inst` using the LPE (lines 10–23). Terminators specified in the `Term` clause of the complex event expression are evaluated from left to right (line 10). Note we use the `foreach` keyword instead of ∀ to indicate that iteration order is important. Every event of the LPE is evaluated with respect to a terminator. A `ctxt-inst` may be terminated by multiple events, but by only a single terminator.

The next step of our algorithm handles all `ctxt-inst` that have been marked as terminated

by the *terminate* function (lines 13–20). A terminated `ctxt-inst` with an `immediate` detection mode is simply deleted (line 20). For other detection modes, our algorithm must perform some additional work before deleting the `ctxt-inst`. If its detection mode is `delayed`, then a `ctxt-inst` may contain events that were detected during the period it was active. We store any such events in a buffer attribute `buf-events` of the `ctxt-inst`. We timestamp the `buf-events` with the current stage time before adding them to the set of detected events (line 15).

A `ctxt-inst` with a `deferred` detection mode requires us to perform event detection when it is terminated (lines 18–19). Any events generated are added to the set of detected events. Note that before the detection algorithm is invoked, contexts of internal operators, (i.e. sub-contexts), must be checked for termination (line 17). As we mentioned in Section 3.3.4.2, a `deferred` context may not be nested within a context that is `deferred` or `delayed`. In addition, a `delayed` context may not be nested within any other context. Thus all sub-contexts of a `deferred` context must have an `immediate` detection mode. Terminated `immediate` sub-contexts are deleted by the *terminate-sub-ctxts* function, and do not contribute to event detection. Termination of the set of `ctxt-inst` continues until there are no more terminators left.

At this point (line 22), the termination phase is almost complete. However, before proceeding to the event detection phase we must check whether any sub-contexts of the remaining `ctxt-inst` need to be terminated (line 23). If there are no sub-contexts with a `deferred` detection mode, then all terminated sub-contexts can simply be deleted. Terminated `deferred` subcontexts (and all their child contexts), must be retained until the end of the stage, since they may generate new events (in lines 26–27).

Having completed the termination phase, the algorithm must now check whether event detection needs to be performed for all remaining `ctxt-inst`. The first step in this process is the collection phase (line 24). In this phase the algorithm adds events to all operands with matching event type and threshold conditions. If any events are added, then a new stage occurs for the `ctxt-inst`. A new stage also occurs if a deferred sub-context is terminated (in line 23).

If a new stage occurs, we use the *detect-consume-project* function to perform complex event detection (lines 26–27). If specified by the consume flag of their operands, this function also performs consumption of the constituent events of any detected events. In addition it deletes any terminated `deferred` subcontexts (from line 23). The projection phase maps any events generated to new complex events. We then add these complex events to the events accumulated for the stratum.

The last major task we need to perform is initiation. Similarly to termination, initiators specified in the `Init` clause of the complex event expression are evaluated sequentially from left to right (line 29). Each initiator that is matched may result in the creation of a new `ctxt-inst` (depending on the *icc* parameter of the initiator and whether a `ctxt-inst` already exists). If multiple events match an initiator, at most one new `ctxt-inst` is created. A newly created `ctxt-inst` is thus initiated by a set of concurrent events. However, if multiple initiators are matched, more than one `ctxt-inst` may be created (if the *icc* parameter of one of them is `add`). A single event that matches more than one initiator may thus initiate multiple `ctxt-inst`.

For all `ctxt-inst` (whether newly initiated or not), we must determine if any of their sub-contexts are initiated by the events in LPE. Initiation of sub-contexts is performed recursively by the *initiate-sub-ctxts* function (line 32).

When no local partition exists for an LPE, a new one must be created (lines 34–39). Since it does not contain any `ctxt-inst` yet, we only have to perform initiation within this partition. Similarly, if no global partition exists for a GPE, a new one must be created (line 42). Within this new global partition, new local partitions must be created for all LPEs (lines 43–44). Once again, within newly created partitions we only need to perform initiation (lines 44–48).

Finally, when detection has been performed for all complex events in a stratum, the algorithm returns the newly generated events (line 50).

3.5 Related Work

There are a variety of languages related to our complex event detection language in the literature. The reader can find a summary of their distinguishing features in Table 3.1. We give an overview here of the main categories.

3.5.1 Composite Event Languages

Composite or Complex event languages arose from work done by the Active Database [PD99] community. They were originally designed to support centralized systems, although subsequent work has attempted to address distribution issues such as the lack of a global clock [Sch96; YC99; PSB04]. We now describe several examples of these languages from the literature.

Ode [GJS92; GJ92] is a regular-expression like language from Gehani et al. where composite events are detected using finite state automata. It supports a variety of composition operators, and also allows correlation across operators using the information contained in event attributes. However it does not allow for extensive control of consumption parameters, and the time model is unsuitable for distributed systems.

The Samos language of Gatziu and Dittrich [GD93; GD94] uses Petri Nets to perform composite event detection. Petri Nets are more powerful than Finite State Automata as they allow for concurrent processing and management of complex data such as event parameters during detection. However the time model of SAMOS does not allow for simultaneous events and is not suitable for distributed systems. In addition SAMOS does not allow fine-grained control over event selection and consumption.

Snoop [CM94; CKAK94] is an expressive CE detection language from Chakravarthy et al. with extensive temporal support. Detection is based on a tree corresponding to the structure of the event expression. Snoop introduced the notion of parameter contexts, also known as consumption policies, to control the constituent events of a composite event in the case of ambiguity. However, consumption policies can only be specified globally, instead of for individual operands. Snoop does not allow the specification of conditions taking into account the attributes of events,

and was not designed with concurrency or distributed systems in mind.

The Event Pattern Language (EPL) of Motakis and Zaniolo [MZ95b; MZ97a] is a composite event language with a formal semantics defined using Datalog_{1S}. Expressions are scoped by modules, and conditions over event attributes, concurrent processing, and simultaneous events are also supported (although the time model does not allow duplicate events). The parameter contexts of Snoop are also defined formally, but sophisticated control over windowing, selection and consumption is not provided. The language is extended in [MZ97b] with extensive support for aggregation operations.

An excellent overview of the commonalities and differences between composite event languages for Active Database systems is given by Zimmer et al [ZMU97; ZU99]. They divide the semantics of composite events into three independent dimensions, broadly corresponding to our detection stages of collection, selection, and consumption. They define a formal meta model for composite event algebras based on their analysis. However, they do not discuss how event attributes can be used in conditions during detection. Their work is focused on centralized systems with a global clock, and is not directly applicable to distributed systems.

A rule based event monitoring language is specified by Mansouri-Samani et al. as part of the GEM system [MSS97]. It uses a tree based approach for selecting events, and allows conditions to be specified over the attributes of events. Detection is performed using the chronicle consumption policy of Snoop, but more sophisticated control is not provided. The language allows rules to be annotated with discard timeouts in order to deal with delayed events in distributed systems. This may not be feasible in an environment with unpredictable delays. In addition the time model assumes a synchronized global clock.

The Amit language was designed by Adi and Etzion for the specification and detection of complex events, which they refer to as *situations* [AE04; AE02]. A situation is defined by a single operator, with input events being stored in a candidate list for each operand of the operator. In addition, each situation has an associated lifespan, which acts as a context for its detection. Our language bears many similarities to Amit, especially with respect to its processing model and operators. However, Amit does not allow nested operators within a single complex event. It also provides different control parameters for operands, and does not support the specification of ranges in keys. Their time model also assumes a globally synchronised clock.

Cayuga is an automata-based event processing language from Cornell [DGP+07]. It provides operators for sequencing different event streams, and aggregating events of the same type. It uses the concept of epochs, which are similar to our idea of stages, to handle concurrent events. In terms of distribution, it relies on a system-wide time delay to handle out of order events, but does not address issues arising from the lack of a globally synchronised clock.

Sase is an event language [WDR06] from Wu et al. targeted towards RFID applications. It uses a query-plan based approach to detection. Sase natively implements operators for efficiency, and presents several query plan optimizations that can help to further improve performance. However, the language does not provide extensive control over collection and selection and does not support static partitions. Furthermore, it assumes a totally ordered event stream, and does not address distributed system issues. Sase+ is an extension of Sase with support for a Kleene

closure operator, which enables a variety of aggregation operations over a single event stream (some of which we do not currently support) [ADGI08].

3.5.2 Continuous Query Languages

Continuous Query and Stream processing languages are another area of research that has influenced our work. In contrast to composite event languages, most continuous query and stream processing languages only delete events using windows (either temporal or tuple based), and do not consume events (see Section 3.3.4.3). In addition, all of the following languages rely on a globally synchronised clock.

Arasu, Babu, and Widom [ABW03] provide a precise semantics for continuous query languages as well as an instantiation of the semantics called CQL. Their language provides three classes of operator—stream to relation, relation to relation, and relation to stream. Stream to stream operators can be built by composing these operators together. This enables their semantics to exploit well understood relational semantics. They suggest the use of a heartbeat mechanism to deal with delayed events in distributed systems, but rely on a global clock for timestamping.

Aurora is data stream management system for monitoring applications [CcC+02; ACc+03] from Carney et al. Queries in Aurora are specified directly as an operator graph. Aurora also allows the specification of *QoS graphs* to control how events should be shed under conditions of high load, and *slack parameters* to help deal with out of order input events.

The TelegraphCQ language of Chandrasekaran [CCD+03] allows flexible specification of windows using a procedural for-loop construct. This enables the definition of landmark, sliding, tumbling, and hopping windows. Windows can also move backwards. However event bounded windows like those provided by our detection contexts are not supported.

ESL [BTW+06] is an event stream language from Bai et al. that emphasizes the importance of compatibility with SQL syntax and semantics for applications that span both DB tables and data streams. It also provides extensive support for and optimization of user defined aggregates over various types of window.

Finally, punctuations [TMSF03] were introduced by Tucker et al. to enable windows over streams to be defined more flexibly. In addition to stream tuples, publishers can generate explicit punctuations with regard to input streams, indicating that a certain condition will hold for the remaining elements of the stream. This allows query operators that must block until the whole stream is seen to be supported. Punctuations can be modeled in our language as explicit punctuation events.

Some other relevant continuous query and stream processing languages include Gigascope, Tapestry, Chronicle, and Stream [CJSS03; TGNO92; JMS95; MWA+02]. However, we refer the reader to the original papers for further details, since they do not provide any additional language features of particular interest to us.

3.5.3 Hybrid Languages

Several researchers have attempted to synthesize ideas from composite event and continuous query languages to create more powerful hybrid languages.

The EStreams language of Jiang, Adaikkalavan and Chakravarthy is one such effort [JAC04]. They introduce the concept of *stream modifiers* in order to generate events from streams. These events are then combined using operators based on those of Snoop. In addition, they propose a novel windowing mechanism for streams called a *semantic* window. Termination of a semantic window is controlled by a condition over newly arriving stream tuples and tuples already contained in the window. Our override parameters could potentially be extended to support a semantic window condition. However, most of the functionality provided by such a condition can be achieved using detection contexts, which in addition to incremental output allows event detection to be deferred until the end of the window.

The complex event language of Rizvi [Riz05] is another attempt to combine continuous queries with composite events. The continuous query language used is TelegraphCQ, whereas the event language is again based on Snoop. The language uses a different notion of semantic window whereby events are used to define window boundaries. This enables, for example, windows that extend back in time based on a newly arriving event.

The ESL stream processing language was extended with event handling capabilities by Bai et al, resulting in the hybrid ESL-Events language [BWL+07]. Although designed for centralized scenarios where integration with static relations is required, their event language contains several novel features. Firstly, they provide a *starred sequence* (`seq*`) operator that enables the specification of conditions between events *in the same operand*. Our language can express the example use cases given for this operator, although in a less direct fashion. Similarly, the *exception sequence* (`exception-seq`) operator can be expressed, but in a much less succinct fashion. Finally, they mention their support for active expiration semantics, and give several examples to emphasize its importance. As mentioned in Section 3.3.4.2, we intend to extend our language to support this as future work. In terms of selection and consumption, their language relies on *tuple pairing modes*, which are similar to the consumption policies of Snoop.

The Esper event stream processing language combines ideas from continuous query languages with support for detection of event patterns [ESP09]. Esper provides a selection of operators for defining event patterns similar to those of the RAPIDE composite event language [RAP09]. Roughly speaking, expressions containing event patterns consume events, whereas expressions based on continuous query joins tend to employ sliding windows. Esper requires multiple expressions to capture the functionality provided by our static detection partitions, and does not provide as much control over collection, selection and consumption. Moreover, Esper does not address distributed systems issues arising from the lack of a global clock.

3.5.4 Durative Event Languages

Durative event languages are concerned with the events that occur over a period of time, instead of instantaneously. For example, complex events generated by the expressions of our language

can be viewed as occurring over a time bounded by the earliest start time of any constituent event and the time at which they were detected.

The SnoopIB [AC03] language of Adaikkalavan and Chakravarthy is an extension of the Snoop composite event language detects events using an *interval based* semantics. They formalize the detection semantics for each Snoop operator using event histories. Apart from the drawbacks discussed with respect to Snoop, expressions are detected in general using only a partial event history. We provide an `sst` function to give more information about the state of input events. However, care must be taken to prevent detection failures in case of expressions where the `sst` value never advances.

Yoneki and Bacon have defined a formal semantics and language for the detection of composite events in distributed WSNs [YB05]. In contrast to our operand level parameters for controlling selection and consumption, they advocate the use of a subset restriction policy, that acts over a whole operator. A similar policy was proposed by Carlson and Lisper [CL04]). However the policy assumes that at most one event should be produced at each time instant. As is the case with SnoopIB, detection of durative events is performed online without the help of an `sst` function.

3.5.5 Production System Languages

Production system languages are also relevant to our work. Most of these are based on first order logic, and thus are more expressive than our language, since it is possible to have recursive rules. In addition they typically require a closed world assumption, and operate over a single centralized database. In contrast the graph representing all complex events in our language is required to be acyclic, and expressions can be decomposed and distributed.

The seminal work in the area is the Rete algorithm of Forgy [For82]. Rete is a fast matching algorithm that was created to detect patterns defined by rules in the OPS5 production system language [BFKM85]. Rule processing is performed over a *working memory*, which contains facts representing knowledge about the current state of the world. Rules are compiled into a *Rete network*, composed of α and β nodes. These nodes store tokens representing partially matched rules in order to speed up evaluation. Changes in the knowledge stored in working memory cause updates to the tokens stored in these nodes. If multiple updates are generated as a result of a change to working memory, they are stored in a *conflict set*. A *conflict resolution strategy* defines the order in which these updates should be applied.

Treat [Mir87; ML91] is an alternative matching algorithm for production systems proposed by Miranker. Treat is similar to Rete in functionality, except that it uses less state at the cost of increased execution times in some cases. It does not cache intermediate results, instead only storing the working memory and the conflict set.

The Rete* algorithm of Wright and Marshall [WM03] is a more flexible matching algorithm than either Rete or Treat. It allows applications to parametrise matching with the maximum amount of memory to be used for caching intermediate results.

Leaps is yet another matching algorithm suggested by Miranker et al [MB90; Bat94]. Leaps

is an extension of Treat that does not enumerate the whole conflict set, but instead fires the first activated rule. This enables it to handle large databases, as its lazy evaluation model means many potentially fireable rules never need to be triggered.

The Gator [HH93] algorithm of Hanson and Hasan is targeted towards condition processing for higher data rate active databases. It uses discrimination networks that are essentially generalized trees, in contrast to the binary trees of Rete.

Finally, Argus is a stream monitoring system from Jin et al. that focuses on applications where highly selective rules are the norm (stream anomalies) [JCH05]. Detection is performed using a restricted class of Rete network that limits the complexity of the queries that can be posed (in comparison to production systems). Argus employs a sliding window style event discard policy, similar to that used by continuous query languages. The fact that application queries are very selective means that the amount of memory required to store intermediate results remains low, one of the main problems for large scale production systems.

3.5.6 State Detection Languages

State detection languages are another class of language relevant to our work. These allow the association of two or more complex events in order to enable the efficient detection of the current state of some entity. In theory, a complex event language could be seen as complementary to a (sufficiently flexible) state detection language, since a complex event could be mimicked by a state whose deactivation fires immediately after its activation.

Roemer and Mattern argue for the use of states instead of composite events, since they allow more natural problem modeling for some applications [RM04a]. They present a language that provides constructors for specifying binary states. Temporal and spatial operators allow the detection of more complex combinations of states. Consumption of events is controlled by procedural code specified as part of the action of the rule in which a state specification is defined. An extensive discussion of issues pertaining to distributed detection in wireless sensor networks is also provided. In contrast, we attempt to provide more declarative control over event selection and consumption, and our operators and support for detection contexts are more suited to applications interested only in events.

Another state detection language is that of Taherian and Bacon [TB07]. Similarly to Roemer, it allows the specification of binary states and conditions over entrance and exit events. In contrast to our language, event consumption is not supported. Instead, events can only be discarded using windows (similar to continuous query languages). However, this restriction does enable an interesting shared state model between query expressions.

Finally, Rizvi [Riz05] motivates an extended complex event detection model where events are used as inputs to a state transition diagram. Events can occur probabilistically, and the state transition diagram gives a probability distribution over the possible current states. This model essentially generalizes the binary states of Romer and Taherian. A description of Rizvi's complex event detection language was given earlier in our discussion of hybrid languages.

3.6 Summary

We have defined a language for describing complex events. The language was designed with the open distributed environment of pervasive computing applications in mind.

In general, due to the lack of a global clock in distributed systems, events cannot be totally ordered. We assume an uncertainty interval representation for event timestamps. Events with disjoint timestamps can be detected normally. Events whose uncertainty interval timestamps conflict are grouped into a stage. Detection of a stage with multiple conflicting events is application-specific.

Our language also simplifies distributed detector placement by enabling the specification of easily decomposable and partitionable expressions directly. Events requiring more sophisticated composition (such as conditions across multiple operators) are also supported. These are typically more difficult to distribute, due to the dependencies between operands of different operators.

Finally, many of our language constructs can be configured extensively. This is important as it increases the generality of our language. Pervasive computing applications may exhibit a wide variety of detection semantics, as shown by the variety of examples used throughout the chapter.

	C	XOC	UC	TIW	TUW	EW	CE	P	RP	DE	AG	SEL	CONS	DT	DBI	ST	NO	CY
O'Keeffe	✓	✓	~	✓	✓	✓	✓	✓	✓	✓	~	✓	✓	✓	✗	✗	✓	✗
CELs																		
Ode	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗
Samos	~	~	~	✓	✗	~	✗	✗	✗	✗	✗	~	✗	✗	✗	✗	✗	✗
Snoop	✗	✗	✗	✓	✗	✓	✗	✗	✗	✗	~	~	~	✗	~	✗	✓	✗
EPL	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✓	~	~	✗	✓	✗	✓	~
Zimmer	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✗	✓	✓	✗	~	✗	✓	✗
GEM	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	~	✗	✗	✗	✗	✗
Amit	✓	✗	✗	✓	✗	✓	✗	✓	✗	✗	~	✓	✓	✗	✗	✗	✓	✗
Cayuga	✓	✓	✓	✓	✓	~	✓	✗	✗	~	✓	✗	✓	✗	✗	✗	✗	✗
Sase	✓	✓	✓	✓	✗	✓	✗	✓	✗	✗	✓	✗	~	✗	✗	✗	✓	✗
CQLs																		
CQL	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓	~	✗	✗	✓	✗	✓	✗
Aurora	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓	~	✗	✗	✓	✗	✓	✗
TeleCQ	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓	~	✗	✗	✓	✗	✓	✗
ESL	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✓	~	✗	✗	✓	✗	✗	✗
HLs																		
EStreams	✓	✗	✓	✓	✓	~	✗	✓	✗	✗	~	~	~	✗	✗	✗	~	✗
Rizvi	✓	✓	✓	✓	~	~	✗	✓	✗	✗	~	~	~	✗	~	~	✗	✗
ESL-EV	✓	✓	✓	✓	✓	✓	✗	✓	✗	✗	✓	✓	~	✗	✓	✗	✓	✗
Esper	✓	✓	✓	✓	✓	✓	~	~	✗	✗	✓	✓	✓	✗	✓	✗	~	✓
DELs																		
SnoopIB	✗	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	~	~	~	✗	✗	✓	✗
Yoneki	✗	✗	✗	✓	~	✓	✓	✗	✗	✓	✓	~	~	✓	✗	✗	✗	✗
SDLs																		
Roemer	✓	✓	✓	~	✗	✓	~	✓	~	✗	✗	~	~	✓	✗	✓	✓	~
Taherian	✓	✓	✓	✓	✓	✓	✓	✓	~	✗	✓	✓	✗	✗	✗	✓	✓	~
PSLs																		
Typical	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓

Table 3.1: Language Comparison: ✓ = Good support, ~ = Partial support, ✗ = No support; Feature acronyms given in Table 3.2

C	Conditions
XOC	Cross-Operator Conditions
UC	Unary Conditions
TIW	Time Windows
TUW	Tuple Windows
EW	Event Windows
CE	Concurrent Events
P	Partitions
RP	Region Partitions
DE	Durative Events
AG	Aggregation
SEL	Selection
CONS	Consumption
DT	Distributed Time
DBI	Database Integration
ST	States
NO	N-Ary Operators
CY	Cycles

Table 3.2: Feature Key

Chapter 4

Reliable Complex Event Detection

4.1 Introduction

Complex event detection is an inherently failure prone endeavour for pervasive computing applications. Detection failures can occur for a variety of reasons, including imprecise time synchronization, lossy communication, and out of order event delivery. Many of these errors must be handled in an application specific manner. Providing applications with a simple way to control how errors affect complex event detection is therefore vital if we are to make pervasive computing practical.

Most existing work on distributed complex event detection takes a relatively simplified view of these problems; they either ignore errors and perform best effort detection, wait indefinitely for errors to resolve themselves, or fail. More sophisticated solutions have been suggested, such as approximate continuous query processing [MWA+02; CcC+02] and probabilistic event predicates [Bau04]. However these approaches either rely on an overly simplified time model, miss some opportunities to improve event detection accuracy, or do not discuss complications that arise for expressions that consume events.

In this chapter, we focus on improving complex event detector handling of several types of communication error, as well as timing errors caused by the lack of a global clock in distributed systems. Our contributions are threefold.

- We define several levels of correctness for complex event detection, and discuss how different types of error can affect the correctness of a detector's output event stream (Section 4.2).
- We propose the use of detection policies to enable applications to control the handling of errors (Section 4.4). We discuss several such policies, and under what conditions each is useful. We then show how the complex event detection language introduced in Chapter 3 can be extended to enable the specification of our detection policies.
- Finally, we describe in detail the implementation of a detection policy to support *no false positives* complex event detection (Sections 4.5 and 4.6). Performance evaluations of our

implementation show significant improvements in detection accuracy in the presence of various system failures. In addition, our evaluations show the extension imposes only a small overhead under failure free operating conditions.

4.2 Background

4.2.1 Communication Errors

Most current attempts to guarantee the correctness of complex event detection require input events to be delivered in a totally ordered fashion, based on the time at which they occurred. However, if event delivery between publishers and detectors is not reliable, this may be impossible to ensure. For example, if publishers are simple devices that do not have much memory (e.g. sensors), it may be unreasonable to require them to store persistently all the events they detect in order to ensure reliable delivery [HGM01]. For expressions with timeliness constraints, even eventually reliable delivery is not enough to guarantee correctness.

