Towards supporting mobility and advanced semantics in Event based systems

by

Kishor S. Mahajan

(Under the direction of Lakshmish Ramaswamy)

Abstract

Continuous advances in networking systems facilitated exponential increase in the use of largely connected computer systems and applications. These advances demand scalability of communication mechanisms while maintaining good performance. Proliferation of pervasive devices and fast growing wireless and sensor networking created a new paradigm – Mobile Computing. Prominent features of event-based systems, mainly, content-based systems provide a great support for the requirements of such a dynamic and vast network, and maintain good performance. Very few of the current content-based systems allow mobility, and those who allow it, do not have support for flexibility in event notifications. This work introduces mobility in Combined Broadcast Content Based system. It also adds support for histograms that aims toward supporting advanced semantics such as manycasting, faircasting and best-casting which will enable flexible event notifications.

Index words: Event based system, Mobile system, Publish/Subscribe system, Histogram, Siena, Combine Broadcast Content-Based Simulation

Towards supporting mobility and advanced semantics in Event based systems

by

Kishor S. Mahajan

B.E., University of Mumbai, India, 2008

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2011

Towards supporting mobility and advanced semantics in Event based systems

by

Kishor S. Mahajan

Approved:

Major Professor: Lakshmish Ramaswamy

Committee: Budak Arpinar
Maria Hybinette

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2011

Dedication

I dedicate this thesis to my loving, supporting, and encouraging parents – Aai & Pappa, elder brother Sujit and to all of my amazing and helpful friends.

Acknowledgments

I would like to thank Dr. Lakshmish Ramaswamy who helped me with his guidance and support in every step of this work. I would also like to thank my committee members Dr. Maria Hybinette and Dr. Budak Arpinar for their valuable time. I thank to all the staff and faculty members of the Department of Computer Science, who directly and indirectly influenced this work. I appreciate the assistance of the writing tutors Dr. Garrison Bickerstaff, Mr. Greg Timmons, and Dr. Karen Braxley from the Department of Academic Environment.

I am thankful to all my friends who have given me with some valuable suggestions and were always been there for me. Last but definitely not the least, I would like to express my gratefulness to my parents and my brother who always have, and will support, and encourage me.

Table of Contents

List of Tables

List of Figures

Chapter 1

INTRODUCTION

The study concerns supporting mobility in event-based systems as well as stepping forward toward supporting advanced semantics. This chapter in particular will provide background information, motivation for and challenges faced by this study, followed by defining the problem and brief discussion of the proposed solution. The remaining chapters will be outlined in the end.

1.1 Background and Motivation

Due to continuous advancements in networking systems, use of largely connected computer systems, mobile systems and applications has increased exponentially. These advances demand scalability of communication mechanisms while maintaining good performance. Building and maintaining such systems with a large number of components and continuously changing dynamic environments is a big challenge. Another interesting aspect of these systems is large amounts of data. Traditional systems which interact with users for fulfilling their requests are insufficient for processing this data. These concerns create the need for automatic processing of data and taking further actions accordingly; then, data-driven or information-driven applications enter the picture. These applications can compute data, perform specific tasks and/or take actions automatically based on data/information provided to the systems which will be used to check conditions and perform computations. The concept of The event-based system [EBS] is loosely based on this notion.

1.1.1 Event-Based Systems and Mobile Networking

An event that has occurred is conveyed by describing all the relevant data which is then automatically processed generating notification that conveys occurance of this event. This notification is delivered to users, who have expressed their interest in this kind of event information (e.g. - stock value of delta goes above 500). The event-based systems (a.k.a. Publish/Subscribe systems) were initially designed for fixed environment but recent developments in wireless networking technologies and the growing success of mobile computing devices, such as laptop computers, third generation mobile phones, personal digital assistants, watches and similar electronic devices are enabling new classes of applications that present challenging problems to designers. The range of mobile computing applications includes but not limited to location-based services, sensor networks, and ad-hoc networking. A combination of two or more of these applications and/or combining with other sources of data give more power for information dissemination. For example, location-based information can be combined with weather reports, information on traffic jams, or the availability of free parking spaces. In such cases, an EBS can propose routes that avoid places where traffic is high or weather conditions are unpleasant, or it can direct the driver to the nearest free parking space.