We define a correctly *detected* output stream as the output stream a detector generates when there are no event losses, and communication and detection delays with publishers do not result in an output event being detected after some subscriber-defined timeliness constraints have passed. In addition, we define a correctly *delivered* output stream as a correctly detected output stream with zero or more events removed, such that every event in the correctly delivered output stream is delivered within some time bound of its occurrence, as defined by the subscriber.

Perfect complex event detection occurs for a subscriber when the output stream it receives is correctly delivered, and exactly equal to the correctly detected output stream. *Ordered perfect* complex event detection occurs when the delivered output stream is perfect, and events are delivered in the same order as they were detected.

The effects of unreliable delivery and timeliness requirements on our ability to perform correct complex event detection are summarized in Table 4.1. The table shows that when event delivery is guaranteed and there are no timeliness requirements defined by the subscriber we can ensure ordered perfect complex event detection. When delivery is still reliable but there are timeliness requirements, only a correctly delivered output stream can be guaranteed. When delivery is unreliable, and in the absence of a mechanism for detecting lost events, or delayed events when there are timeliness requirements, only best effort detection is possible[1]. As we will show, augmenting complex event detection with such a mechanism enables more reliable detection.

4.2.2 Time Synchronization Errors

The lack of a global clock in distributed systems introduces another class of errors. This absence can make it impossible to precisely define the exact time at which an event occurred. In order

[1] Ignoring the trivial case of a detector that intentionally produces no output

	Unreliable Delivery	Reliable Delivery
Bounded Delay	Best Effort	Correctly Delivered
Unbounded Delay	Best Effort	Ordered Perfect

Table 4.1: Possible Detection Guarantees

to cope with this problem, we assume event timestamps use an uncertainty interval to bound their imprecision [LCB99]. Events whose uncertainty intervals overlap can cause problems for current complex event detection languages, since it is not clear how they should be ordered. To model the effect of conflicting uncertainty intervals on complex event detection, we can extend our earlier definition of a correctly detected output stream to require that the output generated be equal to the output that would be generated if the ordering of events was known. As we will show, in the presence of conflicting uncertainty intervals, our ability to perform correct detection depends on the type of complex event expression being detected.

4.2.3 Motivating Example

As a motivating example for our work, consider a container packing application (e.g in a factory, airport or shipping yard). Small packages with RFID tags attached travel along a conveyor belt past an RFID reader (R1). Larger containers with their own RFID tags travel along a different conveyor belt past another RFID reader (R2). The two conveyor belts meet at a loading point (LOADING). A container waits at the loading point until at least one and at most 3 small packages have been stored inside. The container then proceeds along the conveyor belt to a holding area where it waits to be transported. The next container on the belt takes its place and the process repeats. The layout of the conveyor belts is depicted in Figure 4.1.

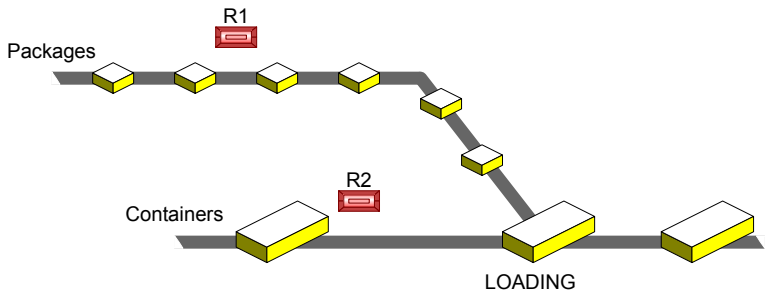

Figure 4.1: Container Packing Application

For tracking purposes, a record of which small packages are in which container must be maintained. To this end, RFID reader R1 generates a sequence of events with the schema `Package(readerId,`

pkgId). Similarly, R2 generates a sequence of events Container(readerId, contId)[1]. These events are sent over a wireless link to an event broker that hosts a complex event detector responsible for matching packages to containers. The detector generates a stream of PackageContainer events, computed using the expression:

```
CreateCE PackageContainer
Select Package.pkgId, Container.contId
Event And(Package<old max 3, consume>,
          Container<old max 1, consume>)
```

A tracking database subscribes to the detector output, and stores the various assignments as they are received. Note that although for our packing example there is only a single source of events for each type, in general there may be multiple sources.

This setup will function correctly if event delivery is totally ordered and timely. However, RFID readings are transmitted wirelessly and may be lost en route to the detector. In addition, the readers may be memory constrained, and persistent storage of events to ensure reliable delivery may impose an unwanted overhead. Such a scenario can result in gaps in the input streams received by the PackageContainer detector.

If the events in these gaps are lost forever, then a detector with a guaranteed detection policy will fail. Unfortunately, using a best effort detection policy instead is not a viable alternative, since it will result in lost package assignments. In addition, blindly assigning packages to containers without taking into account gaps can cause *cascades* of bad assignments.

The only alternative for dealing with containers holding unknown packages due to lost readings is to have a supervisor scan them in manually. To reduce the workload of the supervisor, we would like to minimize the occasions on which this is necessary. A simple option would be to halt the conveyor belts and restart detection after the unknown packages have been scanned. However, this can significantly slow down the processing of packages. Ideally we could isolate the container holding the unknown packages, and let scanning of the remaining packages continue as normal. A policy such as this would help to prevent the cascade problem described above, while allowing packing to continue after an event is lost.

As an example, consider the sample input received by a PackageContainer detector in Table 4.2. The leftmost column gives the time at which events were detected by the RFID readers. The center column gives the Package readings received by the detector, and the rightmost gives the Container readings received. Note especially that a Package event (R1, 3) with (ts = t4) has been lost en route, and is therefore missing from the Package column.

In Table 4.3 we compare the actual assignments of packages to containers with some possible outputs generated by the PackageContainer detector. The leftmost column gives the timestamps of each output event. The actual packages that were assigned to each container is given in the TA (true assignment) column. If no input events had been lost, this is the output that the PackageContainer detector would have generated. The output generated by a detector

[1]Events in both of these streams also have an implicit timestamp field.

using a best effort (BE) detection policy is given in the next column. It can be seen that in addition to having no record with `pkgId = 3`, packages `6` and `9` are also assigned to the wrong container. This is an example of a lost event causing cascading errors.

Finally, the gap detection (GD) column contains the output that could have been generated if the detector was gap aware. Although there is still no record of the container in which the lost package event was stored, all output events generated are correct. In contrast to best effort detection, the cascade problem does not arise.

ts	Package	Container
1	(R1, 1)	–
2	(R1, 2)	–
3	–	(R2, 1)
4	-lost-	–
5	(R1, 4)	–
6	(R1, 5)	–
7	(R1, 6)	–
8	–	(R2, 2)
9	(R1, 7)	–
10	(R1, 8)	–
11	(R1, 9)	–
12	–	(R2, 3)

Table 4.2: Events *Received* by `PackageContainer` Detector

ts	TA	BE	GD
3	(1, 1)	(1, 1)	(1, 1)
3	(2, 1)	(2, 1)	(2, 1)
–	–	–	–
8	(3, 2)	(4, 2)	–
8	(4, 2)	(5, 2)	(4, 2)
8	(5, 2)	(6, 2)	(5, 2)
–	–	–	–
12	(6, 3)	(7, 3)	(6, 3)
12	(7, 3)	(8, 3)	(7, 3)
12	(8, 3)	(9, 3)	(8, 3)

Table 4.3: Different Possible Outputs for `PackageContainer(pkgId, contId)`, with `TA` = True Assignment, `BE` = Best Effort, `GD` = Gap Detection

In addition to errors caused by message loss and delay, detection should also handle errors that arise due to conflicting uncertainty interval timestamps. In our packaging application, two package events with overlapping uncertainty intervals might mean it is not possible to determine

which bag is stored in which container. A suitable detection algorithm would attempt to prevent errors arising from conflicting uncertainty intervals as well as from gaps in the input event stream.

4.3 Service Model

Since events may be lost or arrive out of order at a detector, they must be reordered if possible before being delivered. We refer to this process as *stabilization*. In this section, we discuss how errors in event streams are detected during stabilization.

4.3.1 Basic Service Model

We assume a model where sensors are connected to consumers via a publish/subscribe service. The publish/subscribe service acts as a *mediator* between sensors who publish *primitive events* and consumers who subscribe to events of interest (Figure 4.2). Subscriptions may contain *filters* over the events produced by publishers. These are typically stored in the mediator, although later on (Section 4.3.4) we discuss the implications of storing them at publishers. Of particular interest to us is that consumers can also create *complex* subscriptions to detect *patterns* of primitive events. These result in the creation of complex event *detectors*, which are hosted by the mediator.

Our service model assumes that the complex event detection service is integrated into the

Figure 4.2: Publish/Subscribe Service

publish/subscribe service. This differentiates a detector from a normal subscriber, since it must know about the publishers that contribute to its expression. A *spatial coupling* is thus introduced between detectors and publishers [EFGK03]. This does not necessarily invalidate the use of a publish/subscribe paradigm, since the subscribers to the output of the detector are still decoupled from publishers. However, it does mean that the complex event detection layer is more tightly integrated with the publish/subscribe layer, as in the approach of Padres [LJ05]. For some detection policies (see Section 4.4) this may not be required, and a cleaner separation as in [PSB04] may be more desirable, since the implementation of the complex event detection service is simplified.

The mediator itself may be implemented in a centralized or distributed fashion. We assume for simplicity that once the mediator receives an event from a publisher, it can store the event persistently until it has been delivered to all interested subscribers (if necessary). However, we do not require reliable delivery, as publishers may be lightweight and without the resources

to store events until they have been delivered to all interested parties. Since publishers may be connected to the mediator via lossy links (e.g. a wireless connection), events may be lost irretrievably at some point between the publisher and the mediator.

Finally, if the event generation rate of a publisher is low, periodic heartbeat messages may be sent in addition to event messages. Heartbeat messages contain a timestamp indicating the time at which they were sent, and enable the mediator to determine when the publisher is still alive, but has not yet detected any events.

4.3.2 Event Model

In a similar fashion to the type-and-attribute based filtering employed in Hermes [Pie04], we model an event as having a type and a set of attributes. Every primitive event also has a detection timestamp, which we assume for now is an uncertain point in time bounded by an uncertainty interval timestamp $[t_l, t_h]$. Each publisher advertises the type of the events it publishes. As with subscriptions, advertisements may contain filters over some or all of the attributes of the type. For example, a movement sensor may advertise the movement events it publishes as `MovementEvent(loc = FE02)`. Here the location attribute of any movement event generated by the publisher must be equal to the room `FE02`. Advertisement filters thus indicate that the publisher will only generate events with a restricted set of attribute values.

Each type t defines a *type-space* TS_t of possible events. The advertisement of a publisher p thus defines a subset of this space which we refer to as the *local advertisement-space* LAS_t^p of the publisher for the type. We define the set P_t as the set consisting of all publishers whose local advertisement-space LAS_t^p intersects TS_t (i.e. $LAS_t^p \cap TS_t \neq \emptyset$). The combination of the advertisements of all publishers of a particular type at any one time defines the *global advertisement-space* GAS_t. We can thus define the relationship, in terms of the events they contain, between a local advertisement-space, the global advertisement-space, and the type-space, as $LAS_t^p \subseteq \bigcup_{i \in P_t} LAS_t^i = GAS_t \subseteq TS_t$.

Consumer subscriptions (for primitive events) specify an event type, and optionally a filter over the attributes of that event type. Each such subscription $s \in S_t$ defines a *local subscription-space* LSS_t^s. We define the *match-space* $MS_t^{p,s}$ of a publisher p and a subscriber s as the intersection of LAS_t^p with LSS_t^s (i.e. $MS_t^{p,s} = LAS_t^p \cap LSS_t^s$). A subscription without a filter covers all of the type-space, all of the global advertisement-space, and for each publisher of the type, all of the local advertisement-space. In contrast, subscriptions with a filter cover a subset of the type-space. In addition they may cover all, some or none of the global advertisement space. Finally, each subscription with a filter may cover all, some or none of the local advertisement-space of each publisher.

For each of the three different spaces, we can define a notion of subscription equivalence. Two subscriptions are type-space equivalent (*TS-equivalent*) when they both match all events contained in the same subset of a type-space. Two subscriptions are equivalent with respect to a global advertisement space (*GAS-equivalent*) when they both match all events contained

in the same subset of the global advertisement space. Finally, two subscriptions are equivalent with respect to a local advertisement space (*LAS-equivalent*) when they both match all events contained in the same subset of the local advertisement space.

Given a particular publisher, we can divide the local subscription spaces that intersect with its local advertisement space into two sets; subscription spaces that fully cover the local advertisement space constitute the full coverage set FCS_t^p, and subscription spaces that partially cover the local advertisement space constitute the partial coverage set PCS_t^p. More formally, $FCS_t^p = \{LSS_t^s \mid s \in S_t \wedge MS_t^{p,s} = LAS_t^p\}$, and $PCS_t^p = \{LSS_t^s \mid s \in S_t \wedge MS_t^{p,s} \subset LAS_t^p \wedge MS_t^{p,s} \neq \emptyset\}$. Subscription spaces in the full coverage set are always LAS-equivalent (i.e. $\forall\ x,\ y \in FCS_t^p$, *LAS-equivalent*$(x,\ y)$). Subscription spaces in the partial coverage set may or may not be LAS-equivalent. Finally, the non-equivalent coverage set $NECS_t^p$ of a local advertisement space LAS_t^p with respect to a set of subscriptions S_t consists of the set of all non-empty match spaces of LAS_t^p with respect to the subscriptions in S_t. Thus $NECS_t^p = \{MS_t^{p,s} \mid s \in S_t \wedge MS_t^{p,s} \neq \emptyset\}$. Note that since $NECS_t^p$ is a set, none of its elements are LAS-equivalent.

4.3.3 Detecting Missing Events

Having described the basic service and event models, we discuss how to detect when events sent from the publisher to the mediator are lost in transit[1]. A simple mechanism for achieving this is for publishers to attach sequence numbers to each event that they send. By examining the sequence numbers of events received, the mediator can determine when events sent by a publisher have been lost.

From the perspective of a consumer however, loss detection capabilities can be divided into two different classes, depending on the covering relationship between the consumer's subscription and the local advertisement-space of the publisher. If the subscription completely covers the local advertisement-space (i.e. $LSS_t^s \in FCS_t^p$), then the precise number of lost events that would have been of interest to the subscriber can be determined. We call this level of loss detection *exact-count* loss detection.

In contrast, if the subscription only covers a portion of the local advertisement-space (i.e. $LSS_t^s \in PCS_t^p$), then some of the events lost in transit may not have been of interest (i.e. they would not have matched the subscription). In this case, we can only determine an upper bound on the number of lost events that would have been delivered to the subscriber. Therefore, we refer to this level of loss detection as *max-count* loss detection. Exact-count loss detection is more useful when trying to perform reliable complex event detection.

In the discussion above, we have concentrated on how a subscriber might detect losses in the event stream generated by a single publisher. In general, a subscription may intersect with the local advertisement space of multiple publishers, each of whom will have their own sequence numbers. Obviously, loss detection in these situations must take into account losses in every

[1] We do not attempt to deal with errors arising from publisher failures.

publisher's event stream.

4.3.4 Extended Service Models

In this section we describe some simple variations on the basic service model, and discuss how they affect our ability to detect missing events.

- *Publisher Hosted Filtering:*
  In some situations, publishers may not have enough memory to ensure guaranteed storage of all detected events. For example if events are published at a high rate, and there is a temporary network partition, then the memory capacity of the publisher may be exceeded. In such a scenario, we may be able to push some of the subscriptions for whom only max-count loss detection is possible from the mediator to the publisher. If events generated by the publisher are first filtered locally, and only then sequenced, then exact-count loss detection becomes possible for these subscriptions. The fact that any subscriptions pushed use a constant amount of memory, independent of the rate at which events are generated, means that the complexity of publishers can remain low. However, the publisher must either send a separate event for each matching subscription space in $NECS_t^p$, or batch sequencing information for each matching subscription space in $NECS_t^p$ in a single copy of the event.

- *Reliable Publishers:*
  For expressions with latency requirements, delayed events, as opposed to lost events, can result in gaps in input streams. Indeed for an alternative reliable service model, where publishers are capable of storing events (and possibly subscription filters), gaps in input streams only arise for expressions with latency requirements. In both cases, gaps can be detected using the same sequence number mechanisms as for lost events.

- *Unreliable Mediator:*
  In the service models so far, we imposed the constraint that event delivery within the mediator is reliable. If the mediator is also unreliable (i.e. nodes don't have persistent storage), then care must be taken when pushing subscription filters to the publisher. In particular, filters of subscriptions for whom only max-count loss detection is possible at the detector must be pushed all the way to the publisher if exact-count loss detection is desired. Storing the filter at an intermediate node might help to reduce unnecessary communications, but will not enable exact-count loss detection. This is because the intermediate node could fail, losing any sequencing information it holds about the publisher's event stream.

4.4 Detection Policies

Having introduced our event and service models, we now discuss how a complex event service can cope with various kinds of system error.

4.4.1 Policies

In our system, a user can specify how to handle different types of failure using a variety of *detection policies*. Some useful detection policies might include:

- Guaranteed: Under a guaranteed detection policy, detectors block until they are sure they have received all input events from every publisher. In a distributed system, this may delay event detection indefinitely.
- Best Effort: Under a best effort detection policy, events are delivered to a detector in the order they arrive. Events that arrive out of order with respect to previously delivered events from all sources are discarded.
- Max-Delay: With a max-delay policy, a subscriber can specify a maximum time to wait for delayed events before assuming they are lost. Events arriving after the timeout are discarded under a best-effort semantics, or cause the failure of detection if a guaranteed semantics is required.
- Probabilistic: This is a similar policy to the max delay policy, except the subscriber specifies a confidence level instead of a max delay. The detector can then attempt to model the expected latency between it and any relevant publishers, and detection can proceed when the probability there is no message in transit exceeds the required level. The subscriber does not need to know about the expected latencies between publishers and the detector. Similarly to a Max-Delay policy, late arriving events are discarded under a best effort semantics, or cause detection to fail under a guaranteed semantics.
- No-False-Positives(NFP): An NFP policy ensures that every event a detector generates actually occurred. For example, in the packaging application described earlier, a no false positive detection policy would ensure that no erroneous information is added to the database regarding which packages are stored in which container. In contrast, a best effort policy may generate erroneous events, and a guaranteed detection policy may fail unnecessarily when events are lost, delayed, or have conflicting timestamps.
- No-False-Negatives(NFN): In contrast to an NFP policy, an NFN policy ensures that a subscriber is notified whenever there is any possibility of an event having occurred. An effective implementation of an NFN policy should minimize the number of false positive notifications.

More specific policies can also be designed that attempt to handle one particular type of error, while ignoring another. For example, in many cases a user may wish to use the midpoint of an uncertainty interval timestamp to approximate an event's occurrence time.

4.4.2 Language Integration

Since a detection policy affects the semantics of a complex event, it should be specified as part of the complex event expression. We have extended the language described in the previous chapter with an optional `DETECT` clause for specifying policies. The syntax of the clause is given below:

```
[DETECT (BEST-EFFORT | [MP-]NFP | [MP-]GUARANTEED)
        [(TIMEOUT ...)|(C-LEVEL ...)]]
```

Currently, we support all the policies suggested in the previous section except for the NFN policy. Our `BEST-EFFORT` detection policy ignores gaps, and maps each uncertainty interval timestamp to the midpoint of the interval. A plain no-false-positive (`NFP`) detection policy handles both gaps and uncertainty intervals. However if the `MP-` (mid-point) prefix is specified, uncertainty intervals are mapped to their midpoints by the detector and only gaps can cause errors. Guaranteed detection fails on the occurrence of gaps or a set of conflicting events. Alternatively, an `MP-` prefix allows uncertainty intervals to be ignored.

Policies can be parametrised with either a stabilization timeout or a confidence level. A stabilization timeout allows the specification of a max-delay policy in combination with any of the other policies. Note that for an `NFP` policy with a timeout, late arriving events are discarded. The confidence level parameter requires the system to compute a suitable delay in order to ensure that all input events have arrived at the detector with a certain probability.

Detectors with different policies may be composed together. However, the policy specified for a detector is only enforced over the events and error information generated by its direct inputs. This means that the semantics of detection policies are local to a detector. As an example, consider a complex event with a guaranteed detection policy that has an input generated by a best-effort complex event detector. Any errors in the input streams of the best effort detector will be cleaned by that detector, and therefore will not cause a failure of the guaranteed detector.

One final issue that arises when composing detectors is how detection timeouts at different levels should be handled, since each detector may have its own max-delay bounds for detection. The latency bounds on higher level detectors should be greater than or equal to the latency bounds of all input complex event detectors. From the perspective of policy specification, this could be enforced by having latency bounds at each level be in addition to any maximum latency bounds specified for lower level detectors. Alternatively, we could require that the latency bound for a detector indicate the total maximum latency, and that this be greater than the total maximum latency of any input detector.

4.5 NFP Policy

In the remainder of this chapter we focus on the implementation of an `NFP` detection policy.

4.5.1 NFP Detection

In the presence of gaps or events with conflicting uncertainty interval timestamps, the output of stabilization is an ordered sequence of *valid* and *invalid* stages. A valid stage consists of a set of concurrent events with no gaps or events with conflicting timestamps. A valid stage occurs at a single point in time, and thus corresponds to the stage concept introduced in the previous chapter.

In contrast, an invalid detection stage contains a gap and/or a set of events that cannot be ordered due to conflicting uncertainty interval timestamps. An invalid detection stage may thus occur over a time period, and can represent one or more valid detection stages. However, due to the incomplete nature of the information contained in the invalid stage, it may be impossible to say exactly how many.

4.5.1.1 States of a Detector

Immediately after start-up, a detector is in a correct state, since no invalid input has been received. As long as it receives an ordered stream of valid stages it remains in a correct state. When an invalid stage occurs, the detector transitions into one of several different error states. The possible states of a detector can be summarized as:

```
CORRECT | ((FULL | PARTIAL) (PERMANENT | TEMPORARY)) ERROR
```

Error states can be classified along two main dimensions, *severity* and *duration*. A full severity error state indicates a detector that is not capable of producing any correct output. In contrast, a partial severity error state can produce some correct output events, but may be missing some, due to the invalid stages.

A detector in an error state of permanent duration will never have the severity of its state's error reduced. Conversely, an error state with temporary duration indicates that it is possible the detector will return to a reduced severity error state at some point.

A detector in a full permanent error state can never determine any useful information about the correctness of its output. Thus a detector in this state should typically fail. An example of a partial permanent error state is a detector for the expression:

```
CreateCE PartialPermanentErrorExample
Event And(A, B)
```

If this detector receives an invalid stage, it will never be deleted. Thus although it may continue to generate output, it will never be perfectly correct. Figure 4.3 shows the transitions possible between detector states.

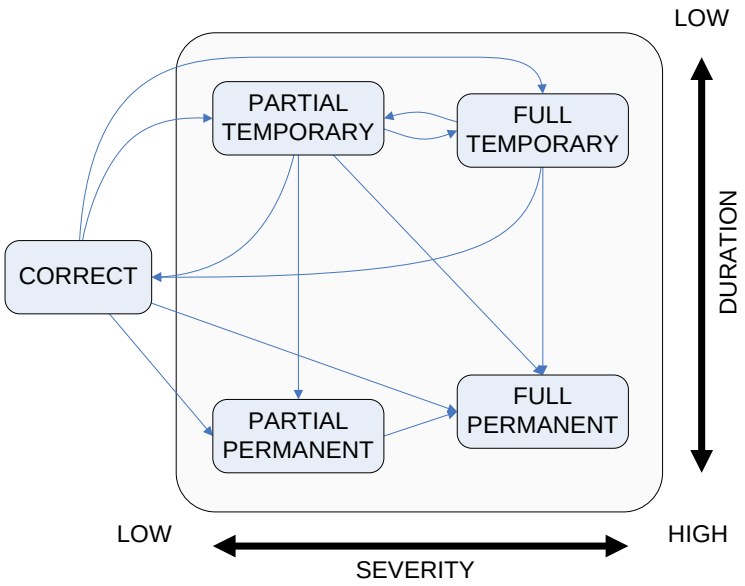

Figure 4.3: NFP Detector State Transition Diagram

4.5.1.2 Output Convergence

When a detector is in an error state with temporary duration, a return to a correct state (or an error state of lower severity) results in the *convergence* of its output stream towards a perfectly correct output stream. Since output is generated based on the state of operands' event histories, convergence occurs when the information contained in invalid stages is no longer relevant. During normal operation, events can become irrelevant in four ways: expiry, override, consumption, and context termination.

Event *expiration* occurs when detection is performed over a temporal sliding window, and the window slides over the event. We use the term *override* to describe situations where an event is discarded due to the arrival of new events, independently of whether any output is generated by either the new or old events. A typical example of this is an operand with a tuple based sliding window, where the arrival of new events causes older events to be discarded. Next, *consumption* is the removal of an event from an event history in response to the detection of some output event of which it may have been a constituent. Note that in our classification, the recent 'consumption' policy of Snoop [CKAK94] causes events to be overridden, not consumed. The chronicle consumption policy of Snoop is closer to our definition of consumption. Finally, the *termination* of a context instance results in all the information contained in its operands being discarded.

Determining the state of a detector is a two stage process, namely the static and dynamic analysis phases. Static analysis of a detector's expression occurs at compile time. It firstly determines the loss detection capabilities of the underlying delivery service. Secondly, it looks at whether and how events delivered to the detector can be deleted. From this information it determines the possible states of the detector, and also the possible transitions between states

that can occur. As an example, the static analysis may determine that all invalid stages cause the detector for an expression to enter an error state from which it will never exit. Thus at run-time the detector should fail immediately whenever it receives an invalid stage.

At run-time the occurrence of an invalid stage triggers further dynamic analysis. The information needed to perform this dynamic analysis includes the current state of the detector, the invalid stage, and all future stages (both valid and invalid) up until convergence has occurred. Obviously, the dynamic analysis will be simpler for some expressions than others. For example, if events can only be discarded through expiry (e.g. in a temporal sliding window), then all that needs to be analysed is the time since the occurrence of the invalid stage.

Currently, we do not allow users to limit the amount of time taken for an erroneous detector to reconverge to a correct state. Instead, the decision is taken statically. It should be possible to extend the language quite easily with parameters to control the maximum allowable time for convergence.

4.6 Implementation

In this section we discuss our implementation of an `NFP` detection policy for the complex event detection language described in the previous chapter.

Our current implementation handles most, but not all, of the cases where convergence is possible. In particular, it does not attempt to perform detection for expressions with a condition specified in the `WHERE` clause. A more sophisticated implementation could improve on this, especially if the `WHERE` clause refers only to the temporal attributes of events. Conditions that refer to non-temporal attributes will typically be more difficult to handle. However, in some cases information about the content of a missing event can be gleaned from its publisher's advertisement. This may help in determining whether the missing event would have matched the condition.

The first task for our extended detection algorithm to perform is stabilization of incoming events. The outputs of stabilization are valid stages and invalid stages. As described earlier, a valid stage contains either a single event or a set of truly concurrent events. An invalid stage may contain a gap in the input stream, or a set of events with conflicting uncertainty interval timestamps, or both. Note that a globally invalid stage may become a sequence of valid stages with respect to a single complex event expression when only some of the events represented by the invalid stage are relevant to that expression. For example if only a non-conflicting subset of a set of conflicting events are relevant to a detector, the irrelevant events can be filtered, leaving a sequence of valid stages.

As long as no invalid stages are received by a detector, detection can be performed using the algorithm described in the previous chapter. However, as soon as an invalid stage is received, a detector with an `NFP` policy must perform collection and detection using our extended detection algorithm. Until the detector returns to a correct state, both valid and invalid stages must be processed using this extended algorithm. Detection for a complex event using an `NFP` detection

policy is performed by the *process-ce* pseudo-code given below:

```
1 process-ce(ce, stage)
2   if (correct?(ce) ∧ valid?(stage))
3      normal-process(ce, stage)
4   else
5      nfp-process(ce, stage)
```

If the detector is in a correct state, and a valid stage is received, then normal detection can be performed using *normal-process* (line 2). However, if an invalid stage is received or the detector is in an error state, we perform NFP detection (line 5). We give a pseudo-code description of the *nfp-process* function below:

```
1 nfp-process(ce, stage)
2   ce := nfp-collect(ce, stage)
3
4   if (!failed?(ce))
5      if (should-branch?(ce))
6         if (single?(ce))
7            ce := single->multi(ce)
8         else
9            ce := single->multi(multi->single(ce))
10
11      if (valid?(stage))
12         if (consumable?(ce))
13            nfp-consume(nfp-detect(ce))
14         else
15            nfp-detect(ce)
```

We firstly perform collection of the new stage (line 2), in accordance with the processing model defined for our language in Chapter 3. If the state of the complex event `ce` is unaffected by consumption, collection is straightforward. For a valid stage, collection simply adds the stage events to matching operands, and applies any windows that are defined. For an invalid stage, information about missing and conflicting events must also be added.

A complex event `ce` is consumable when at least one of its operands is consumable (i.e. has its consumption flag set to `consume`). If `ce` is consumable, a previous invalid stage may have required us to *branch* (i.e. create multiple copies of) the detector's state. In this situation, collection adds the new stage to each branch independently. Note that a detector maintains these branches internally, and they are not visible to external entities such as subscribers. Given an invalid stage, a detector must determine all the possible orders in which the events and gaps it contains may have actually occurred. It then creates and speculatively executes a separate

branch for each order. By inspecting the input operands of different branches, a detector can subsequently determine whether they are equivalent and can be merged into a single branch. A detector converges back to a correct state when only a single branch remains.

Having finished collection, the next step of our algorithm checks whether `ce` is in a permanent error state (line 4). If it has then detection fails. If not, then before performing detection, the algorithm checks whether branching is required (line 5). Branching is only necessary if `ce` is consumable, and can be avoided in many cases. One such case is when `ce` could not possibly be fired by `stage`, meaning its next state will not be affected by detection. Note that if `ce` has previously branched, we collapse it back to a single worst case state before re-branching it (line 9). Although this delays convergence, it prevents a combinatorial explosion of branched detectors.

Finally, if `stage` is a valid stage, we attempt to perform detection using the information stored at each operand (lines 11 - 15). This information typically includes an upper and lower bound on the number of events, deduced from the events, gaps, and conflicting events added to each operand during collection. If `ce` is consumable, then we must also update its state after detection (lines 12 - 13).

Note that no output is generated by *`nfp-process`* in response to the occurrence of an invalid stage. This is because it is not clear what the timestamp of any output should be. In contrast, the occurrence of a valid stage for a detector in an erroneous state may result in some output being generated by *`nfp-process`*.

4.7 Evaluation

In this section we give the results of some experiments we performed using our implementation of an `NFP` detection policy. Our evaluation uses a simple simulator that takes in a test input trace containing errors, the corresponding error-free input trace, and a complex event expression. For these experiments, we used the example described earlier in Section 4.2.3 (and repeated below) as a test query.

```
CreateCE PackageContainer
Select Package.pkgId, Container.contId
Event And(Package<old max 3, consume>,
          Container<old max 1, consume>)
```

Since its operands are consumable, the query allows us to examine the operation of our *`nfp-process`* algorithm for a relatively complex class of expressions.

For queries without consumable operands, the handling of errors introduces negligible overhead to the detection algorithm. For queries with consumable operands, the algorithm may have to maintain multiple copies of each consumable operand in order to determine convergence. In the worst case, $O(2^{n_{co}})$ copies of each consumable operand are maintained by our algorithm, where n_{co} is the number of consumable leaf operands in the query. In addition, worst case processing times increase by a factor of $2^{n_{co}}$. However, for many queries optimizations that enable

us to avoid this worst case are possible.

We examine three main aspects of our system as part of our evaluation. First, given an input trace containing some errors, we investigate which factors affect the accuracy of the output stream generated by the detector. Second, we analyse how varying the error rate affected the accuracy. Third, we determine the processing cost incurred during recovery from errors in input streams. All our experiments involved feeding pre-generated input traces into our detection engine and observing the resulting behaviour, such as the correctness of the resulting output and the total execution time.

4.7.1 Experiments

We generated an input trace of `Package` and `Container` events containing 5000 stages. The type of event generated at each stage was determined probabilistically using a random variable T, where $Pr\{T = Package\} = p_{pkg}$ and $Pr\{T = Container\} = p_{cont} = 1 - p_{pkg}$. The expected ratio R of `Package` to `Container` events is therefore given by $R = p_{pkg}/p_{cont}$. We refer to this input trace as the *correct* input trace. We also created a second input trace by introducing errors in the form of gaps into the `Package` event stream of the correct input trace. These were created randomly with a probability p_{gap}.

We generated three output traces based on the input traces described above. The first output trace was generated by a best-effort detector, the second using an `NFP` detector. Both of these were generated based on the input trace containing errors. We also generated a correct output trace based on the correct input trace for comparison. The correct output trace thus models the output stream that would have been generated if no gaps occurred in the input stream.

4.7.1.1 Correctness

In this experiment, we evaluated the accuracy of the output generated by the `NFP` detector in comparison to the best-effort detector. Since the output traces were finite, we used the information retrieval notions of *precision* and *recall* to compare performance. Precision (PRN) measures the fraction (percentage of events) of a test output trace (i.e. the best-effort or the `NFP`) contained in the correct output trace. Conversely, recall (REC) gives the fraction of the correct output trace contained in the test trace.

We varied two parameters for each run of the experiment. These were the error rate ($E = p_{gap}$) of the package event input stream, and the ratio ($R = p_{pkg}/p_{cont}$) of package events to container events. In total, we tested nine different combinations of these parameters. For each combination, we repeated the experiment three times and averaged the results. The results are summarized in Figure 4.4.

Two main trends are evident from these results. Firstly, the higher the error rate, the worse the performance of BE detection relative to NFP detection. Secondly, the higher the ratio of package events to container events, the worse the performance of BE detection relative to NFP

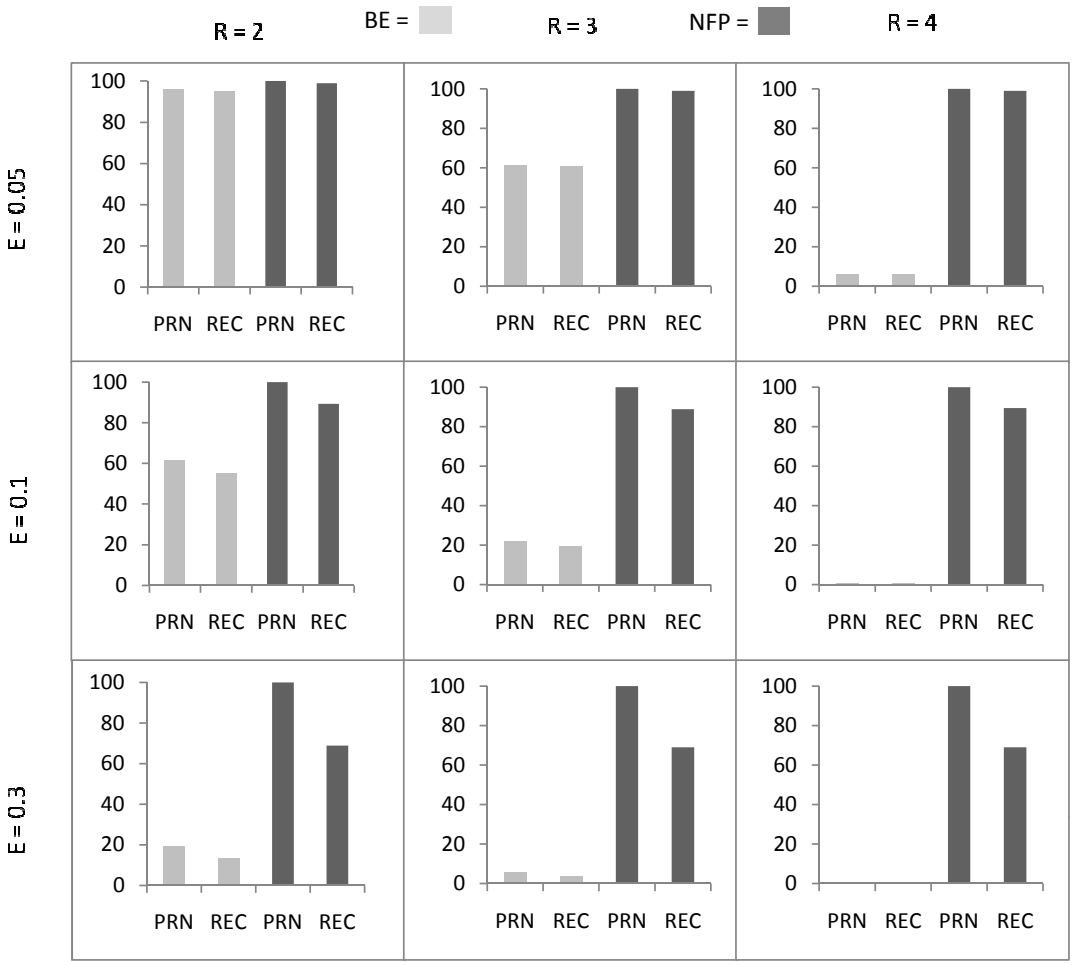

Figure 4.4: [Precision (PRN) (%), Recall (REC) (%)] for Best Effort (BE) and No False Positive (NFP) detection

detection. This is because cascading errors become more likely as the ratio increases. Note that in all cases, the precision of NFP detection is 100%.

4.7.1.2 Execution Time

We also measured the run times for each experiment. To do this, we fed each stage in the test input traces into the detector, and measured the time taken for processing of each stage to complete. We then summed the processing time for all stages to get a total processing time for each trace. The results are shown in Figure 4.5. All times are given in seconds.

The measurements show that the run time of both NFP and correct detection increase with

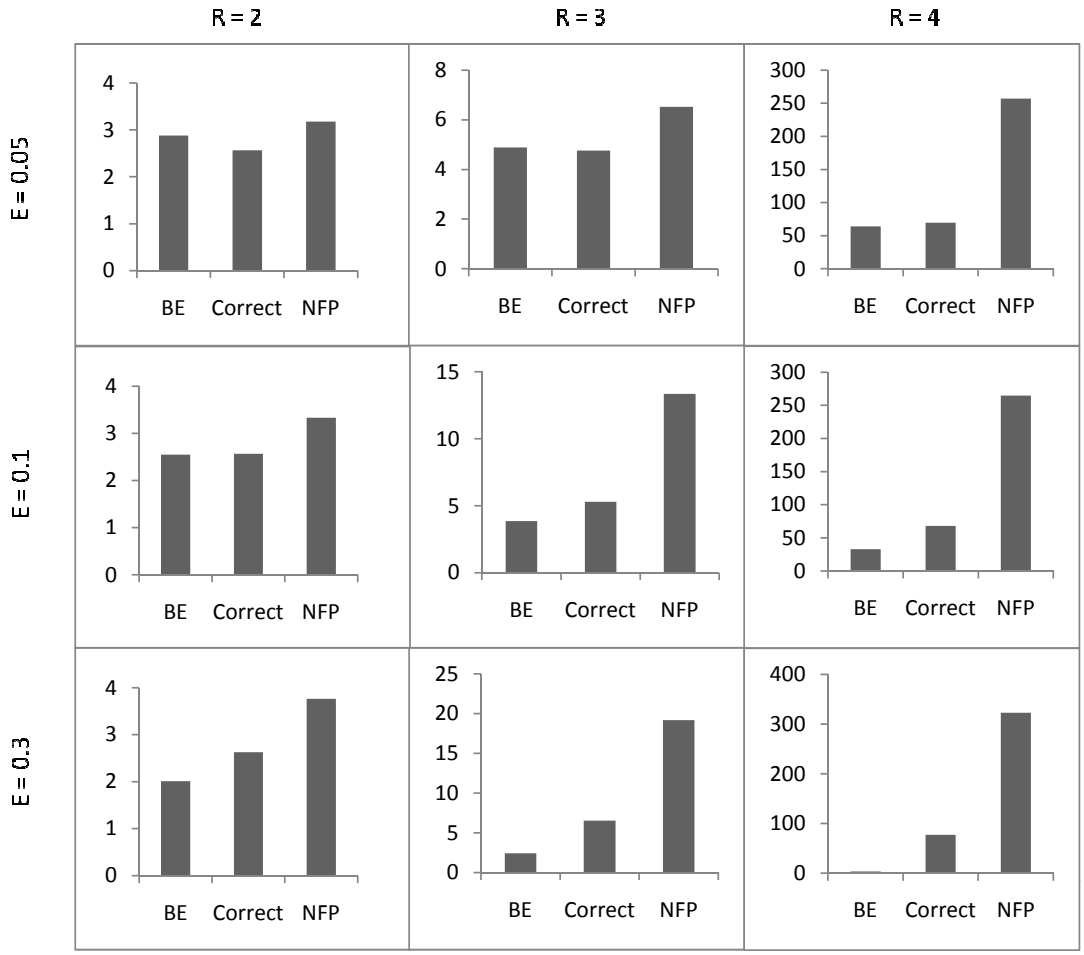

Figure 4.5: Cumulative processing times for Best Effort (BE), Correct, and No False Positive (NFP) detection

both increasing error rates and an increasing ratio of package to container events. The run time of best effort detection also increases when the ratio of package events is increased. However, the run time actually decreases when the error rate is increased. This is because the error events in our experiment are missing events, and are ignored by best effort detection.

The run time of NFP detection is generally higher than that of BE detection. However, this

is somewhat misleading, since the BE detection ignores missed events. A better comparison is with correct detection, which measures the run time that would have occurred if errors were replaced with correct events instead of being dropped. For high package to container event ratios and high error rates, the relative performance cost is still significant. For example, an error rate of 0.3 and a ratio of 4 causes NFP detection to take 4.19 times as long as correct detection. However, this extra run time occurs only when the detector is in an error state.

4.8 Related Work

Issues related to the modeling of time in distributed systems have been addressed by several designers of complex event detection languages.

Some early work on distributed systems issues relating to composite event detection is that of Hayton [Hay96]. He defines an event language that allows the specification of minimum and maximum probabilities for timestamps to deal with unsynchronized clocks. However he does not discuss how the semantics can be implemented. In many cases meaningful probability distributions may not be available. He suggests two mechanisms for dealing with delayed events, tunable heartbeats and maximum delay annotations. A delay mechanism requires the user to have knowledge of likely delays. In addition if quality of service is not binary (e.g. there is an advantage to performing detection when you are 95 per cent sure no further events will arrive), a confidence parameter is more useful.

Schwiderski [Sch96] proposed the use of a global clock granularity for ordering events during distributed composite event detection. This requires a granularity coarse enough to handle the most inaccurate clock in the system. In an open distributed environment, knowledge of the worst case granularity may not even be available. Her detection engine implementation supports two evaluation modes, asynchronous and synchronous, equivalent to our best effort and guaranteed detection policies. Similarly to Schwiderski, Yang and Chakravarthy propose a formal semantics for distributed composite event detection based on a global clock granularity [YC99]. However they do not address issues related to delayed or lost input events. Both Schwiderski and Chakravarthy represent composite event timestamps as a set of conflicting primitive event timestamps, and define joining procedures for merging two timestamps. This enables a style of detection similar to one that might be provided by a No False Negatives detection policy. However neither Schwiderski or Chakravarthy address the implications of conflicting event timestamps for event consumption in a manner consistent with our definition of correct detection.

Liebig et al. [LCB99] argue for the use of uncertainty intervals in event timestamps. Uncertainty intervals may be provided for example by a global time service such as NTP. They are more suited to open distributed systems than defining a global clock granularity. In addition they allow for more accurate event composition, since timestamps no longer need to be set to accommodate the worst case clock. We have adopted this timestamping approach in our composite event detection semantics. However, Liebig et al. simply raise an exception whenever a set of conflicting events occurs, they do not provide NFP detection, and they do not attempt to

deal with gaps in event streams.

DistCED is a distributed composite event detection service for event based middlewares [PSB04]. Similarly to us, it employs uncertainty interval timestamps to bound clock inaccuracy. It also provides operators (weak sequence, parallel) to detect patterns over primitive events whose ordering is unknown. However there is no way to control event consumption, and the interaction of consumption and conflicting uncertainty intervals is not addressed. DistCED does allow the specification of best effort, guaranteed, and probabilistic detection policies to deal with transmission delays and lost events. However there is no support for NFP detection.

Yoneki and Bacon [YB05] mention the difficulty of timing in Wireless Sensor Networks where no GPS is available. They propose lightweight local clock propagation for these situations. They also emphasize the distinction between durative event timestamps and uncertainty interval timestamps. However they do not discuss the interaction of uncertainty intervals and consumption, or attempt to provide detection policies.

GEM is an event monitoring language for distributed systems designed by Mansouri-Samani and Sloman [MSS97]. It allows rules to be annotated with tolerable delays in order to cope with late events. Instead of using a stabilization phase, events are delivered in arrival order, but detection is delayed according to explicit user annotations. This is similar to our Max Delay detection policy. In GEM however, detection is performed eagerly, with partially detected results materialized, and only invalidated when a late event arrives. The effectiveness of this approach in reducing detection latency depends very much on the composite event operator and consumption policy, since materialized events may be invalidated by new events even when there are no delays in event delivery. In addition, the GEM detection algorithm assumes a synchronized global clock.

Bauer [Bau04] discusses the design of an event notification service in the context of Nexus, a platform for mobile context-aware applications. Applications indicate their interest in a particular state of the environment using a predicate template (e.g. onEnterArea(⟨Person⟩,⟨Area⟩)). The service then notifies applications about activations and deactivations of this state. To deal with noisy sensor data, applications associate a *threshold probability* with each predicate. A high probability reduces the number of false positives at the expense of false negatives, and vice versa. Uncertainty interval timestamps for which a probability distribution is available can also be incorporated into their model. Although their language is quite different in style to ours, we believe much of our work is complementary. They do not address issues related to gap detection, and do not discuss cascading errors as the result of interactions between event consumption and no false positive detection.

Brito et al. suggest using a software transactional memory (STM) to speculatively process out of order events [BFSF08]. The STM ensures that this does not result in erroneous output. This approach can take advantage of the parallelism provided by multi-core CPUs to improve performance. However, it does not handle lost events or deal with issues relating to conflicting uncertainty intervals.