The event-based style has lots of potential to easily integrate autonomous and heterogeneous components into complex systems that are easy to evolve and scale. Hence event-based systems are very much advantageous in mobile networking. With the increase in mobile handheld devices, some of the prominent features of these systems will provide solutions for the challenges faced by mobility in todays dynamic environment. Challenges, such as disconnectedness and low computing powers can be addressed by having a few modifications in existing fixed environmental EBSs, so the existing applications will not be affected in terms of their performance and/or interface.[Carzaniga et al., 2004, Mascolo et al., 2002b,

Huang and Garcia-Molina, 2004] Finding out such modifications effectively and efficiently is the main purpose of this study.

1.1.2 Advanced Semantics

With the ability of handling mobile clients, the study is directed toward featuring advanced semantics in EBSs. Consider a scenario in which people subscribe to certain services, so they can be in touch with many service providers but at a time, publishers only want to notify a few of them. One of the best examples would be subscriptions for medical services during a natural disaster. For example, a person could become injured and need medical attention. Now, he must have subscribed to more than one doctor (for example 20), in case the regular one is busy during the crisis. If he sends notifications to all the 20 doctors, imagine complications that could result if all of them responding at the same time. Such an adverse situation is not acceptable since many people might need medical attention at the same time. Thus, system should be able to randomly choose a few (e.g. 3) of the 20 doctors along with the regular one. We have constructed *histograms* for this purpose. Histogram is a technique to keep statistics related to some data. The way histograms are designed, depends upon the type of data and requirements. In EBS, it can be designed to identify ranges of data that falls into different types of subscriptions.

1.1.3 Combined Broadcast and Content-Based Systems

A content-based network is a communication network that features a new advanced communication model in which messages are not given explicit destination addresses and the destinations of a message are determined by matching the contents of the message against selection predicates declared by nodes.[Carzaniga et al., 2004] When designed as content-based systems, event-based systems provide a powerful architecture for decoupled asynchronous communication which is highly desirable for mobile systems. Antonio Carzania, et al. combined traditional broadcast protocol along with content-based routing protocol (also

developed by them) and provided a powerful scheme for EBSs. Unfortunately, the default implementation is mostly concentrated on scalable communication through notifications and deals with fixed networks. This default implementation does not have support for unsubscription and mobility. In this study, we add both the supports within this model without affecting much of the performance. Also the support for providing advanced semantics, such as manycasting and faircasting is added with the required trade-off.[Carter et al., 2003] All these concepts will be explained in later chapters.

1.2 Mobility and Advance Semantics in Event-based Systems

As discussed before, huge increase in the use of mobile gadgets such as laptops, PDAs and cell phones is seen in recent years. With more than half of the world's population owning a cell phone, it is increasingly becoming the primary communication/computation device. Sensors are widely used in areas such as supply chain management, environment monitoring, surveillance, military applications and highway monitoring. In medical fields, recent research increased use of in-body sensors for patient monitoring.[Ramaswamy et al., 2009, Zhong et al., 2008, Yoon et al., 2006, Repantis et al., 2006] In dynamic traffic monitoring many sensors such as video cameras, speed checkers, etc. are evolving day by day.[Li et al., 2007] Location based advertising is another application area which highly desire the knowledge of moving subscribers. All these changes indicate a massive use of wireless sensors in upcoming years. In the context of these developments, the event-based middleware has to handle different applications in terms of variety, complexity, scale, and importance of applications. We can already see a few rudimentary event-based services in context of social networks such as Twitter. People who are following someone, acts as subscribers to all the events that person twitting, and the twitting person acts as a publisher. The servers used by www.twitter.com are brokers. This form of EBS is rudimentary because all the messages twitted by a person, will be delivered to all the followers.