In addition to distributed complex event detection languages, several research efforts in the stream processing community on approximate query handling are related to our work. However,

they typically assume a simplified timing model where timing uncertainty is ignored. In addition, most stream processing languages do not support event consumption, simplifying error handling. We now give a brief overview of some representative work in the area.

The Stream query processor [MWA+02] of Motwani et al. is a stream processing system that attempts to perform both static and dynamic approximation of query results in cases of high load or limited resources. This is achieved through intelligent load shedding (e.g. by reducing window sizes or dropping tuples). Similarly to us, they model the precision of a stream of query results by measuring the number of false positives and negatives produced. They mention the usefulness of exploiting constraints over streams, expressed as adherence parameters. Our loss detection constraints (e.g. max count) could be viewed as an example of such a parameter.

Aurora is another stream processing system that attempts to provide approximate answers to queries under conditions of high load [CcC+02; ACc+03]. They enable users to specify quality of service requirements for queries using *QoS graphs*. Three types of QoS graph are supported. Delay based graphs are mandatory and similar to (although more expressive than) our Max Delay policy. Optionally, Drop based and Value based QoS graphs may also be defined. These specify a rough estimate of the percentage of tuples needed and the importance of tuples with different values respectively. Although many of their techniques used in Aurora are complementary to our work, they are more concerned with providing QoS in conditions of high load than with coping with errors introduced by conflicting event timestamps or lost primitive events.

The Borealis stream processing system [AAB+05; RMCZ06] extends Aurora in several directions including support for *revisions*. Tuples may be of three types: Insertion, Deletion and Replacement. Instead of discarding tuples that arrive late or are dumped during load shedding, Borealis can send revision messages to downstream operators (and to applications) to enable them to correct their state. Borealis also extends the QoS model of Aurora to enable the specification of QoS requirements for each box, instead of just query endpoints.

CEDR is an event streaming system that attempts to handle out of order events [BGAH07]. It can perform event processing at a variety of *consistency levels*. Strong consistency is similar to our notion of guaranteed detection. Middle consistency may produce erroneous output initially, but uses retractions and insertions to correct these errors when input events finally arrive. Weak consistency does not have to try and fix errors in output streams. We view CEDR as being complementary to our work, and may be useful in supporting a No False Negatives detection policy. However it is not clear whether CEDR can handle lost as opposed to delayed events, and the time model assumes a globally synchronised clock.

4.9 Summary

Complex event processing for pervasive computing must deal with various sources of error. We have focused in this chapter on handling errors caused by lost input events and timing uncertainty. We proposed the use of detection policies for specifying how to deal with these errors. We discussed several such policies, and under what conditions each policy might be

useful. We then showed how the language introduced in the previous chapter could be extended with a clause for specifying a detection policy. Finally, we described the implementation of a No False Positives (NFP) detection policy for our language. Our experiments show that such a policy can improve both the precision and recall of complex event detection in the presence of the aforementioned errors. Furthermore, the processing overhead required is negligible in the absence of errors.

Chapter 5

Detector Placement

5.1 Introduction

Event based middleware is a useful distributed programming platform for many pervasive computing applications. Its loose coupling enables the creation of highly scalable applications. Distributed complex event detection is a common service to provide as part of an event based middleware. In addition to enabling the specification of high-level events, a distributed complex event detection service is advantageous for a number of reasons, such as its ability to reduce the load on resource-constrained clients.

A key problem when distributing complex event detection is the efficient placement of detectors. Several factors may need to be considered, including latency, bandwidth, processing, storage, and reliability. For many placement policies, the optimal placement depends on the routing algorithm. Current approaches have been based on event based middlewares with optimal routing paths for events [LJ05]. In contrast, rendezvous based publish/subscribe middlewares sacrifice route optimality in order to improve scalability by reducing routing state. It is not clear how placement over a rendezvous based middleware should be performed for several common policies, or whether such a middleware is even suitable.

In this chapter, we investigate this problem using a rendezvous based P2P middleware called Hermes. We describe several placement strategies for efficient complex event detection over Hermes. The best choice of strategy depends on the likely reuse of sub-expressions within a complex event tree. We discuss some reuse patterns that are likely to be common for pervasive computing applications, and how our placement strategies perform with respect to these patterns. The main contributions of this chapter are:

- A discussion of the main benefits of distributing complex event detection, with an emphasis on the desirability of cross-domain complex event detection for pervasive computing applications.
- An analysis of two *static* placement strategies for complex event detectors over a rendezvous based middleware. Although these strategies are simple to implement, they have several disadvantages, including a lack of support for cross-domain complex event detection.

- A *dynamic* placement strategy for complex event detectors over a rendezvous based middleware. This strategy maximizes cross-domain complex event detection, but can still detect events efficiently in the central domain.
- An evaluation of the network usage of each of our strategies. Using complex event expressions exhibiting a variety of reuse patterns, we compare these strategies to a reference strategy from the literature. Our experiments indicate that the dynamic strategy performs well when reuse is high, having roughly the same network usage as one of the static strategies, while also allowing for cross-domain detection.

5.2 Motivation

5.2.1 Multi-Domain

As motivation for our work, we envision an Active City scenario where an event-based middleware supports intelligent transport applications. Sensors are deployed throughout the city to detect phenomena of interest, such as congestion, accidents or dangerous road conditions. Sensors send the information they generate via our middleware to users. We envision the middleware being deployed over an open shared infrastructure to reduce costs. This allows sensor data to be reused for different purposes by different parties (subject to security and privacy constraints). An open infrastructure such as this helps to maximize the benefits of deploying sensor networks, since they are typically expensive to create.

The middleware uses a publish/subscribe communication paradigm, where sensors act as publishers, and users as subscribers. This *data-centric* communication [1] is particularly suited to pervasive computing applications, since users are typically more interested in sensor data than in the sensor nodes themselves. As in Chapter 3, we anticipate the separation of the middleware into several distinct *domains* [Hom02]. Sensors will typically be lightweight and energy constrained. They connect to the infrastructure via a base station, which will typically have more resources and be connected to a power supply. A central domain is responsible for connecting the different sensor domains together.

Our architecture relies on an event based middleware called Hermes [Pie04] in the central domain. Hermes uses peer-to-peer techniques to provide a scalable and robust publish/subscribe communication service. Hermes disseminates events using a *type- and attribute-based* routing algorithm, whose basic operation we depict in Figure 5.1. The algorithm assigns to each event type a rendezvous node (R) based on the hash of the type name. Event publishers connect to brokers (PHB1 and PHB2), and route *advertisements* for the events they wish to publish towards the appropriate rendezvous. Subscribers also connect to brokers (e.g. SHB), through whom they route subscriptions. Subscriptions to an event type are forwarded to the rendezvous, while also following the reverse path of any advertisements they encounter en route. Finally, publications are generated by publishers and follow the reverse path of matching subscriptions.

[1] As opposed to *node-centric*.

A more detailed description of the Hermes type- and attribute-based routing algorithm is given by Pietzuch [Pie04].

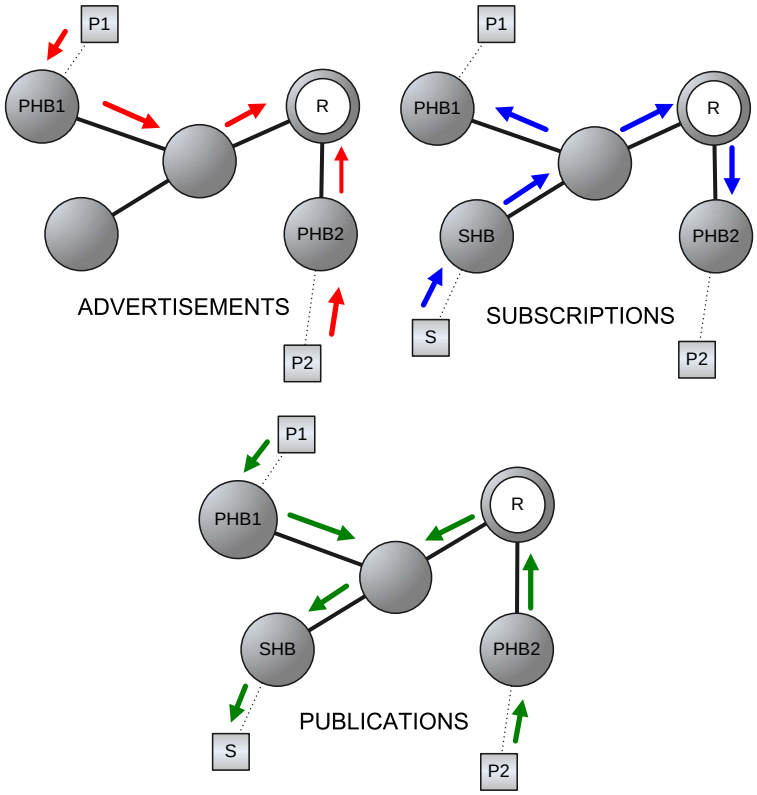

Figure 5.1: Hermes Type- and Attribute-Based Routing

5.2.2 Benefits of Distributed Detection

In [PSB04], a framework is described for providing *distributed* complex event detection as an extra service for event based middlewares. Instead of consumers performing complex event detection locally, a complex subscription is sent to the service. It then performs detection of the complex event within the broker network. There are several beneficiaries when a complex event service is provided in a distributed fashion by the middleware.

Subscribers Without a distributed complex event detection service, subscribers must accumulate events and detect patterns themselves. In contrast, complex event detectors hosted by event brokers can act as filters over event streams, reducing unnecessary communications between a subscriber and its host broker. This is particularly important when the link between subscribers and subscriber hosting brokers is bandwidth constrained (e.g. for wireless devices).

As an additional benefit, a complex event detection service reduces processing and storage requirements at the subscriber. This is advantageous when the subscribing device is lightweight and resource constrained. If many subscribers are interested in the same complex event, the

total amount of processing and storage needed for detection may also be reduced, depending on where detectors are placed, since the same detector can be shared by subscribers. *Decomposition* of complex events by the complex event detection service further increases the opportunities for reuse by enabling the sharing of common subexpressions.

Finally, a complex event service can sometimes detect patterns that a subscriber cannot detect itself due to privacy constraints. This happens when privacy-sensitive attributes of input events are needed to detect a complex event, but are not used to generate its output attributes. So long as the values of the sensitive attributes are not mapped to the attributes of the complex event, a subscriber can delegate detection of the pattern to the complex event service and still respect privacy constraints (assuming the service is trusted).

Event Brokers Distributed complex event detection can also help to improve the performance of the event broker network itself. Pushing detection of complex events close to publishers allows irrelevant events to be filtered nearer to their source. This helps reduce the bandwidth needed by a complex subscription. However, detector placement may need to take other metrics into account, including detector reuse and event delivery latency.

Publishers Apart from reducing communications over subscriber links and within the broker network, in certain cases it may be possible to reduce communication with event publishers using distributed complex event detection. We envision such an optimization being especially useful when event publishers are acting as gateways to other domains (such as a wireless sensor network), and Hermes connects these domains to subscribers. Such *cross-domain* complex event detection has been suggested in [Hom02]. Pushing complex event detectors to an event publisher acting as a gateway to a WSN gives the gateway the option of further distributing the detector into the WSN. This can help reduce unnecessary communications in WSNs, which is very important in conserving the energy of power constrained sensors.

5.3 Complex Events Over Hermes

An expression describing an event pattern is created by a subscriber based on the event types known to be available in the system. We will assume that each expression gets parsed into a tree of detectors. The expressions defined by all subscribers together form a directed acyclic detector graph. We refer to this graph as the *query graph*. If the language allows the outputs of an operator to be renamed, then checks must be made to ensure that no cycles occur in the query graph, (e.g. due to an output event name clashing with a pre-existing primitive or complex event name). How this checking is achieved is discussed further in Section 5.6.2.2.

Once the complex event expression has been parsed, the task is to merge the resulting detector tree into the existing query graph, placing any new detectors on appropriate nodes in the network. This placement can be done in various ways, depending on the routing algorithm of the publish/subscribe middleware, and which aspects of the distribution are to be optimized.

However before discussing placement strategies, we will firstly describe the ways in which the query graph can be decomposed.

5.3.1 Decomposition

Decomposition of the complex query graph can be divided into two classes, hierarchical and partitioned.

5.3.1.1 Hierarchical Decomposition

The first type of decomposition we discuss is *hierarchical*. Given a complex event expression that has been converted into a tree of detectors, hierarchical decomposition splits off the internal nodes representing subexpressions of the top level complex event. Subexpressions whose total input data rate is greater than their output data rate are good candidates for distribution. Hierarchical decomposition also increases reuse, since detectors for common subexpressions of complex events created by different subscribers can be shared.

5.3.1.2 Partitioning

Another type of decomposition is *partitioning* of complex event expressions. This is slightly different from hierarchical decomposition, in that it divides, based on their attribute values, the possible output event instances for a complex event type, and the input event instances used to generate them, into several disjoint partitions. Multiple detectors are created for the type, each responsible for generating a different subset of the events. For pervasive computing applications, complex events partitioned by geographic location are particularly common.

Take for example the complex event `SlowingVehicle`:

```
CreateCE SlowingVehicle
Select Before.loc as loc
Event Sequence(AverageSpeed as Before <"speed > 60">,
               AverageSpeed as After <"speed < 10">)
GKey *.loc as GeoKey (In "Madingley Road" as Madingley,
                      "Bridge Street" as Bridge,
                       Rest as Other)
```

This expression detects speed reductions by vehicles at a location. The global key `GeoKey` divides detection of `SlowingVehicle` into three partitions, one for each of its regions (i.e. `Madingley`, `Bridge`, and `Other`), such that a separate detector is responsible for each partition. The first two detectors are responsible for detecting `SlowingVehicle(loc = "Madingley Road")` and `SlowingVehicle(loc = "Bridge Street")` respectively. The third detector is responsible for detecting the rest of the type's attribute space, i.e. `SlowingVehicle(loc != "Madingley Road" AND loc != "Bri-dge Street")`. Each `AverageSpeed` input event is routed to one of the detectors based on the value of its `loc` attribute.

The detector for each region has an associated set of query graph neighbours, containing publishers of its input events and subscribers to its output events. If the neighbours of any two detectors are not equal, then their optimal placement in the network may also differ. In such a scenario, separate detectors enable more efficient distributed detection by the middleware.

5.4 Static Placement

Having discussed our motivations for distributing detection, and the types of query decomposition we expect to perform, we now describe several strategies for placing detectors in the overlay network. We begin by introducing two *static* placement strategies. These strategies are relatively simple to implement, but do not attempt to relocate detectors in response to changing network conditions.

5.4.1 Rendezvous Broker Placement

Our first static detector distribution strategy over Hermes is rendezvous (RV) placement. This strategy places the detector for a particular complex event type at the rendezvous broker whose address matches the hash of the type name. It achieves this by routing new complex subscriptions towards their rendezvous broker. If no detector exists, the rendezvous broker creates a new one. The broker then sends subscriptions to the rendezvous brokers of sub-detectors and primitive input events.

Once detectors have been set up, events flow from primitive publishers to the rendezvous of their parent detectors in the query graph, and in turn to the rendezvous of their grandparents' detectors, and so on until the top level rendezvous is reached. This results in reduced traffic on the link between the subscriber and the broker hosting it, in addition to moving computation into the broker network.

Although simple to implement, the problem with rendezvous placement is that at each level of the query graph, a different rendezvous broker must be used as the destination for routing. Since rendezvous brokers are effectively distributed randomly, this could result in an unacceptable latency penalty when detecting complex events with multiple levels. Figure 5.2 gives an example of rendezvous detector placement for the complex event `F = f(G, C)`, where `G` is in turn a complex event such that `G = g(A, B)`. The detector for `F` is hosted by its rendezvous broker (R_F). Similarly, the detector for `G` is hosted by its rendezvous broker (R_G). Boxes represent publishers (P_A, P_B, and P_C) and subscribers (S_F1 and S_F2). Labeled arrows give the paths followed by event notifications, both primitive and complex, through the overlay network. Note that the only publishers for types `A` and `B` (i.e. P_A and P_B), are connected to the same broker. However their publications must travel to the detector for `G` at the other side of the network. A better placement strategy might push the detector for `G` closer to its inputs in order to reduce network traffic.

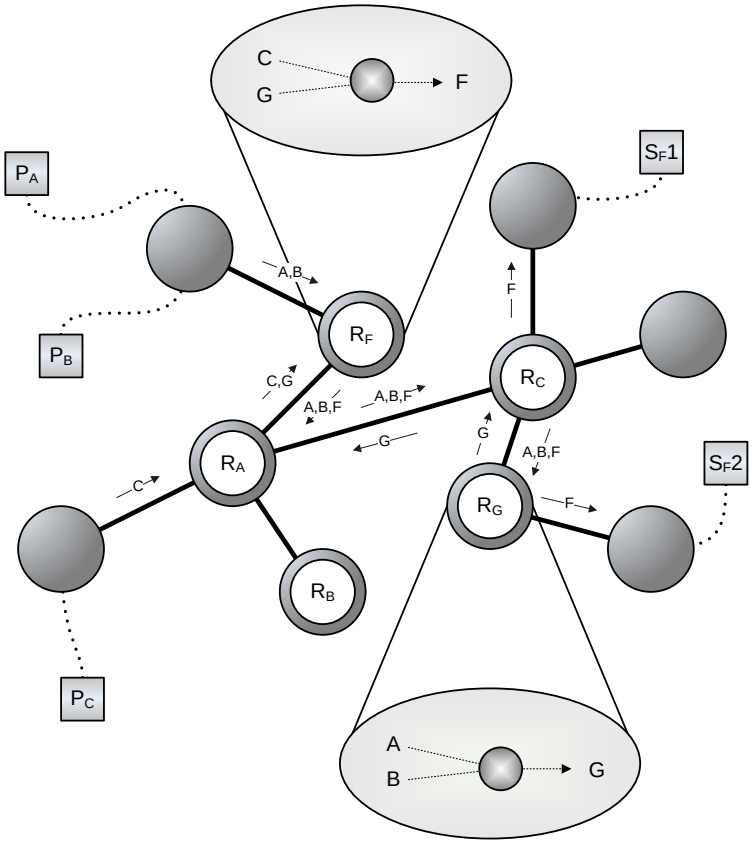

Figure 5.2: Rendezvous Broker (RV) Placement

5.4.2 Subscriber Hosting Broker Placement

An alternative static detector placement strategy is to place all detectors at the broker to which the subscriber is connected. All detectors for the complex subscription remain at the subscriber hosting broker. Subscriptions to the primitive events needed by detectors are made using Hermes. We refer to this strategy as subscriber hosting broker (SHB) placement.

This solution is distributed only insofar as clients offload detection to the broker network. However, cross-domain detection (see Section 5.2.2) is not possible unless the subscriber happens to connect to the gateway broker for a domain. Furthermore, sharing of detectors for common subexpressions is only possible for subscribers at the same broker. Finally, there is no opportunity to reduce network usage by placing detectors nearer to publishers. However, in situations where the network is saturated with subscribers for all primitive event types, it is possible that distributing detectors into the network may not result in worthwhile reductions in network usage anyway.

As an example, Figure 5.3 shows how detectors for the types `F` and `G` (as used to explain RV placement in Figure 5.2), are distributed by an SHB placement strategy. Since there are two subscribers to `F` at separate brokers, two detectors for `F` (and for `G`) are created, one at each subscriber hosting broker. Only primitive event notifications travel through the overlay network, with detectors performing complex event detection locally.

5.5 Dynamic Placement

In contrast to static placement strategies, dynamic detector placement allows us to exploit opportunities for cross-domain complex event detection anywhere in the overlay. We now propose such a strategy for Hermes. We begin by discussing how a broker network can be modeled as an approximate *latency space*, with each broker having its own *network coordinate*. We then describe briefly how other researchers have used network coordinates to optimize, within a latency space, the relative position of detectors in a query graph. Finally, we discuss our dynamic placement strategy for Hermes, called Hermes Network Coordinate (HNC) placement, which relies on event brokers extended with network coordinates.

5.5.1 Network Coordinates

Virtual network coordinates are a mechanism for approximating the latency between different nodes in a network [DCKM04; PCW+03; TC03; GSG02]. They allow us to think of each node as residing at some position in a *latency space*. The best type of coordinate space to use depends on the structure of the underlying network. For the Internet, several types of coordinate have been suggested (e.g. Euclidean, Spherical, Hyperbolic). Although none of these model the Internet's topology precisely, several studies have shown that good estimates are achievable using coordinates with few dimensions [LPS06].

We generate virtual coordinates for the brokers in our overlay network using the Vivaldi algorithm [DCKM04]. Vivaldi is a scalable distributed algorithm based on the concept of *spring*

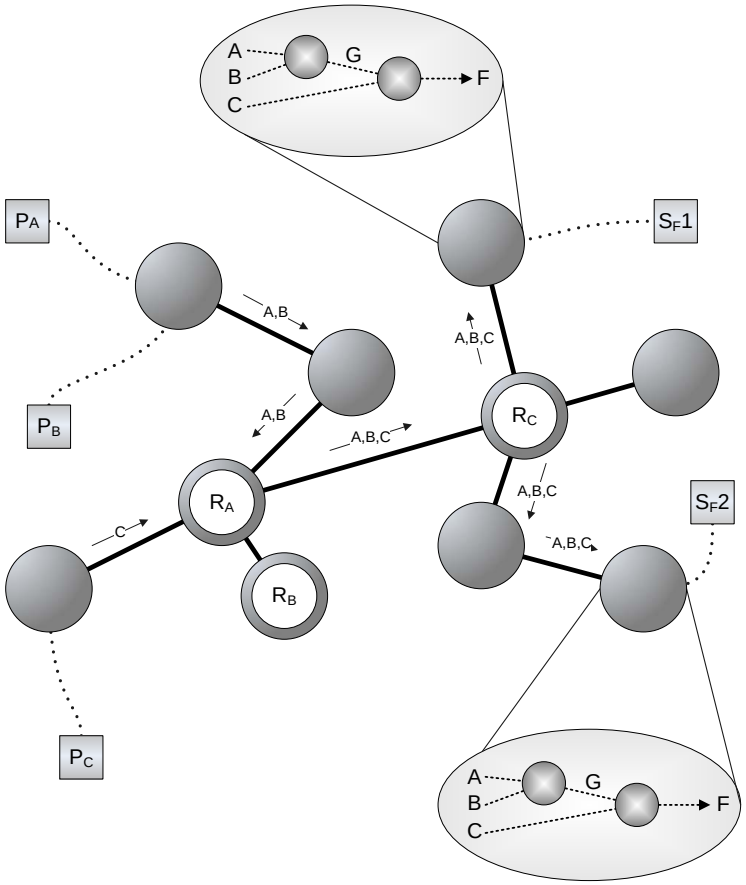

Figure 5.3: Subscriber Hosting Broker (SHB) Placement

relaxation. Vivaldi models the broker network as a system of springs, where tension in a spring connecting two brokers corresponds to a difference in the true latency between them and that predicted by the latency space. Each broker periodically measures the distance to a neighbouring broker (e.g. using ping), or to one of a small number of randomly selected faraway brokers. Brokers then refine their coordinate to minimize the difference between measurements and the distance predicted by coordinates. By iteratively refining broker coordinates in a decentralized fashion, Vivaldi attempts to find a minimum energy state for the springs, minimizing the prediction error of coordinates in the process.

Thus in addition to their Pastry identifier, each broker in our overlay network is equipped with a virtual coordinate generated using Vivaldi. This enables a broker to compute an estimate of the latency between it and any other point in the latency space (e.g. another broker). Note however that latency measurements (and thus coordinates) reflect the time taken to send a message directly between brokers (i.e. using IP addresses), and not the time taken to send a message via the overlay network.

5.5.2 Query Graph Placement

As described in Section 5.3, we convert the complex event expressions created by subscribers into a query graph. The source leaf nodes of this directed acyclic graph are the publishers of primitive events, the internal nodes are detectors of complex events, and the sink leaf nodes are external subscribers to complex events. Primitive publishers and external subscribers have a fixed location in the network, where by fixed we mean that we cannot exert control over their location. The task of our dynamic placement strategy is to map the internal nodes of the query graph (i.e. detectors) to nodes in the broker network. In this section we describe how detectors already existing in the network can use a latency space to optimize their location relative to each other. Mechanisms for performing the initial mapping of new detectors are discussed later in Section 5.5.4.

To optimize the placement of detectors, we adopt a solution based on spring relaxation (e.g. as done in [BB03] and [PLS+06]). We assume each detector knows the coordinate of its local broker and the coordinates of brokers hosting its parents and children in the query graph. This information can be attached to event publications for example, or to subscription refreshes in the case of parents informing their children. The latter may also include information about observed data rates.

Detectors are pulled in various directions by their query graph neighbours. The force exerted by each neighbour is proportional to the difference between the measured network distance and the distance predicted by the latency space, weighted to take into account other factors such as expected data rates. However, instead of spring relaxation resulting from changes to the coordinates of brokers (as with the Vivaldi algorithm for generating broker coordinates), springs are relaxed by the migration of detectors to other brokers in order to minimize the force being exerted by their query graph neighbours (Figure 5.4).

Thus each detector periodically calculates the net force being exerted on it by its neighbours.

It then adds this force to the coordinate of its host broker to get its optimal location. If a broker with a coordinate closer to the optimal location can be found[1], the detector migrates to that broker. Otherwise it stays where it is.

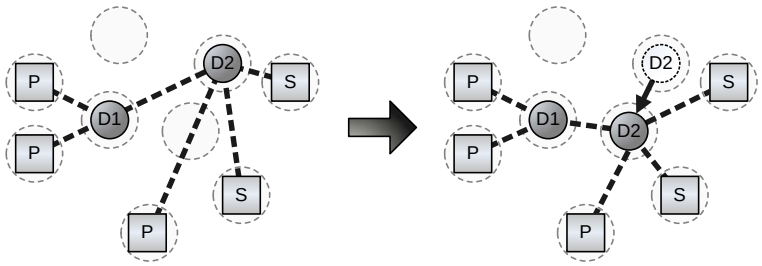

Figure 5.4: Spring Relaxation of a Complex Event Query Graph

5.5.3 Placement with Reverse Path Routing

Unlike some previous dynamic solutions for query graph placement, we do not route events between detectors directly over the underlying physical network. Instead we forward events using the type- and attribute-based routing algorithm of Hermes. This makes it easy to set up efficient event dissemination trees when there are large numbers of subscribers to a detector, and reduces the chances of the output bandwidth of a detector's broker becoming saturated due to high fan-out. However, type- and attribute-based routing forwards events through the Pastry distributed hash table along the reverse path of subscriptions. Does a placement that minimizes the distance between detectors in the physical network correspond to a good placement when reverse path forwarding through an overlay network is used? We claim that it does due to the *locality* properties of Pastry [CDHR03]. We now describe the intuition behind our claim.

The Pastry DHT has the property that routes to the same destination from nearby sources converge quickly on average, a property we refer to as *local route convergence*. This is because at each consecutive routing step, messages travel exponentially larger distances towards an exponentially shrinking set of nodes. Thus the probability of routes converging increases after every step, even in the case where earlier hops caused messages on each route to move further away. Early hops tend to move a small distance in the underlying network's proximity space, but large distances in the Pastry node ID space.

The type- and attribute-based routing algorithm of Hermes takes advantage of this property by forwarding subscriptions along the reverse path of any matching advertisements. The algorithm routes subscriptions and advertisements to the same destination (i.e. the rendezvous node). Therefore their paths will converge quickly if the senders (i.e. publishers and subscribers) are near to each other in the physical network. Since an event produced by a publisher follows the reverse path of subscriptions, the notification delay experienced by a subscriber is in turn

[1]Mechanisms for finding such a broker are discussed in Section 5.5.4.2.

likely to be small if it is located close to the publisher.

For a complex event expression, reverse path forwarding is performed repeatedly at each level of the query graph. At the bottom level, primitive publishers reverse path forward their events to subscribers, some of whom may be detectors. These detectors in turn publish events that they reverse path forward towards their own subscribers, some of whom may again be detectors. This process is repeated until the top level of the graph is reached. The efficiency of event dissemination increases at a particular level when publishers are located close to subscribers in the network latency space. Our dynamic placement strategy uses spring relaxation to minimize the distance between detectors at all levels of the query graph. Therefore due to the locality properties of Pastry spring relaxation should give a good placement when type- and attribute-based forwarding is used.

Figure 5.5 illustrates HNC placement for the complex events `F` and `G` we used earlier to describe RV and SHB placement. HNC placement locates the detector for `G` at the publisher hosting broker of $\texttt{P}_A$ and $\texttt{P}_B$. It then places the detector for `F` at a nearby broker that happens to be the rendezvous of type `A`, roughly halfway between the detector for `G` and the only publisher for `C`, i.e. $\texttt{P}_C$. Assuming that the output rate of each detector is lower than its input rates, this should be an efficient placement. If the locations of publishers or subscribers change, HNC can adapt by moving the detectors to a better location. Note that in this example, cross domain complex event detection may be possible for detector `G`, since both of its input publishers are on the same broker, and no other subscribers have overlapping subscriptions.

5.5.4 Initial Placement Mechanisms

We have described how detectors that already exist in the network can use network coordinates to optimize their placement. However, we have not discussed how to create and place them initially. In this section we give two mechanisms for initially placing detectors over Hermes as part of our dynamic placement strategy. The first approach instantiates detectors at rendezvous nodes. Detectors then gradually migrate to better locations as described earlier. The second approach gathers information about primitive publishers and existing detectors centrally. It then calculates locally a good starting position for each new detector. This approach avoids having to perform long distance migrations to correct a bad initial placement.

5.5.4.1 Rendezvous Placement with Gradual Migration

A simple way to map a query graph onto the broker network is to place detectors initially at the rendezvous broker for their type, in a similar fashion to the static rendezvous placement strategy. Once setup is complete, events start to flow through the detectors. Each detector can now determine a better placement locally using the coordinates of its query graph neighbours and the data rates it observes, as described in previous sections.

Initial placement at rendezvous nodes requires little extra effort to route a complex subscription in comparison to a primitive subscription, and is the approach we implemented for our experiments in Section 5.7. However, since rendezvous nodes are distributed randomly, detector

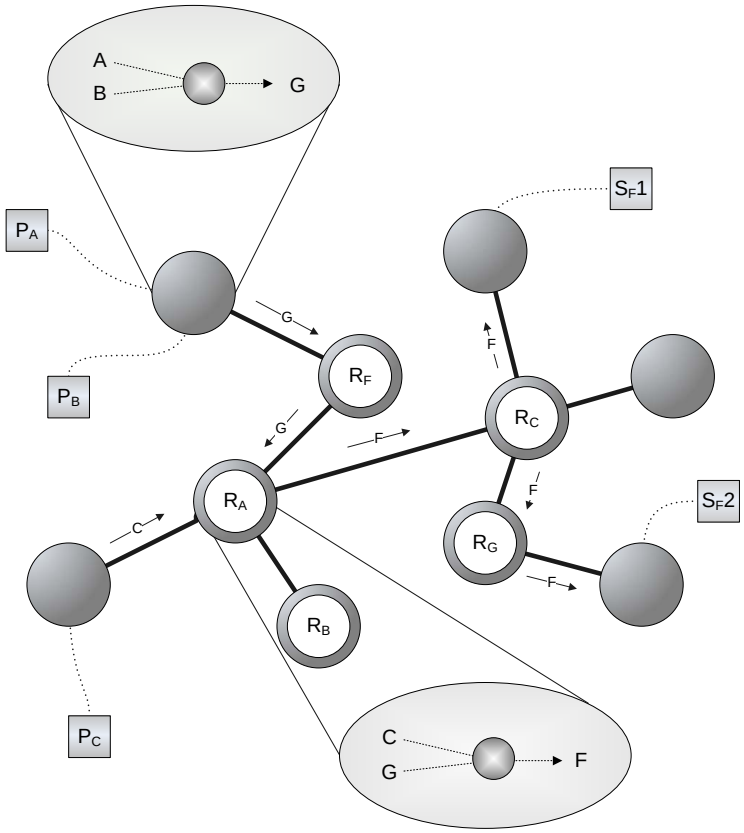

Figure 5.5: Hermes Network Coordinate (HNC) Placement

migration may take a non-negligible amount of time to converge to a good placement if no existing detectors can be reused. In addition, long distance detector migrations to overcome bad initial placements can be expensive.

5.5.4.2 Centralized Initial Placement Calculation

As an alternative approach to initial placement of detectors, we can firstly gather coordinates of primitive publishers and existing detectors at a single broker. We can then calculate centrally a good initial placement for each new detector, and create them at the computed locations. Only then does we set up advertisements and subscriptions between detectors, allowing us to avoid long-distance migrations. This approach is similar to that taken by previous work [BB03; PLS+06]. However, it involves several stages, as we describe next.

As a first step, the subscriber hosting broker determines whether each detector in the complex event expression already exists in the network. It does this by routing a lookup message towards the rendezvous broker for the top level detector's type in the complex event query graph. When a detector already exists, the lookup encounters an advertisement for it en route to the rendezvous. The lookup message then follows the reverse path of this advertisement to the detector. In response, the detector sends a reply message containing its network coordinate. Since the detector already exists, there is no need to look up the location of its sub-detectors or primitive type inputs. If the look up does not encounter an advertisement, the rendezvous broker informs the subscriber hosting broker. The subscriber hosting broker then sends further lookup messages to the rendezvous brokers of the complex event's sub-events. If there are multiple publishers matching a lookup (as will typically be the case for a primitive event input of a detector), then either each publisher responds with its coordinate, or the rendezvous responds with its own coordinate as an approximation.

Once the relevant coordinates have been gathered, we use a centralized version of the distributed spring relaxation algorithm described earlier to calculate placements for any detectors that don't already exist. We then place these detectors in the network at the broker closest to the placement calculated for them. This requires a mechanism for finding the nearest broker to a particular coordinate (e.g. the space filling curves used by Relaxation [PLS+06], or scaled-θ routing [HP01; LPMS07]).

Once they have been placed, detectors send advertisements to their own rendezvous broker, and subscriptions to their input type rendezvous brokers. Events then flow through the query graph to subscribers. Finally, detectors monitor the locations of their query graph neighbours as before, and migrate when a better placement can be found.

The mechanisms used by our centralized initial placement algorithm are more complex than those for simply starting detectors at rendezvous brokers. However, they make sense if it is likely that a significant portion of the detectors in a complex event expression do not already exist in the network. Migrating a detector between two brokers is effectively the same as performing a handover of a publisher and multiple subscribers between two brokers, so it is better to reduce the number of long distance migrations required if possible.

5.6 Other Issues

Having described the basic operation of our placement strategies, we now discuss several further issues that must be addressed by our distributed complex event detection service.

5.6.1 Reliability

For scalability reasons, Hermes relies on a *soft-state* reliability model. Under this model, brokers send advertisement and subscription refreshes between themselves periodically. When a failure is detected by a broker, it routes advertisements and subscriptions around it. This model allows for fast, localized recovery from overlay failures. However it also means that these failures may cause gaps in event streams. As discussed in Chapter 4, some complex event expressions are more tolerant to lost input events than others. The strategies for detector placement over Hermes described in this chapter are therefore best suited to applications for which such expressions are common.

Complex event detection must also deal with detector failure due to broker crashes. We can handle this kind of detector failure in various ways, depending on the placement strategy being used, whether brokers can recover from failure, and the reliability requirements of subscribers. For SHB placement, detectors are hosted on the subscriber hosting broker. A simple mechanism for handling detector failure with this strategy is for the subscriber to connect to multiple brokers. However detectors will still have to tolerate lost input events. In addition, if detectors receive different input streams (e.g. an event is delivered to one but not to another), the events they publish may differ.

Similar problems arise for RV and HNC placement. However for our experiments, we are primarily interested in the performance of each strategy in the absence of failures. We therefore adopt a best effort approach whereby one or more failure monitors for each detector are created on another broker. When a detector failure is observed, one of the monitors restarts it on a new broker.

For RV placement, failure of a detector implies failure of the rendezvous node for its type. Therefore, we place failure monitors on rendezvous replicas, with the monitor on the new rendezvous broker responsible for restarting the detector. For HNC placement, failure monitors are located on neighbours of the detector's broker. When the detector's broker fails, a failure monitor restarts it on a neighbouring broker as before. For both strategies, all state of the old detector is lost using this mechanism, meaning it is only suitable when applications are satisfied with best effort detection or broker failure is rare.

Replication of detectors is also possible for RV and HNC placement, although we have not evaluated its effect on network usage in our experiments. Replication can help to reduce event loss as a result of detector failure. However, as with SHB placement, replicas must deal with lost input events. These can result in subscribers receiving duplicate or conflicting events when a replica takes over from a failed detector. For RV placement, we place passive detector replicas at the replicas of the rendezvous broker. They subscribe to the output events of the primary

detector as well as the same input events. Backup replicas do not create any advertisements or begin to publish until failure of the primary detector occurs. At this point, one of them advertises and starts publishing, continuing at the last event received from the failed primary detector.

A similar replication strategy may be used for HNC placement, with replicas located at a neighbour of the detector's broker. However, replicas must also coordinate with the primary broker if cross-domain complex event detection is required. Otherwise, opportunities to perform cross-domain detection may be ignored as the domain gateway cannot distinguish between a replica and a normal subscriber on another broker. This coordination is relatively straightforward and we do not discuss it further here.

5.6.2 Types

Distributed complex event detection raises several issues with regard to type handling.

5.6.2.1 Type Directories

In an open environment, types may be created by different clients. They simplify application development; clients may wish to reuse the types created by each other. To address this problem, we assume that a directory is available listing the currently active types in the system. Such a directory also increases reuse of detectors when renaming of complex events is possible. Our directory keeps weakly consistent with the middleware by subscribing to a special event topic that notifies it when a new type is defined, modified, or removed. Several such directories may be required at larger scales.

5.6.2.2 Type Checking

When a new complex event subscription is created by a subscriber, its validity must be checked by the subscriber hosting broker before being forwarded. For many complex event languages, the name of the complex event is given by the expression used to define it. If this is the case, the rendezvous broker of the complex event type is given by the hash of the complex event expression. Type checking of the complex event involves type checking each of its constituent primitive events.

Some languages, such as the complex event detection language described in Chapter 3, allow complex event definitions to be explicitly named. This simplifies programming somewhat by enabling a programmer to refer to existing complex events by name. Explicit naming makes most sense when a language allows a variety of different options to be specified in a complex event definition, such as consumption policies or override conditions (e.g. as for the language in Chapter 3, or the AMIT situation detection language [AE04]), as definitions can become quite verbose.

However, languages with explicit naming of complex events do have some implications for type checking. In particular, they require brokers to type check constituent complex events in

addition to primitive events. To do this, a broker must have knowledge of all sub-events of a complex event (e.g. to detect cycles in the event graph). These may be retrieved as usual from the broker's cache, or from the relevant rendezvous broker. In some cases the input events to the complex event being defined by a subscriber may themselves be complex events. We store the whole definition of a complex event (including all sub-events) in caches and at rendezvous brokers. This prevents having to recursively look up all sub-event definitions.

5.6.2.3 Schema Evolution

An interesting question is what should happen if the definition of a primitive event type that is a component event of a complex event is modified? In some cases the complex event may not be affected, but in others it may become inconsistent. For example, if the complex event definition contains a filter over an event attribute that is removed, the complex event type will no longer make sense. Each detector must therefore be notified of changes to the definitions of any of its component types (e.g. via its rendezvous using the *meta-events* proposed for notifying brokers of type definition changes in Hermes [Pie04]). If a change makes its complex event inconsistent (or if the creator of the complex event specifies that any change to sub-event definitions should cause a failure), then a detector should terminate, perhaps sending a failure message.

For languages with explicit naming of complex events, evolution of internal complex events can occur in addition to evolution of primitive event definitions. However, the mechanisms required to support this are the same as for languages without explicit naming.

5.7 Evaluation

In order to evaluate our placement strategies, we compare their performance to a dynamic strategy from the literature called Relaxation Placement (RELAX) [PLS+06]. Like our dynamic strategy, RELAX optimizes detector placements based on information provided by a network coordinate layer. However, all event routing is done directly between detectors without the use of a publish/subscribe layer. RELAX thus serves as a reference placement strategy. It allows us in particular to determine the overhead of our strategies when there is little reuse of events between subscribers.

The metric we use to compare the performance of each strategy is *network usage* [PLS+06]. This enables us to understand the relative cost of each strategy in terms of both event delivery latency and the bandwidth used for event dissemination. For pervasive computing applications, we would like to maximize our ability to perform cross-domain complex event detection. Since this is usually impossible for our static placement strategies, our experiments let us estimate the burden imposed on the central domain by dynamic placement, if any, in order to exploit such opportunities.

We run six sets of experiments in total. Our first two sets of experiments are based on a complex event query graph where types are only decomposed, and not partitioned or split (CQG1 in Figure 5.6). The next two sets of experiments use a complex event query graph containing

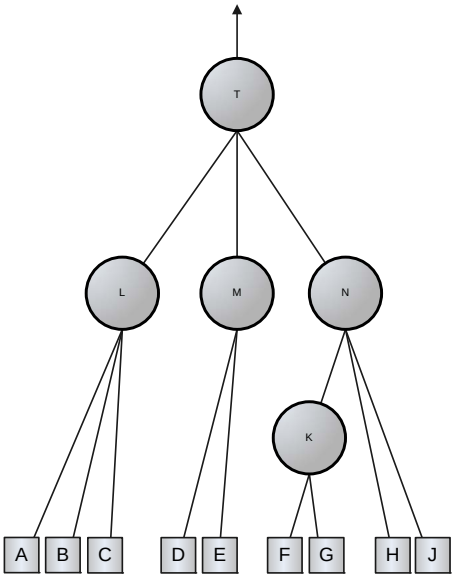

Figure 5.6: Complex Event Query Graph 1 (CQG1)

expressions that are both decomposable and partitionable (CQG2 in Figure 5.7). Our final two sets of experiments also use a query graph that is both decomposable and partitionable (i.e. CQG2). However, we firstly divide the network itself into several regions. We then cluster the publishers and subscribers of a query partition into the same region of the network. Note that from experiment 3 on, we reduced the maximum number of subscribers to a detector from 20 to 10 (see 5.2). This was forced on us for reasons of simulation scalability, as CQG2 contains many more detectors than CQG1. However, we don't believe it would change the conclusions we draw from our experiments.

The number of subscribers interested in each of the primitive and complex events in a com-

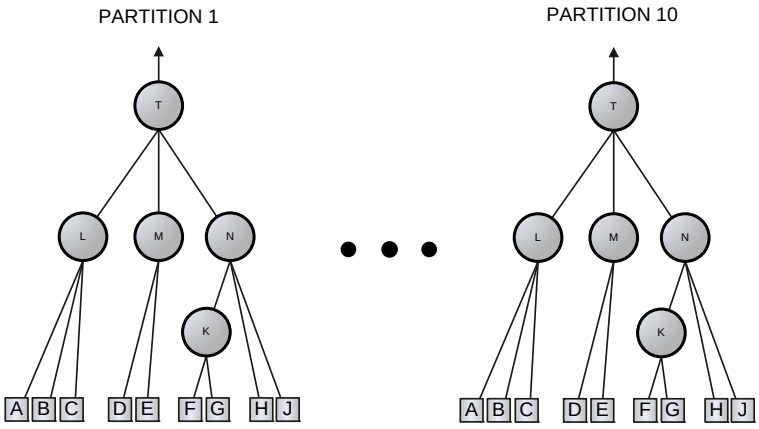

Figure 5.7: Complex Event Query Graph 2 (CQG 2)

plex event query graph can have a major impact on the performance of a particular strategy. Therefore for each set of experiments, we evaluate network usage for several *reuse patterns* of

the query. An example of a reuse pattern is a complex event where multiple subscribers are interested in the high level complex event itself, but very few are interested in the primitive events used to detect it. Some reuse patterns can reduce our ability to perform cross-domain complex event detection. A good dynamic placement strategy for these patterns should avoid imposing a large network usage penalty since the chances of cross-domain detection counterbalancing it are low.

In the following discussion of our experiments, RV refers to the strategy of placing detectors statically at rendezvous brokers, SHB refers to static placement at subscriber hosting brokers, and HNC to dynamic placement of detectors over Hermes with network coordinates. All our experiments were performed via simulations on top of Pietzuch's Distributed Systems SIMulator (DSSIM) [Pie04]. We implemented SHB, RV, and HNC placement as a complex event service over an existing implementation of Hermes [Pie04]. Each experiment took place over a transit stub topology generated using BRITE [MLMB01]. The topology contained 1000 physical nodes in total, divided into 10 autonomous systems (AS) containing 100 nodes each. We created a broker overlay network containing 500 brokers on top of this topology.

Finally, we note that our experiments measure only the network usage of event publications. We assume that in comparison other sources of network traffic, such as the overhead of maintaining a latency space, or differences in the cost of setting up advertisements and subscriptions, result in negligible differences in network usage between strategies.

Parameter	Description	Value
N_N	number of network nodes	1000
N_{AS}	number of ASs	10
N_{as}	number of nodes per AS	100
k	max hops for existing detector search in RELAX	3
b_{PAN}	base for PAN nodeIDs	4
l_{PAN}	leaf set size in PAN	4
ϵ	median relative error of network coords	20.5...23.1
n_E	number of event brokers	500
n_P	number of primitive event publishers per type partition	0...20
n_S^p	number of primitive event subscribers per type partition	0...20
n_S^i	number of internal complex event subscribers per type partition	0...20
n_S^{tl}	number of top level complex event subscribers per type partition	1...20
c	number of clusters	1...8
s	selectivity of detectors	0.5
i	number of experiment runs	3
q	query name	CQG1, CQG2

Table 5.1: Global Experiment Parameter Values

Experiment 1 Our first experiments compare the performance of each strategy for CQG1 (Figure 5.6) when there are multiple external (i.e. not detectors) subscribers to every primitive type. The first variation of this experiment (Exp = 1.i) assumes very little reuse of complex

Exp		n_P	n_{SP}	n_{SI}	n_{STL}	c	q
1	.i	20	20	0	1	1	CQG1
	.ii	20	20	20	20	1	CQG1
2	.i	20	0	0	1	1	CQG1
	.ii	20	0	20	20	1	CQG1
	.iii	20	0	0	20	1	CQG1
3	.i	10	10	0	1	1	CQG2
	.ii	10	10	10	10	1	CQG2
4	.i	10	0	0	1	1	CQG2
	.ii	10	0	10	10	1	CQG2
	.iii	10	0	0	10	1	CQG2
5	.i	10	10	0	1	8	CQG2
	.ii	10	10	10	10	8	CQG2
6	.i	10	0	0	1	8	CQG2
	.ii	10	0	10	10	8	CQG2
	.iii	10	0	0	10	8	CQG2

Table 5.2: Individual Experiment Parameter Values

event types (in fact there is just a single subscriber to CQG1). This scenario could arise for example when most primitive events are directly meaningful to subscribers, with complex event expressions tending to be of specialized interest. The second variation (Exp = 1.ii) assumes multiple external subscribers to CQG1, as well as to internal types of CQG1. This scenario arises when all events, both primitive and complex, are of interest to a large number of subscribers.

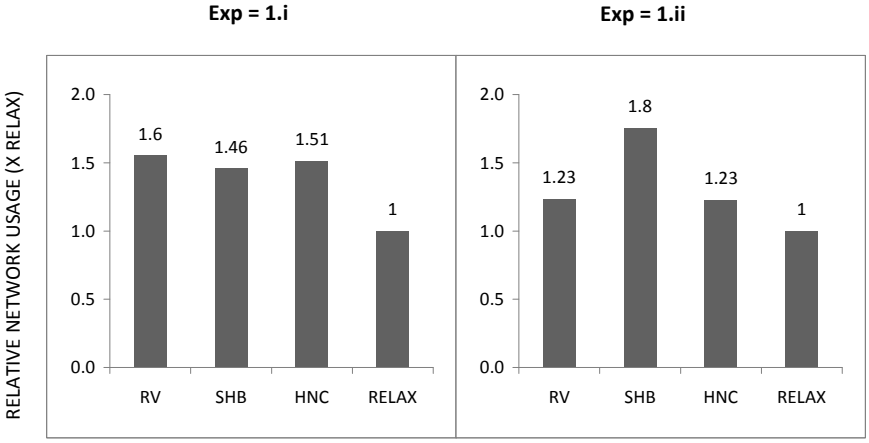

Figure 5.8: Experiment 1

The graphs in Figure 5.8 give network usage for each strategy as a multiplier of the network usage for RELAX placement. For both experiments, RELAX has the lowest network usage. When there is little reuse of complex events (Exp = 1.i), SHB placement has slightly better performance than RV and HNC placement. When there are multiple subscribers to complex events, RV and HNC placement perform better than SHB placement. In both situations, RV

and HNC placement have roughly the same network usage.

Experiment 2 Our second set of experiments compare the performance of each strategy for CQG1 when there are no external subscribers to primitive types. This scenario can be common in pervasive computing when primitive types represent raw data that requires further processing to be of interest to users.

We examined three variations of this experiment. In the first variation (Exp = 2.i), there is a single external subscriber to the complex event CQG1. In this scenario, both primitive and complex events are of specialized interest. Our second variation (Exp = 2.ii), has multiple external subscribers to CQG1, and also to the internal types of CQG1. This models a situation where once primitive events have been interpreted to a higher semantic level, a large number of users find them of interest. Our third and final variation of this experiment has multiple external subscribers to CQG1, but none to its internal types. This models a situation where a complex event is of interest to multiple users, but its internal types are only useful insofar as they generate the top level type.

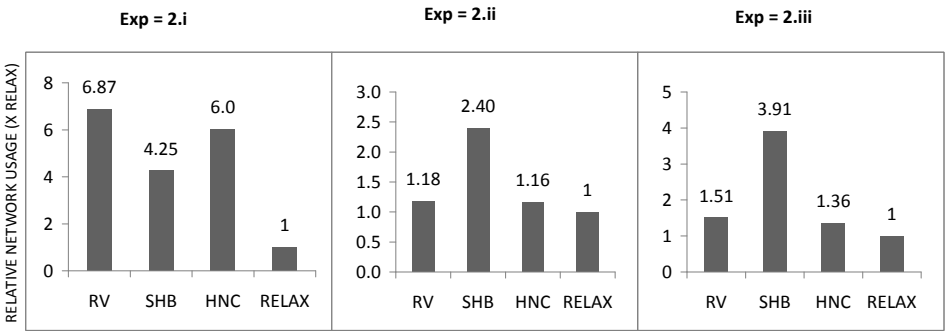

Figure 5.9: Experiment 2

For all three experiments, RELAX has the lowest network usage. When there is little reuse of complex events (Exp = 2.i), SHB placement has better performance than RV and HNC placement. When there are multiple subscribers to all complex events (Exp = 2.ii), RV and HNC placement perform better than SHB placement. The performance penalty of SHB relative to RV and HNC is worse when only the top level complex type is reused (Exp = 2.iii). In all situations, RV and HNC placement have roughly the same network usage.

Experiment 3 Our third set of experiments are a mirror of those performed in the first set, except that they are performed with respect to the partitionable complex event CQG2 (Figure 5.7). Thus CQG2 consists of several different complex event queries (CQG2.1,...,CQG2.n). The primitive events used by each partition are disjoint. Partitioned complex events can be quite common in pervasive computing (e.g. partitioning by geographic location or object identity).

As shown by Figure 5.10, RELAX again has the lowest network usage for both experiments. SHB placement performs better than RV and HNC placement in the first variation where there

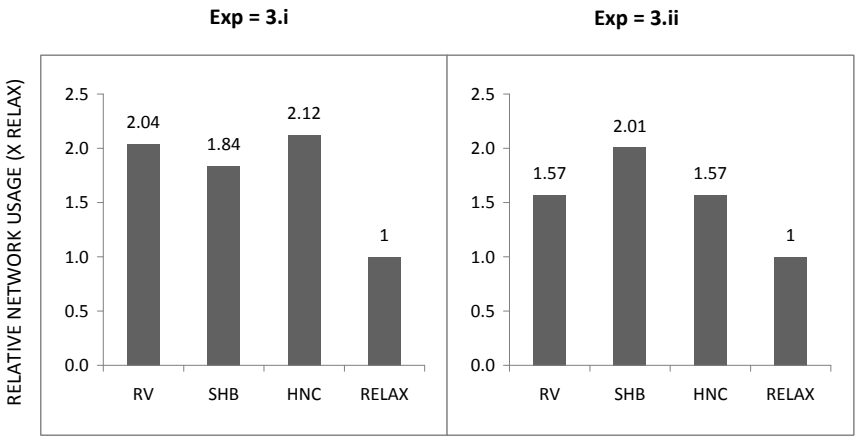

Figure 5.10: Experiment 3

is little reuse of complex events (Exp = 3.i). Conversely, RV and HNC placement perform better than SHB placement for the second variation where there is reuse of complex events (Exp = 3.ii). RV and HNC placement again have approximately the same network usage. The results thus follow the same pattern as Experiment 1.

Experiment 4 In the same way that Experiment 3 corresponds to a partitioned version of Experiment 1, our fourth set of experiments are a partitioned version of Experiment 2.

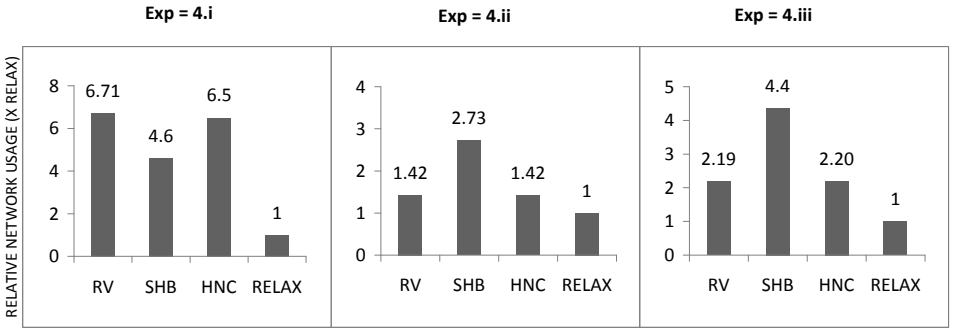

Figure 5.11: Experiment 4

Once again, RELAX has the lowest network usage across the board. SHB placement performs poorly for all three variations. When reuse of complex events is low (Exp = 4.i), SHB placement performs better than RV and HNC placement, but they all perform poorly in comparison to RELAX. However, when there is reuse of every complex event type (Exp = 4.ii), or even just of the top level complex type (Exp = 4.iii), RV and HNC placement perform better than SHB, and the performance gap with RELAX is lower. RV and HNC placement again have approximately the same network usage for all three variations.

Experiment 5 For our fifth set of experiments, we divide the network into 8 clusters based on network latency. We assign each partition of complex query CQG2 to one of these clusters. This models a scenario where publishers and subscribers to the same query partition are located close to each other. We believe such a distribution may be common for geographic partitions. Apart from clustering, the experiment setup is similar to that of Experiment 1 and Experiment 3, in that there are multiple subscribers to each primitive type. Our first variation (Exp = 5.i) models a scenario where there is no reuse of complex events, whereas our second variation (Exp = 5.ii) examines the case where there are multiple subscribers to each complex event partition.

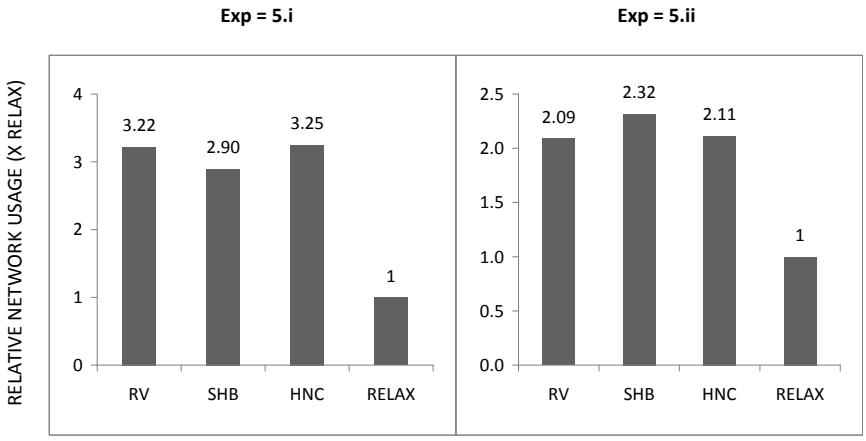

Figure 5.12: Experiment 5

Similarly to all our previous experiments, RELAX placement has the lowest network usage for both variations of Experiment 5. In the first variation of this experiment (Exp = 5.i), there is no reuse of complex events, and SHB placement performs better than RV and HNC placement. However for the second variation (Exp = 5.ii), SHB placement is outperformed by RV and HNC placement. For both variations, RV and HNC placement have approximately the same network usage as each other. The performance of RV, HNC, and SHB placement improves with respect to RELAX when there are multiple subscribers to complex types (Exp = 5.ii).

Experiment 6 Our sixth and final set of experiments is also over a pre-clustered network. Similarly to Experiment 5, we assign each partition of CQG2 to one of these clusters. However, the reuse patterns in this experiment correspond to those used for Experiment 4, in that there is no reuse of primitive event types. The first variation tests the query with no complex event reuse, the second with reuse of every complex event, and the third with reuse of the top level complex event only.

Once again RELAX placement is the best performing strategy. RV and HNC placement again have approximately the same network usage as each other. They perform best when there is reuse of complex events (Exp = 6.ii and Exp = 6.iii). However they perform poorly when there is no complex event reuse (Exp = 6.i). Once again, RV and HNC placement perform better

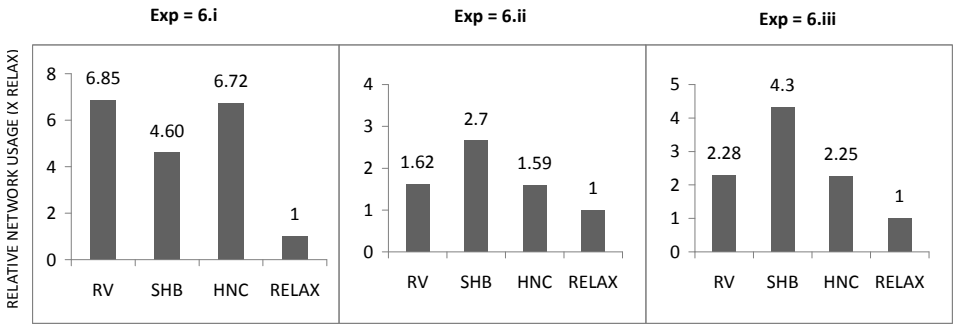

Figure 5.13: Experiment 6

than SHB placement when complex events are reused, and SHB performs better when they are not.

5.7.1 Summary of Experiments

Several trends are apparent from our experiments.

RELAX vs RV and HNC Firstly, Relaxation placement has the lowest network usage out of all the strategies. For RV and HNC, the biggest performance penalties occur for experiments 2.i (6.87 and 6.0), 4.i (6.71 and 6.5), and 6.i (6.85 and 6.72). This is to be expected, since these are the queries with minimal reuse of both primitive and complex event routing paths. Sending events directly between producers and consumers is therefore the most efficient option for these queries.

The performance penalty is generally much less for queries with reuse of primitive event routing paths (1.i, 3.i and 5.i), reuse of complex event routing paths (2.ii, 4.ii, and 6.ii), and reuse for both primitive and complex routing paths (1.ii, 3.ii, and 5.ii). Cross-domain complex event detection is usually only possible for the second category.

For the first category, the (RV, HNC) penalty is (1.6, 1.51) for 1.i, (2.04, 2.12) for 3.i, and (3.22, 3.25) for 5.i. We expect the increase in performance penalty for 5.i is due to the clustering of publishers and subscribers. This reduces the benefits of path sharing, since paths are typically shorter. We expect the reason 3.i performs worse than 1.i is that we used less publishers and subscribers for the latter due to simulation scalability constraints ($n_P = n_{SP} = 10$ for 3.i against 20 for 1.i, see Table 5.2).

For the second category, the (RV, HNC) penalty is (1.18, 1.16) for 2.ii, (1.42, 1.42) for 4.ii, and (1.62, 1.59) for 6.ii. Again, RV and HNC placement perform slightly worse over the clustered network (6.ii) than the unclustered network (4.ii).

For the third category, the (RV, HNC) penalty is (1.23, 1.23) for 1.ii, (1.57, 1.57) for 3.ii, and (2.09, 2.11) for 5.ii. Similarly to the first category, clustering affects network usage adversely. Thus when there is reuse of primitive type routing paths, the benefits of path sharing are reduced by clustering.

A fourth category we tested (2.iii, 4.iii, and 6.iii) models a situation where the only reuse is of the top level complex type. Cross-domain detection of these queries may be possible in many cases. The (RV, HNC) penalties were (1.51, 1.36) for 2.iii, (2.19, 2.20) for 4.iii, and (2.28, 2.25) for 6.iii. As we expected, the lack of reuse of primitive or internal type routing paths results in slightly worse performance relative to RELAX than the previous three categories.

SHB vs RELAX SHB placement performs worse than RELAX placement for all experiments. Its performance follows a similar pattern across the variations of experiments 2, 4, and 6. Performance is worst when there is no reuse of primitive or complex events (2.i, 4.i, and 6.i). Performance is slightly better when there is reuse of top level complex events only (2.iii, 4.iii, and 6.iii). Performance is at its best when there is reuse of internal and top level complex events (2.ii, 4.ii, and 6.ii). This pattern was as we expected, since the opportunities for reusing primitive event dissemination paths increases as the number of subscribers increases.

Experiments 1, 3, and 5, model a situation in which there are lots of subscribers to primitive events, together with varying numbers of subscribers to complex events. For experiment 5, the performance of SHB relative to RELAX is better when there is reuse of primitive and complex events (5.ii) than when there is reuse of primitive events only (5.i). This is in keeping with the trend apparent from experiments 2, 4, and 6. However, for experiments 1 and 3, the performance of SHB relative to RELAX deteriorates slightly as reuse of complex events increases (1.i vs 1.ii and 3.i vs 3.ii). We believe performance deteriorates for experiments 1 and 3 because early filtering of complex events by RELAX becomes more beneficial as the number of complex event subscribers increases. However, in experiment 5, any such improvement is masked by clustering of publishers and subscribers for the same partition, as nodes in a cluster are more likely to have received a primitive event anyway (e.g. due to leaf set overlap) than in an unclustered network.

Finally, we note that the contrast between the deterioration in performance of SHB placement as complex event reuse increases in experiments 1 and 3 and the improvement in performance for experiments 2, 4, 6. We expect the reason for this disparity is less to do with clustering effects and more to do with the lack of primitive event subscribers whose routing paths SHB placement can reuse (in experiments 2, 4, and 6). Overall, SHB placement is a good option when there is a lot of primitive event reuse, little complex event reuse, and no clustering.

SHB vs RV and HNC In comparison to RV and HNC placement, SHB placement performs better when there is no reuse of complex events (1.i, 2.i, 3.i, 4.i, 5.i, and 6.i). When there is reuse of internal or top level complex events, RV and HNC outperform SHB. SHB performs worst relative to RV and HNC when there is reuse of top level complex events only (2.iii, 4.iii, and 6.iii).

These results were as expected, since the benefit of performing complex event detection to filter events earlier should increase as the number of users of a complex event increases. Furthermore, this benefit should be maximised when there is reuse of top level complex events only. This is because the number of subscribers interested in lower level events primitive or

complex events is smaller, and there is less opportunity for SHB placement to piggy back on their event routing paths.

RV vs HNC Unexpectedly, the network usage of RV placement was virtually identical to that of HNC for all of our test queries (excluding the cost of maintaining network coordinates). The best performing scenario for RV with respect to HNC was experiment 3.i, where RV had 96% of the net usage of HNC (2.04 vs 2.12). The best performing scenario for HNC over RV was for experiment 2.i, where HNC had 88% of the net usage of RV (6.0 vs 6.87). Except for perhaps experiment 2.iii, all other experiments had virtually the same network usage.

Given that HNC placement requires us to maintain a network coordinate layer, overall network usage is likely to be slightly higher. This implies that, for our current implementation and with respect to our original motivations, HNC placement should only be used instead of RV placement if the amount of energy saved through cross domain detector placement outweighs the cost of maintaining the network coordinate layer. This result was surprising to us, especially as rendezvous brokers are distributed randomly within the network. HNC placement may have other advantages over RV placement however, especially in situations where the rendezvous node for a type does not have the resources to support a detector. The likelihood of the rendezvous node becoming a bottleneck is especially acute for partitioned complex expressions, since the detector for every partition would be located at the rendezvous.

5.8 Related Work

We have described a complex event detection service for the Hermes type- and attribute-based routing algorithm. There are of course other content based routing algorithms over which event pattern detection could be performed. An example of these is the complex subscription language proposed by Li and Jacobsen as part of the PADRES publish/subscribe middleware [LJ05; LMJ07]. In [LJ05], they show how to integrate the routing and placement of complex event expressions into the PADRES middleware, whose routing algorithm is based on the advertisement routing algorithm of SIENA [CRW01] over an acyclic peer-to-peer topology. Apart from needing to flood advertisements, and the lack of robustness of the acyclic peer-to-peer topology (in comparison to a general peer-to-peer overlay), placement of expressions is determined by the topology of the overlay network, and does not take into account differing publication rates of event producers.

However, this work is extended in [LMJ07] to perform distributed complex event detection over a general peer-to-peer topology. In addition, the ability to place detectors is decoupled from the topology of the broker network. However, the routing algorithm still requires flooding of advertisements. In addition, to prevent cycles in the network, each advertisement predicate is appended with a unique *tree ID*. This can reduce the number of times matching must be performed for an event. However, it further increases state requirements, since it reduces the

algorithms ability to exploit covering relationships between advertisements generated by different publishers. On the other hand, operators in a complex expression tree are placed optimally in the broker network, independently of their height in the tree, since there is no need to route towards a rendezvous node as in Hermes. Thus the reduction in state from using Hermes comes at the cost of a potential increase in detection latency for complex event trees, depending on the height of the tree.

Chen et al. [CRL09] suggest an algorithm for placing aggregation operators in distributed publish/subscribe systems. The algorithm attempts to find the graph center of publishers and subscribers to a single type in a decentralized fashion. However they do not support placement of generalized complex event query graphs, and the placement of aggregation operators does not enable migration in response to changing subscriptions or networks conditions.

In [PLS+06], a decentralized, iterative, spring relaxation algorithm called Relaxation is used to generate a solution to the problem of placing stream operators in a network aware fashion. Our proposed solutions uses latency coordinates and a spring relaxation algorithm in a similar way. However, their solution is designed to support the placement of operator graphs where routing between operators is done directly at the network layer, and does not investigate how this placement can be done over publish/subscribe, or how the placement might be affected if routing is done using a DHT.

Synergy is another stream operator placement solution that attempts to maximize reuse of existing operators when placing a new query plan [RGK06]. Synergy uses a distributed hash table to store the locations of existing streams and operators, and attempts to minimize QoS conflicts between users when sharing query operators. Again however, it does not investigate placement over publish/subscribe.

5.9 Summary

We describe in this chapter the problem of performing efficient complex detection for pervasive computing applications over a large-scale multi-domain publish/subscribe middleware. We highlight the potential of cross-domain complex event detection to help prolong the lifetime of sensor networks by reducing unnecessary communications. We argue for the use of a type- and attribute-based publish/subscribe middleware (Hermes) in the central domain of our architecture, and define static and dynamic detector placement strategies to complement its routing algorithm. We compare the network usage of our strategies to that of a stream operator placement algorithm from the literature called Relaxation placement [PLS+06]. Our results indicate that when reuse of complex event types is high, our dynamic strategy results in network usage of between roughly 1.25 and 2.5 times that of Relaxation. However, our strategy makes it easier to avoid saturating the bandwidth of outgoing links and maximizes reuse of complex event detectors.

Chapter 6

Conclusions and Future Work

Pervasive computing is characterized by omnipresent distributed applications that adapt transparently to changes in the environment. An important facet of pervasive computing applications is their ability to respond to information received from sensors. However, raw sensor data is often uninformative and impersonal. Bridging the semantic gap between sensor data and application-level knowledge is a challenging task for developers.

6.1 Summary

In this dissertation we argue that middleware support for complex event detection can help ameliorate this problem. A complex event detection service enables application developers to specify and detect patterns over incoming sensor data. We define a complex event language for pattern specification that is targeted specifically towards pervasive computing applications. A domain-specific language such as this eases application development by providing abstractions that more closely match developers' model of the domain.

However, the design of a complex event detection service entails more than just deciding what features the expression definition language should provide. In particular, matters are complicated by the distributed nature of pervasive computing applications. We examine how a range of distributed systems issues affect complex event detection. These include the lack of a globally synchronised clock, and the need to cope with message loss and delays in an appropriate fashion. Our language provides a variety of detection policies to enable application-specific control over how these problems are handled by the service.

A distributed complex event detection service also provides a degree of flexibility. In particular, it allows detection of patterns to be performed anywhere in the network. This has several potential benefits, such as offloading computation from lightweight clients to more capable devices and reducing unnecessary communication with energy-constrained sensors. We thus investigate the performance of a variety of detector placement strategies for a distributed complex event detection service in the context of a scalable event-based middleware.

6.2 Conclusions

We began by introducing our complex event detection language for pervasive computing applications. In order to cope with the lack of a global clock in distributed systems, our language bounds the precision of event timestamps using uncertainty intervals. We found uncertainty intervals fairly straightforward to integrate into our detection model, especially for applications that wish to either ignore them or fail immediately on the occurrence of conflicting timestamps. Providing more sophisticated handling of conflicting timestamps can be beneficial too, as our no false positives detection policy shows.

Our language permits composition of complex event expressions, so long as they together form a directed acyclic graph. Hierarchical decomposition of this graph allows for distributed detection. From an implementation perspective, we found this decomposition relatively straightforward, due primarily to the graph's acyclicity and the lack of conditional dependencies between complex event types. Our language also provides constructs to facilitate partitioning of a single complex event expression, such that partitions may be detected separately in a distributed fashion. We found geographic partitions to be particularly useful for pervasive computing given the common need to detect a complex event at each of a number of different locations.

Our language provides a selection of complex event operators, and a variety of parameters for configuring detection. We believe these configuration parameters are necessary given the breadth of pervasive computing applications we consider. Furthermore, through the judicious use of default parameters, much of the complexity they introduce is hidden from programmers for the most common types of expression. We found that combining conditions across operators with operand-granularity control over windowing, selection, and consumption results in a highly expressive complex event language.

Having described our complex event language, we then discussed how a variety of distributed systems issues can impinge on our ability to perform correct and efficient complex event detection. We focused in particular on difficulties arising from events with conflicting uncertainty interval timestamps, and gaps in input event streams due to lost or delayed messages. To deal with these issues, we extended our complex event detection language with an additional clause that allows programmers to specify a variety of detection policies.

Although not a panacea, these detection policies enable complex event detectors to tolerate a variety of errors such that the output they produce is sensible with respect to the semantics required by individual applications. For the policies we have implemented, we found our extension straightforward to use at both the syntactic and semantic level. However, we believe a no false negatives detection policy will likely be more problematic to both implement and use. This is due primarily to the extensions it would require to the event model (e.g. tentative or possibly erroneous events).

Of particular interest is our detection policy that ensures no false positives are received by an application. We analysed the amenability of various types of complex event expression to no false positives detection. As a result of this analysis, we conclude that expressions with windows are typically easier to handle than those where input events are consumed in response to the

detection of a complex event. We evaluated an implementation of such a policy for our complex event detection language, and showed how performance is unaffected during normal operation, but that overhead increases with the number of errors that must be tolerated.

Our final chapter addresses the question of *where* detection of a complex event expression should be performed. We suggest several motivations for performing distributed detection. In particular we emphasized the need for detector placements that reduce unnecessary communication with energy-constrained sensor devices. We addressed the problem of detector placement in the context of large-scale multi-domain pervasive computing architectures, and examined in particular how distributed detection might be performed over a scalable event-based middleware called Hermes [Pie04].

We described several detector placement strategies and divide them into two classes, static and dynamic. Dynamic solutions allow detectors to migrate in response to changing network conditions, but are more complex to implement than static solutions. We evaluated our detector placement strategies with respect to each other and to a reference strategy from the literature.

Our results show that the inefficiency of detector placement over Hermes in comparison to the reference strategy depends on the level of reuse of complex event expressions. In addition, they indicate that our dynamic placement strategy allows us to exploit cross-domain detector placements using a similar amount of network resources as the best of our static placement strategies. The ability to push detectors into adjoining domains is important when those domains contain energy-constrained sensor networks, since it can reduce unnecessary communication and hence prolong their effective lifetime. However, we found that the decision to use publish/subscribe instead of direct routing for communication between detectors generally only pays off when there are enough subscribers to share the dissemination overhead. In cases where it is known a priori that the output of each detector will only be of interest to a very small number of clients, direct routing may be more advisable. In either case, we found the flexibility a dynamic placement strategy affords to be extremely useful.

6.3 Future Work

We have highlighted and addressed a variety of problems that arise when attempting to interpret and convey sensor information to pervasive computing applications. However, several open questions and research challenges remain.

Privacy-Aware Complex Event Detection As mentioned in Chapter 5, one of the advantages of a complex event detection service is its ability to provide high-level information to applications but still withhold privacy sensitive raw data. Expressions written in a dedicated complex event language are usually more amenable to analysis than arbitrary code. This may allow a complex event service to automatically determine whether a complex event expression with sensitive input events obeys privacy constraints. Although in theory this might seem straightforward, in practice it may be possible to correlate high-level information with other events or external data in order to guess the obscured data. A comprehensive framework

to assist decision making in terms of which sensitive data to make available for complex event processing is a challenging goal.

Learning and Evolution of Complex Event Expressions We have assumed that users' initial complex event subscriptions reflect the required mapping from raw sensor data to application-level knowledge. However, some parts of a complex event expression may need to change at runtime. In particular, support for a feedback mechanism that enables subscribers to fine-tune certain aspects of an expression may be useful (e.g. the confidence level associated with the occurrence of an event). Currently, all learning must be performed prior to deployment. Thus an interesting research question is what kind of support for learning and evolution of complex events should be provided.

Placement in Heterogeneous Broker Networks In our discussion of placement strategies for complex event detectors, we optimized network usage for an overlay where brokers in the central domain were assumed to be essentially homogeneous in terms of resources. Network usage optimization becomes more difficult when broker resources, such as bandwidth, processing, and storage, vary considerably. Furthermore, if brokers are owned by different parties, a mechanism for rewarding resource contributions may be necessary. When brokers can set their own prices for resources, variations in pricing may further complicate placement decisions. Whether a better mechanism can be found than local search for a broker with spare capacity is an interesting research question.

Sensor Coverage We have suggested detection policies to handle a variety of errors that affect distributed complex event detection. However, the guarantees we provide are only with respect to publishers active in the system. Some applications may require meta-information about what portion of their subscriptions are covered by publishers in order to ensure sufficient sensor coverage. In addition, redundant coverage may be required in some cases for increased reliability. Furthermore, when several publishers produce redundant information, they may wish to coordinate in order to save resources. Providing scalable middleware support for sensor coverage guarantees and publisher coordination is challenging.

Appendix A

Language Definitions

Below, we give a full definition of the *ce-data* data structure into which a complex event expression is parsed.

Complex Event Data Structure

ce-data = (*name*, *ctxt*, *ep*, *map-expr*, *gkeys*, *lkeys*, *gps*) ∈ *CE-DATA*
name ∈ *STRING*

Detection Contexts

ctxt = (*inits*, *terms*)
inits =< *init*+ >
init = *start* | *event-init* `-- Default` = *start*
start ∈ *TIME-STAMP*
event-init = (*type*, *as*, *threshold*, *icc*)
type ∈ *STRING*
as ∈ *STRING*
threshold = *predicate*
icc ∈ “*add*”, “*ignore*”} `-- Default` = *ignore*
terms =< *term*∗ >
term = *expiry* | *event-term*
expiry ∈ *TIME-INTERVAL*
event-term = (*type*, *as*, *threshold*, *cross-threshold*, *tcc*)
cross-threshold = *agg-predicate*
tcc ∈ {“*new*”, “*old*”, “*each*”} `-- Default` = *each*

Event Patterns

ep = (*op-name*, *operand-defs*, *detect-mode*, *params*)
op-name ∈ {"*Sequence*", "*And*", "*Or*", "*Concurrent*", "*Not*", "*Unless*", "*At*", "*Every*", "*After*", "*Nth*", "*AtLeast*", "*AtMost*"}
operand-defs =< *operand-def*+ >
operand-def = *leaf-def* | *internal-operator-def*
detect-mode ∈ {"*immediate*", "*delayed*", "*deferred*"} `-- Default =` *immediate* | *deferred*
params = (*counting-n*, *at-ts*, *timing-interval*, *after-size*, *after-acc*)
counting-n ∈ *NUMBER*
at-tp ∈ *TIME-PATTERN*
timing-interval ∈ *TIME-INTERVAL*
after-size = (*after-rows*, *after-n*)
after-rows ∈ *BOOLEAN*
after-n ∈ *NUMBER* | {"*unbounded*"}
after-acc ∈ {"*add*", "*ignore*"}
leaf-def = (*type*, *as*, *threshold*, *collect*, *quant*, *consume*, *cond*)
collect = (*size*, *range*)
size = (*rows*, *size-val*)
rows ∈ *BOOLEAN* `-- Default` = *TRUE*
size-val ∈ *NUMBER* | {"*unbounded*"} `-- Default` = "*unbounded*"
range ∈ *TIME-INTERVAL* | {"*unbounded*"} `-- Default` = "*unbounded*"
quant = "*all*" | (*strict*, *age*, *max*, *staged*) `-- Default` = "*all*"
strict ∈ *BOOLEAN* `-- Default` = *FALSE*
age ∈ {"*new*", "*old*"} `-- Default` = "*new*"
max ∈ *NUMBER* | {"+"} `-- Default` = "+"
staged ∈ *BOOLEAN* `-- Default` = *FALSE*
consume ∈ *BOOLEAN* `-- Default` = *FALSE*
cond = *predicate*

Internal Event Patterns

internal-operator-def = (*sub-ctxt*, *sub-ep*, *sub-gkeys*, *sub-lkeys*)
sub-ctxt = (*inits*, *terms*)
sub-ep = (*op-name*, *operands*, *internal-detect-mode*, *params*)
sub-gkeys = {(*key*, *ranges*)} | {∅}
sub-lkeys = {(*key*, *ranges*)} | {∅}
internal-detect-mode ∈ {"*immediate*", "*deferred*"} `-- Default =` "*immediate*" | "*deferred*"

Mapping Expression

map-expr = {*attr-def*} | {∅} `-- Default` = ∅
attr-def = (*attr-name*, *attr-expr*)
attr-expr = *tuple-attr-expr* | *group-attr-expr*
event-name ∈ *STRING*
tuple-attr-expr = (*event-name*, *attr-name*) |
(*event-name*, *attr-name*, *attr-agg-fn*) |
(*event-name*, *event-agg-fn*, *set*) |
attr-agg-fn ∈ {"*MAX*", "*MIN*", "*SUM*", "*AVG*", "*MEDIAN*"}
event-agg-fn ∈ {"*COUNT*"}
group-attr-expr = (*event-name*, *attr-name*, *attr-agg-fn*) |
(*event-name*, *event-agg-fn*, *set*) |
(*attr-agg-fn*, *set*, *tuple-attr-expr*)
set ∈ *BOOLEAN*

Keys

gkeys = {(*key*, *ranges*)} | {∅}
lkeys = {(*key*, *ranges*)} | {∅}
key = {(*event-name*, *attr-name*)}
ranges = (*key-name*, *range-defs*) | *unnamed-range-defs* | {∅}
range-defs = *named-range-defs* | *unnamed-range-defs*
named-range-defs = {*named-range-def*}
unnamed-range-defs = {*range-descr*}
named-range-def = (*range-descr*, *as*)
range-descr ∈ *STRING-RANGE* | *NUM-RANGE* | {"*Rest*"}

Conditions

predicate = {*and-clause*} | {∅}
and-clause = {*or-clause*} | {∅}
or-clause = (*negated* ∈ *BOOLEAN*, *rel-expr*)
rel-expr = (*math-expr*, *math-comparator*, *math-expr*) |
(*string-expr*, *string-comparator*, *string-expr*) |
(*boolean-expr*, *string-comparator*, *boolean-expr*) |
(*timestamp-expr*, *timestamp-comparator*, *timestamp-expr*) |
b ∈ *BOOLEAN*
math-comparator ∈ {" < ", " > ", " <= ", " >= ", " = ", "! = "}
string-comparator ∈ {" = ", "! = "}
timestamp-comparator = *math-comparator*

$\textit{math-expr} = (\textit{math-expr},\ \textit{binary-math-op},\ \textit{math-expr}) \mid$
$\quad (\textit{unary-math-op},\ \textit{math-expr}) \mid$
$\quad (\textit{event-name},\ \textit{attr-name}) \mid n \in \textit{NUMBER} \mid \textit{number-is-null}$
$\textit{string-expr} = (\textit{event-name},\ \textit{attr-name}) \mid s \in \textit{STRING} \mid \textit{string-is-null}$
$\textit{boolean-expr} = (\textit{event-name},\ \textit{attr-name}) \mid b \in \textit{BOOLEAN} \mid \textit{boolean-is-null}$
$\textit{timestamp-expr} = \textit{fn} \mid \textit{ts} \in \textit{TIME-STAMP} \mid \textit{ts-is-null}$
$\textit{binary-math-op} \in \{\text{“}+\text{”},\ \text{“}-\text{”},\ \text{“}*\text{”},\ \text{“/”}\}$
$\textit{fn} = (\textit{start-ts},\ \textit{event-name}) \mid (\textit{end-ts},\ \textit{event-name}) \mid (\textit{sst},\ \textit{event-name}) \mid \textit{stable-time}$
$\textit{start-ts} \in (\textit{STRING} \mapsto \textit{TIME-STAMP})$
$\textit{end-ts} \in (\textit{STRING} \mapsto \textit{TIME-STAMP})$
$\textit{sst} \in (\textit{STRING} \mapsto \textit{TIME-STAMP})$
$\textit{stable-time} \in (\{\emptyset\} \mapsto \textit{TIME-STAMP})$
$\textit{number-is-null} = (\textit{event-name},\ \textit{attr-name},\ n \in \textit{NUMBER})$
$\textit{boolean-is-null} = (\textit{event-name},\ \textit{attr-name},\ b \in \textit{BOOLEAN})$
$\textit{string-is-null} = (\textit{event-name},\ \textit{attr-name},\ s \in \textit{STRING})$
$\textit{ts-is-null} = (\textit{event-name},\ \textit{attr-name},\ \textit{ts} \in \textit{TIME-STAMP})$

Conditions with Aggregates

$\textit{agg-predicate} = \{\textit{agg-and-clause}\} \mid \{\emptyset\}$
$\textit{agg-and-clause} = \{\textit{agg-or-clause}\} \mid \{\emptyset\}$
$\textit{agg-or-clause} = (\textit{agg-negated} \in \textit{BOOLEAN},\ \textit{rel-expr})$
$\textit{agg-rel-expr} = (\textit{agg-math-expr},\ \textit{math-comparator},\ \textit{agg-math-expr}) \mid$
$\quad (\textit{agg-string-expr},\ \textit{string-comparator},\ \textit{agg-string-expr}) \mid$
$\quad (\textit{agg-boolean-expr},\ \textit{string-comparator},\ \textit{agg-boolean-expr}) \mid$
$\quad (\textit{agg-timestamp-expr},\ \textit{timestamp-comparator},\ \textit{agg-timestamp-expr})$
$\textit{math-comparator} \in \{\text{“}<\text{”},\ \text{“}>\text{”},\ \text{“}<=\text{”},\ \text{“}>=\text{”},\ \text{“}=\text{”},\ \text{“}!=\text{”}\}$
$\textit{string-comparator} \in \{\text{“}=\text{”},\ \text{“}!=\text{”}\}$
$\textit{timestamp-comparator} = \textit{math-comparator}$
$\textit{agg-math-expr} = (\textit{agg-math-expr},\ \textit{binary-math-op},\ \textit{agg-math-expr}) \mid$
$\quad (\textit{unary-math-op},\ \textit{agg-math-expr}) \mid$
$\quad (\textit{attr-agg-fn},\ \textit{event-name},\ \textit{attr-name}) \mid$
$\quad (\textit{event-agg-fn},\ \textit{event-name}) \mid$
$\quad n \in \textit{NUMBER} \mid \textit{number-is-null}$
$\textit{agg-string-expr} = (\textit{attr-agg-fn},\ \textit{event-name},\ \textit{attr-name}) \mid$
$\quad (\textit{event-agg-fn},\ \textit{event-name}) \mid$
$\quad s \in \textit{STRING} \mid \textit{string-is-null}$
$\textit{agg-boolean-expr} = (\textit{attr-agg-fn},\ \textit{event-name},\ \textit{attr-name}) \mid$
$\quad (\textit{event-agg-fn},\ \textit{event-name}) \mid$
$\quad b \in \textit{BOOLEAN} \mid \textit{boolean-is-null}$
$\textit{agg-timestamp-expr} = \textit{fn} \mid \textit{ts} \in \textit{TIME-STAMP} \mid \textit{ts-is-null}$

Dynamic Data Structures

$gps = \{(values,\ lps)\}$
$lps = \{(values,\ ctxts)\}$
$ctxts = \{ctxt\text{-}inst\}$
$ctxt\text{-}inst = (init\text{-}events,\ term\text{-}events,\ buf\text{-}events,\ ep\text{-}inst)$
$init\text{-}events \in \mathbb{P}\ E\text{-}INST$
$term\text{-}events \in \mathbb{P}\ E\text{-}INST$
$buf\text{-}events \in \mathbb{P}\ CE\text{-}INST$
$ep\text{-}inst =< operand\text{-}inst+ >$
$operand\text{-}inst = leaf\text{-}inst \mid internal\text{-}operator\text{-}inst$
$leaf\text{-}inst =< leaf\text{-}event* >$
$leaf\text{-}event \in E\text{-}INST$
$internal\text{-}operator\text{-}inst = sub\text{-}gps$
$sub\text{-}gps =< (values,\ sub\text{-}lps)* >$
$sub\text{-}lps =< (values,\ sub\text{-}ctxts)* >$
$sub\text{-}ctxts =< sub\text{-}ctxt\text{-}inst* >$
$sub\text{-}ctxt\text{-}inst = (init\text{-}events,\ term\text{-}events,\ ep\text{-}inst)$
$values \in STRING\text{-}RANGE \mid NUM\text{-}RANGE$

Appendix B

Input Type Schemas for Examples

In this appendix, we list the type schemas of the input events used in the examples of Chapter 3. The timestamp attribute of an event is implicit, and thus is not included in its schema.

```
TempEvent(value:NUMBER, loc:NUMBER)
HeartRateEvent(value:NUMBER, patientId:NUMBER)
24Hrs()
At8am()
At6pm()
CarEvent(regNo:NUMBER, linkId:NUMBER, speed:NUMBER)
Fall(patientId:NUMBER)
HeartRateIncrease(patientId:NUMBER, amount:NUMBER)
BloodPressureDecrease(patientId:NUMBER, amount:NUMBER)
BloodPressure(patientId:NUMBER, level:STRING)
SignificantHeartRateIncrease(patientId:NUMBER, amount:NUMBER)
BagEvent(id:NUMBER, carouselId:NUMBER)
ContainerEvent(id:NUMBER, trackId:NUMBER)
BusEvent(regNo:NUMBER, linkId:NUMBER, speed:NUMBER)
```

Bibliography

[AAB+05] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The Design of the Borealis Stream Processing Engine. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005. 112

[ABC+04] Tarek F. Abdelzaher, Brian M. Blum, Qing Cao, Yong Chen, David Evans, Joshua George, Selvin George, Lin Gu, Tian He, Sudha Krishnamurthy, Liqian Luo, Sang Hyuk Son, Jack Stankovic, Radu Stoleru, and Anthony D. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 582–589. IEEE Computer Society, 2004. 39

[ABW03] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford University, 2003. 48, 83

[AC03] Raman Adaikkalavan and Sharma Chakravarthy. SnoopIB: Interval-Based Event Specification and Detection for Active Databases. In *Proceedings of the East European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 190–204, 2003. 70, 85

[ACc+03] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003. 34, 48, 83, 112

[ADB+99] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a Better Understanding of Context and Context-Awareness. In *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing (HUC)*, pages 304–307. Springer-Verlag, 1999. 35

[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD*

international conference on Management of data (SIGMOD), pages 147–160, New York, NY, USA, 2008. ACM. 83

[AE02] Asaf Adi and Opher Etzion. The situation manager rule language. In Michael Schroeder and Gerd Wagner, editors, *RuleML*, volume 60 of *CEUR Workshop Proceedings*, pages 36–57. CEUR-WS.org, 2002. 46, 82

[AE04] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, 2004. 46, 67, 82, 130

[ASSC02] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114, August 2002. 21

[Bar04] Jakob E. Bardram. Applications of context-aware computing in hospital work: examples and design principles. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1574–1579, New York, NY, USA, 2004. ACM Press. 20

[Bat94] Don Batory. The LEAPS Algorithm. Technical report, University of Texas at Austin, Austin, TX, USA, 1994. 51, 85

[Bau04] Martin Bauer. Event Management for Mobile Users. Technical Report 2, Universitat Stuttgart, 2004. 36, 91, 111

[BB03] Boris Jan Bonfils and Philippe Bonnet. Adaptive and Decentralized Operator Placement for In-Network Query Processing. In *Proceedings of the Second International Workshop on Information Processing in Sensor Networks (IPSN)*, volume 2634 of *Lecture Notes in Computer Science*, pages 47–62. Springer–Verlag Berlin Heidelberg, 2003. 41, 124, 128

[BBE+08] Jean Bacon, Alastair Beresford, David Evans, David Ingram, Niki Trigoni, Alexandre Guitton, and Antonios Skordylis. TIME: An open platform for capturing, processing and delivering transport-related data. In *Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC)*, pages 687–691, Las Vegas, NV, USA, January 2008. Session on Sensor Networks in Intelligent Transportation Systems. 55

[BBHM95] Jean Bacon, John Bates, Richard Hayton, and Ken Moody. Using Events to Build Distributed Applications. In *Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments (SDNE)*, page 148, Washington, DC, USA, 1995. IEEE Computer Society. 29

[BC04] Gregory Biegel and Vinny Cahill. A Framework for Developing Mobile Context-aware Applications. In *Proceedings of the Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, Orlando, FA., USA, March 2004. IEEE. 37

[BCSS99] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A Case for Message Oriented Middleware. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC)*, pages 1–18, London, UK, 1999. Springer-Verlag. 25

[Ber96] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996. 23

[BFKM85] Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. *Programming expert systems in OPS5: an introduction to rule-based programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1985. 85

[BFSF08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the second international conference on Distributed event-based systems (DEBS)*, pages 265–275, New York, NY, USA, 2008. ACM. 111

[BGAH07] Roger S. Barga, Jonathan Goldstein, Mohamed H. Ali, and Mingsheng Hong. Consistent Streaming Through Time: A Vision for Event Stream Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* [DBL07], pages 363–374. 112

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards Sensor Database Systems. In *Proceedings of the International Conference on Mobile Data Management (MDM)*, pages 3–14. Springer-Verlag, 2001. 41

[BKS+99] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 114–121. IEEE Computer Society Press, 1999. 32

[BKZD04] Michael Beigl, Albert Krohn, Tobias Zimmer, and Christian Decker. Typical Sensors needed in Ubiquitous and Pervasive Computing. In *Proceedings of the International Conference on Networked Sensing Systems (INSS)*, pages 153–158, 2004. 21

[BM07a] Jean Bacon and Ken Moody. CareGrid: Autonomous Trust Domains for Healthcare Applications. http://www.cl.cam.ac.uk/research/srg/opera/projects/CareGrid/case.html, 2007. Last accessed on 20/11/2009. 55

[BM07b] Jean Bacon and Ken Moody. TIME-EACM: A Transport Information Monitoring Environment - Event Architecture and Context Management. http://www.cl.cam.ac.uk/~jmb25/TIME-EACM.htm, 2007. Last accessed on 20/11/2009. 55

[BMB+00] Jean Bacon, Ken Moody, John Bates, Richard Hayton, Chaoying Ma, Andrew McNeil, Oliver Seidel, and Mark Spiteri. Generic Support for Distributed Applications. *Computer*, 33(3):68–76, 2000. 29

[BSB+02] Sumeer Bhola, Robert E. Strom, Saurabh Bagchi, Yuanyuan Zhao, and Joshua S. Auerbach. Exactly-once delivery in a content-based publish-subscribe system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 7–16, Washington, DC, USA, 2002. IEEE Computer Society. 32

[BTW+06] Yijian Bai, Hetal Thakkar, Haixun Wang, Chang Luo, and Carlo Zaniolo. A data stream language and system designed for power and extensibility. In *Proceedings of the 15th ACM international conference on Information and knowledge management (CIKM)*, pages 337–346, New York, NY, USA, 2006. ACM. 71, 83

[BV06] Roberto Baldoni and Antonino Virgillito. Distributed event routing in publish/subscribe communication systems: a survey. Technical Report MIDLAB 1/2006, Dipartimento di Informatica e Sistemistica, University di Roma la Sapienza, 2006. 29

[BWL+07] Yijian Bai, Fusheng Wang, Peiya Liu, Carlo Zaniolo, and Shaorong Liu. RFID Data Processing with a Data Stream Query Language. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 1184–1193. IEEE, 2007. 84

[BZA03] Sumeer Bhola, Yuanyuan Zhao, and Joshua Auerbach. Scalably Supporting Durable Subscriptions in a Publish/Subscribe System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, volume 0, page 57, Los Alamitos, CA, USA, 2003. IEEE Computer Society. 32

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Ugur Çetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003. 34

[CBo09] Crossbow's Homepage. `http://www.xbow.com`, 2009. Last accessed on 20/11/2009. 21

[CcC+02] Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 215–226, Hong Kong, China, August 2002. 34, 48, 83, 91, 112

[CCD+03] Srirash Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Sam Madden, Vijayshanker Raman, Fred Reiss, and Mehul Shah. TelegraphCQ: Continuous Dataflow

Processing for an Uncertain World. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*, 2003. 83

[CDHR03] Miguel Castro, Peter Druschel, Y. Charlie Hu, and Antony Rowstron. Exploiting network proximity in peer-to-peer overlay networks. Technical Report MSR-TR-2003-82, Microsoft, 2003. 125

[CEE+01] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: Application driver for wireless communications technology. In *Proceedings of the ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, 2001. 21

[CFJ03] Harry Chen, Tim Finin, and Anupam Joshi. Semantic Web in a Pervasive Context-Aware Architecture. *Artificial Intelligence in Mobile System*, pages 33–40, October 2003. 36

[CJSS03] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: a stream database for network applications. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 647–651, New York, NY, USA, 2003. ACM Press. 83

[CK05] Guanling Chen and David Kotz. Policy-Driven Data Dissemination for Context-Aware Applications. In *Proceedings of the Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, pages 283–289, Washington, DC, USA, 2005. IEEE Computer Society. 35, 36

[CKAK94] Sharma Chakravarthy, Vidhya Krishnaprasad, Eman Anwar, and Seung-Kyum Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 606–617, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. 45, 69, 81, 103

[CL04] Jan Carlson and Björn Lisper. An event detection algebra for reactive systems. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT)*, pages 147–154, New York, NY, USA, 2004. ACM. 85

[CLK04] Guanling Chen, Ming Li, and David Kotz. Design and implementation of a large-scale context fusion network. In *Proceedings of the 1st Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pages 246–255. IEEE Computer Society, 2004. 35, 36

[CM94] Sharma Chakravarthy and Deepak Mishra. Snoop: an expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994. 45, 69, 81

[CRL09] Jianxia Chen, Lakshmish Ramaswamy, and David Lowenthal. Towards efficient event aggregation in a decentralized publish-subscribe system. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems (DEBS)*, pages 1–11, New York, NY, USA, 2009. ACM. 141

[CRW01] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001. 29, 140

[DAS99] Anind K. Dey, Gregory D. Abowd, and Daniel Salber. A Context-Based Infrastructure for Smart Environments. In Paddy Nixon, Gerard Lacey, and Simon Dobson, editors, *1st International Workshop on Managing Interactions in Smart Environments (MANSE)*, pages 114–128, Dublin, Ireland, December 1999. 35

[DBL07] *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*. www.crdrdb.org, 2007. 157, 160

[DBS+01] Ken Ducatel, Marc Bogdanowicz, Fabiana Scapolo, Jos Leijten, and Jean-Claude Burgelman. Scenarios for ambient intelligence in 2010. Technical report, IST Advisory Group, February 2001. 19

[DCKM04] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. Vivaldi: a decentralized network coordinate system. *SIGCOMM Comput. Commun. Rev.*, 34(4):15–26, 2004. 122

[DGP+07] Alan J. Demers, Johannes Gehrke, Biswanath Panda, Mirek Riedewald, Varun Sharma, and Walker M. White. Cayuga: A General Purpose Event Monitoring System. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)* [DBL07], pages 412–422. 82

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003. 26, 27, 96

[EG01] W. Keith Edwards and Rebecca E. Grinter. At Home with Ubiquitous Computing: Seven Challenges. In *Proceedings of the 3rd international conference on Ubiquitous Computing (UbiComp)*, pages 256–272, London, UK, 2001. Springer-Verlag. 20

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 254–269, New York, NY, USA, 2001. ACM. 28

[ESP09] Esper's Homepage. http://esper.codehaus.org, 2009. Last accessed on 20/11/2009. 84

[ESS02] Deborah Estrin, Mani Srivistava, and Akbar Sayeed. Tutorial on Wireless Sensor Networks. ACM MobiComm, September 2002. 21

[Eug07] Patrick Eugster. Type-based publish/subscribe: Concepts and experiences. *ACM Trans. Program. Lang. Syst.*, 29(1):6, 2007. 28

[FM09] Derek Fagan and René Meier. Using context and behavioral patterns for intelligent traffic management. In *Proceedings of the 1st International Workshop on Context-Aware Middleware and Services (CAMS)*, pages 61–66, New York, NY, USA, 2009. ACM. 20

[For82] Charles Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. *Artif. Intell.*, 19(1):17–37, 1982. 50, 85

[GD93] Stella Gatziu and Klaus R. Dittrich. Events in an Active Object-Oriented Database System. Technical report, University of Zurich, 1993. 81

[GD94] Stella Gatziu and Klaus R. Dittrich. Detecting Composite Events in Active Database Systems Using Petri Nets. In *Proceedings of the International Workshop on Research Issues in Data Engineering (RIDE-ADS)*, pages 2–9, 1994. 81

[GGG05] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming Wireless Sensor Networks Using *Kairos*. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, 2005. 40, 42

[GJ92] Narain H. Gehani and Hosagrahar V. Jagadish. Composite event specification in active databases: Model and implementation. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 327–338, 1992. 45, 81

[GJS92] Narain H. Gehani, Hosagrahar V. Jagadish, and Oded Shmueli. Event specification in an active object-oriented database. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 81–90. ACM Press, 1992. 45, 81

[GK02] Koichi Goto and Yahiko Kambayashi. A new passenger support system for public transport using mobile database access. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 908–919. VLDB Endowment, 2002. 20

[GSG02] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. *SIGCOMM Comput. Commun. Rev.*, 32(3):11–11, 2002. 122

[GSR+03] Richard Glassey, Graeme Stevenson, Matthew Richmond, Paddy Nixon, Sotirios Terzis, Feng Wang, and Ian Ferguson. Towards a middleware for generalised context management. In M. Endler and D. Schmidt, editors, *Proceedings of the International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC)*, pages 45–52, June 2003. 36

[Hay96] Richard Hayton. *An Open Architecture for Secure Interworking Services.* PhD thesis, Fitzwilliam College, University of Cambridge, March 1996. 29, 110

[HGM01] Yongqiang Huang and Hector Garcia-Molina. Replicated condition monitoring. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–237, New York, NY, USA, 2001. ACM. 92

[HH93] Eric Hanson and Mohammed S. Hasan. Gator: An Optimized Discrimination Network for Active Database Rule Condition Testing. Technical report, University of Florida, 1993. 86

[HHM03] Joseph M. Hellerstein, Wei Hong, and Samuel R. Madden. The sensor spectrum: technology, trends, and requirements. *SIGMOD Rec.*, 32(4):22–27, 2003. 40

[HMCP04] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18:6–14, Jan/Feb 2004. 21, 41

[Hom02] A. Hombrecher. *Reconciling Event Taxonomies across Administrative Domains.* PhD thesis, Jesus College, University of Cambridge, June 2002. 116, 118

[Hop00] Andy Hopper. The Clifford Paterson Lecture, 1999. Sentient computing. *Philosophical Transactions of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 358(1773):2349–2358, August 2000. 19, 37

[HP01] Yehuda Hassin and David Peleg. Sparse communication networks and efficient routing in the plane. *Distrib. Comput.*, 14(4):205–215, 2001. 128

[IBM09] IBM Corporation. IBM WebSphere MQ. `http://www.ibm/com/software/integration/wmq/`, October 2009. Last accessed on 20/11/2009. 26

[JAC04] Qingchun Jiang, Raman Adaikkalavan, and Sharma Chakravarthy. Estreams: Towards an Integrated Model for Event and Stream Processing. Technical Report CSE-2004-3, The University of Texas at Arlington, 2004. 54, 58, 84

[JCH05] Chun Jin, Jaime G. Carbonell, and Philip J. Hayes. Argus: Rete + dbms = efficient persistent profile matching on large-volume data streams. In Mohand-Said Hacid, Neil V. Murray, Zbigniew W. Ras, and Shusaku Tsumoto, editors, *ISMIS*, volume 3488 of *Lecture Notes in Computer Science*, pages 142–151. Springer, 2005. 86

[JMS95] Hosagrahar V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz. View maintenance issues for the chronicle data model. In *Proceedings of the fourteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems (PODS)*, pages 113–124, New York, NY, USA, 1995. ACM Press. 83

[JS03a] Yuhui Jin and Rob Strom. Relational subscription middleware for Internet-scale publish-subscribe. In *Proceedings of the International Workshop on Distributed Event-based Systems (DEBS)*, pages 1–8, New York, NY, USA, 2003. ACM. 32

[JS03b] Glenn Judd and Peter Steenkiste. Providing Contextual Information to Pervasive Computing Applications. In *Proceedings of the Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, page 133, Washington, DC, USA, 2003. IEEE Computer Society. 37

[KBM04] Eli Katsiri, Jean Bacon, and Alan Mycroft. An Extended Publish/Subscribe Protocol for Transparent Subscriptions to Distributed Abstract State in Sensor Driven Systems using Abstract Events. In *Proceedings of the International Workshop on Distributed Event-based Systems (DEBS)*, pages 68–73, Edinburgh, Scotland, May 2004. 37

[KKH+08] Sunil Kumar, Kashyap Kambhatla, Fei Hu, Mark Lifson, and Yang Xiao. Ubiquitous computing for remote cardiac patient monitoring: a survey. *Int. J. Telemedicine Appl.*, 2008:1–19, 2008. 20

[KKP99] Joseph M. Kahn, Randy H. Katz, and Kristofer S. J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 271–278, 1999. 21

[Kna00] Ara Knaian. A Wireless Sensor Network for Smart Roadbeds and Intelligent Transportation Systems. Master's thesis, MIT Department of Electrical Engineering and Computer Science and the MIT Media Laboratory, April 2000. 54

[Kra05] Mark Kranz. SENSID: a Situation Detector for Sensor Networks. Honours Thesis, School of Computer Science and Software Engineering, University of Western Australia., June 2005. 47

[KW03] Holger Karl and Andreas Willig. A short survey of wireless sensor networks. Technical Report TKN-03-018, Telecommunication Networks Group, Technische Universität Berlin, October 2003. This technical report also appeared as a contribution to the report of the Working Group 2 "Ad hoc networks" of the Arbeitsgruppe Mobilkommunikation, DLR/BMBF. 21

[LBBN04] Othmar Lehmann, Martin Bauer, Christian Becker, and Daniela Nicklas. From home to world - supporting context-aware applications through world models. In *Proceedings of the Annual IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, page 297. IEEE Computer Society, March 2004. 36

[LCB99] Christoph Liebig, Mariano Cilia, and Alejandro Buchmann. Event Composition in Time-Dependent Distributed Systems. In *Proceedings of the IECIS International Conference on Cooperative Information Systems (CoopIS)*, page 70. IEEE Computer Society, 1999. 56, 93, 110

[LCL+03] Jie Liu, Maurice Chu, Juan Liu, James Reich, and Feng Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 2(4):50–62, 2003. 40

[LHJ05] Guoli Li, Shuang Hou, and Hans-Arno Jacobsen. A Unified Approach to Routing, Covering and Merging in Publish/Subscribe Systems Based on Modified Binary Decision Diagrams. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 447–457, Washington, DC, USA, 2005. IEEE Computer Society. 31

[LJ05] Guoli Li and Hans-Arno Jacobsen. Composite Subscriptions in Content-Based Publish/Subscribe Systems. In Gustavo Alonso, editor, *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware)*, volume 3790 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2005. 31, 96, 115, 140

[LMJ07] Guoli Li, Vinod Muthusamy, and Hans-Arno Jacobsen. Adaptive Content-based Routing in General Overlay Topologies. Technical report, University of Toronto, Middleware Systems Research Group, 2007. 31, 140

[LPMS07] Jonathan Ledlie, Peter Pietzuch, Michael Mitzenmacher, and Margo Seltzer. Wired Geometric Routing. In *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007. 128

[LPS06] Jonathan Ledlie, Peter Pietzuch, and Margo Seltzer. Stable and Accurate Network Coordinates. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, page 74, Washington, DC, USA, 2006. IEEE Computer Society. 122

[LSS03] Shuoqi Li, Sang H. Son, and John A. Stankovic. Event Detection Services Using Data Service Middleware in Distributed Sensor Networks. In F. Zhao and L. Guibas, editors, *Proceedings of the Second International Workshop on Information Processing in Sensor Networks (IPSN)*, volume 2634 of *LNCS*, pages 502–517, Palo Alto, CA, USA, April 2003. Springer-Verlag. 41

[MB90] Daniel P. Miranker and David A. Brant. On the Performance of Lazy Matching in Production Systems. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI)*, pages 685–692. AAAI Press/The MIT Press, 1990. 51, 85

[MB98] Chaoying Ma and Jean Bacon. COBEA: a CORBA-based event architecture. In *Proceedings of the 4th conference on USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, page 9, Berkeley, CA, USA, 1998. USENIX Association. 29

[MFHH02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002. 40

[MFHH03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. The design of an acquisitional query processor for sensor networks. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 491–502. ACM Press, 2003. 41, 53

[Mir87] Daniel P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. PhD thesis, Columbia University, New York, NY, USA, 1987. 85

[ML91] Daniel P. Miranker and Bernie J. Lofaso. The Organization and Performance of a TREAT-Based Production System Compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10, 1991. 85

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: an approach to universal topology generation. In *9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 346–353, August 2001. 133

[MPS+02] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless Sensor Networks and Applications (WSNA)*, pages 88–97. ACM Press, 2002. 21

[MR03] Sven Meyer and Andry Rakotonirainy. A survey of research on context-aware homes. In *Proceedings of the Australasian information security workshop conference (ACSW Frontiers)*, pages 159–168, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc. 20

[MSS97] Masoud Mansouri-Samani and Morris Sloman. Gem: a generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4(2):96–108, 1997. 44, 82, 111

[MWA+02] 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 ina Data Stream Management System. Technical Report 2002-41, Stanford InfoLab, 2002. 48, 83, 91, 112

[MZ95a] Iakovos Motakis and Carlo Zaniolo. Composite temporal events in active database rules: A logic-oriented approach. In *Proceedings of the International Workshop on Temporal Databases: Recent Advances in Temporal Databases*, pages 332–351. Springer Verlag, 1995. 66

[MZ95b] Iakovos Motakis and Carlo Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 19–37, 1995. 82

[MZ97a] Iakovos Motakis and Carlo Zaniolo. Formal Semantics for Composite Temporal Events in Active Database Rules. *Journal of Systems Integration*, 7(3/4):291–325, 1997. 82

[MZ97b] Iakovos Motakis and Carlo Zaniolo. Temporal aggregation in active database rules. *SIGMOD Rec.*, 26(2):440–451, 1997. 82

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information Processing in Sensor Networks (IPSN)*, pages 489–498, New York, NY, USA, 2007. ACM. 42

[OMG08] OMG. The Common Object Request Broker Architecture: Core Specification, Revision 3.1, Object Management Group (OMG), January 2008. 25

[OPSS93] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)*, pages 58–68, Asheville, NC, USA, December 1993. 27

[PCW+03] Marcelo Pias, Jon Crowcroft, Steve R. Wilbur, Tim Harris, and Saleem N. Bhatti. Lighthouses for Scalable Distributed Location. In M. Frans Kaashoek and Ion Stoica, editors, *Proceedings of the 6th International Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2735 of *Lecture Notes in Computer Science*, pages 278–291. Springer, 2003. 122

[PD99] Norman W. Paton and Oscar Díaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999. 81

[Pie04] Peter Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, Queens' College, University of Cambridge, February 2004. 14, 23, 32, 97, 116, 117, 131, 133, 145

[PLS+06] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 49, Washington, DC, USA, 2006. IEEE Computer Society. 34, 41, 124, 128, 131, 141

[Pow96] David Powell. Group communication. *Commun. ACM*, 39(4):50–53, 1996. 27

[PSB04] Peter Pietzuch, Brian Shand, and Jean Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network Magazine, Special Issue on Middleware Technologies for Future Communication Networks*, 18(1):44–55, January/February 2004. 55, 81, 96, 111, 117

[RAF+02] Jan Rabaey, Edward Arens, Clifford Federspiel, Ashok Gadgil, William Nazaroff David Messerschmitt, Kristofer Pister, Shmuel Oren, and Pravin Varaiya. Smart Energy Distribution and Consumption: Information Technology as an Enabling Force. Technical report, University of California at Berkeley, 2002. 21

[RAP09] The Stanford Rapide (TM) Project. `http://pavg.stanford.edu/rapide`, 2009. Last accessed on 20/11/2009. 84

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware)*, pages 329–350, London, UK, 2001. Springer-Verlag. 32

[RGK06] Thomas Repantis, Xiaohui Gu, and Vana Kalogeraki. Synergy: Sharing-Aware Component Composition for Distributed Stream Processing Systems. In Maarten van Steen and Michi Henning, editors, *Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware)*, volume 4290 of *Lecture Notes in Computer Science*, pages 322–341. Springer, 2006. 141

[Riz05] Shariq Rizvi. Complex Event Processing Beyond Active Databases: Streams and Uncertainties. Technical Report UCB/EECS-2005-26, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, 2005. 54, 84, 86

[RKCD01] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*,

volume 2233 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2001. 35

[RKM02] Kay Romer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):59–61, 2002. 38

[RM04a] Kay Römer and Friedemann Mattern. Event-Based Systems for Detecting Real-World States with Sensor Networks: A Critical Analysis. In *DEST Workshop on Signal Processing in Sensor Networks at ISSNIP*, pages 389–395, Melbourne, Australia, December 2004. 86

[RM04b] Kay Römer and Friedemann Mattern. The Design Space of Wireless Sensor Networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004. 21

[RMCZ06] Esther Ryvkina, Anurag S. Maskey, Mitch Cherniack, and Stan Zdonik. Revision Processing in a Stream Processing Engine: A High-Level Design. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 141, Washington, DC, USA, 2006. IEEE Computer Society. 112

[SAW94] Bill Schilit, Norman Adams, and Roy Want. Context-Aware Computing Applications. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society, 1994. 19

[Sch96] Scarlet Schwiderski. *Monitoring the Behaviour of Distributed Systems*. PhD thesis, Selwyn College, University of Cambridge, April 1996. 44, 81, 110

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008. 38, 40

[SPL+04] Jeffrey Shneidman, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, Margo Seltzer, and Matt Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical report, Harvard, 2004. 34

[Sun02] Sun Microsystems, Inc. Java Message Service (JMS) Specification. `http://java.sun.com/products/jms`, 2002. Last accessed on 20/11/2009. 26

[Sun05] Sun Microsystems, Inc. Core Java J2SE 5.0. `http://java.sun.com/j2se/1.5.0`, 2005. Last accessed on 20/11/2009. 28

[Sun06] Sun Microsystems, Inc. Java Remote Method Invocation (RMI) Specification. `http://java.sun.com/javase/6/docs/platform/rmi/spec/rmiTOC.html`, 2006. Last accessed on 20/11/2009. 25

[TB07] Salman Taherian and Jean Bacon. SPS: a middleware for multi-user sensor systems. In *Proceedings of the International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC)*, pages 19–24, New York, NY, USA, 2007. ACM. 86

[TC03] Liying Tang and Mark Crovella. Virtual landmarks for the internet. In *Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement (IMC)*, pages 143–152, New York, NY, USA, 2003. ACM. 122

[TGNO92] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 321–330, New York, NY, USA, 1992. ACM Press. 83

[TMSF03] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, 2003. 83

[W3C07] W3C. XQuery 1.0: An XML Query Language. W3C Recommendation, World Wide Web Consortium, January 2007. Last accessed on 20/11/2009. 33

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 407–418, New York, NY, USA, 2006. ACM. 82

[Wei93] Mark Weiser. Some computer science issues in ubiquitous computing. *Commun. ACM*, 36(7):75–84, 1993. 19

[WHFaG92] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1):91–102, 1992. 21

[WLLP01] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001. 21

[WM03] Ian Wright and James Marshall. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *International Journal of Intelligent Games and Simulation*, 2(1):36–48, February 2003. 50, 85

[WM04] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation (NSDI)*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association. 39

[WMG04] Alec Woo, Sam Madden, and Ramesh Govindan. Networking support for query processing in sensor networks. *Commun. ACM*, 47(6):47–52, 2004. 40

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the The International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 99–110, New York, NY, USA, 2004. ACM. 39

[Xu02] Ning Xu. A Survey of Sensor Network Applications. Technical report, Computer Science Department, University of Southern California, 2002. 21

[YB05] Eiko Yoneki and Jean Bacon. Unified Semantics for Event Correlation over Time and Space in Hybrid Network Environments. In *OTM Conferences (1)*, pages 366–384, 2005. 70, 85, 111

[YC99] Shuang Yang and Sharma Chakravarthy. Formal Semantics of Composite Events for Distributed Environments. In *Proceedings of the International Conference on Data Engineering (ICDE)*, page 400, Washington, DC, USA, 1999. IEEE Computer Society. 46, 81, 110

[YG02] Yong Yao and Johannes Gehrke. The Cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002. 41, 53

[YKP04] Yang Yu, Bhaskar Krishnamachari, and Viktor K. Prasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18:15–21, Jan/Feb 2004. 38

[ZMU97] Detlef Zimmer, Axel Meckenstock, and Rainer Unland. A General Model for Event Specification in Active Database Management Systems. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases (DOOD)*, pages 419–420, 1997. 82

[ZU99] Detlef Zimmer and Rainer Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 392–399. IEEE Computer Society Press, 1999. 82