Over past decade, event-based systems have evolved into a distinct communication paradigm.[Chen et al., 2009] There is also research on event composition, complex event detection, automatic service composition, advanced event processing and notification in service runtime environment, and many other fields related to event-driven applications.[O'Keeffe and Bacon, 2010, Keeney et al., 2008, Michlmayr et al., 2008]

However, state of art infrastructures are inadequate for pervasive environment (mobile and sensor networks) for many applications. Currently, publishers and subscribers do not have any control over any aspect of message delivery. They only know how to publish or subscribe for the events. Such a model is too rigid for many of the modern applications. Current system, also, do not consider many of the issues related to mobile applications such as dynamic environment. Moreover, we need advanced semantics in event based system to make it more flexible. We need such flexibility for different issues (one example – natural disaster is given above). A system with such flexibility will also help to reduce the communication cost by avoiding unnecessary propagation of data.

Consider a simple scenario. People move from one place to another for various reasons, e.g. person leaving in Athens (a small town near Atlanta) and having a job in Atlanta. It is possible that the server (called as event broker in EBS) which is handling the subscriptions/notifications related to services connected subscriber(s) have subscribed for will not have coverage for such a greater distance. Eventually there is a need of moving a few, or all of the subscriptions along with them depending on the services desired, a few subscription related strictly to Athens need not be moved e.g. pizza offers in Athens between 8 am to 5 pm. One easy way to handle this is unsubscribing for all the subscriptions from broker in Athens and resubscribing them with broker in Atlanta. But a subscriber may no have a device powerful enough to carry his subscription all the way to the new broker. Also, the subscriber may not even realize that he is leaving vicinity of one broker and entering another's.

Also, there might be multiple brokers between the original broker and the destination broker. It is not feasible to perform unsubscription and re-subscription at each of them. This study helps to do this process more efficiently so that we have to perform unsubscription only once at source and have to re-subscribe again only once at the destination broker. This is not possible directly because a broker only has knowledge of his neighbors, nad may not know about the new broker. Also, original CBCB routing scheme which we are using, does not provide good support for subscription and no support at all for unsubscription. If a new subscription is done at one event broker, old one was being overwritten. In reality, it should merge old subscription with new one. Also, unsubscription support is necessary or else we have to reset whole system and resubscribe for all the subscriptions except the one we want to unsubscribe, which, clearly, will consume higher number of resources and will have much higher cost of operations. While implementing mobility, we are also addressing these two issues.

Different applications are already mentioned above where support for mobility and advanced semantics is necessary. Some of those scenarios explained in brief here, will outline the importance of the goals of this study.

Consider a *traffic monitoring and report system*. It will serve individuals who are interested in personalized reports on traffic situations in relevant locations on their mobile devices. I will also serve traffic management agencies which are interested in monitoring overall traffic in an area/city. Since developing worlds may have subscribers with low end mobile devices such as auto-rickshaw drivers in country like India, etc., we consider enhanced version of system able to handle the additional needs. Current system considers event data generated by statically deployed sensors and short messages sent by users in predefined format.

Another motivating scenario will be *monitoring patients/elderly persons*. Many elderly people or chronically ill patients remain at their home. Various kinds of in-body as well as external sensors monitoring their health (such at heart rate, etc.) are becoming critical tools. Doctors, nurses and other services need to be notified in the situation where their assistance is required. One or more sensor may have to report one more events to trigger the notification in this scenario.

Such scenarios inspired the study and implementation of mobility and advanced semantics in event-based system. Next chapters will be discussing this study in details.

1.3 Defining Problem

Among different types of filtering styles used in EBSs such as subject-based filtering, type-based filtering and content-based filtering, we needed to choose the most powerful yet easy to manage style. We chose content-based filtering since subscriber can easily express powerful subscriptions which not only describe the events of interest but also have some constraints on them. For example, a client is interested in Delta shares and has brought a few for 400 but he wants to be notified whenever the stock value goes above 500, so he can sell some of them and enjoy the profit. In content-based systems, he can express this wish easily by subscribing for Company = Delta && Stock > 500. He can add or remove some constraints according to his requirements. Topology which we selected, consists of broker networks in which all the brokers are interconnected forming a massive network, and each broker has his own network of subscribers and publishers connected to it.

While implementing mobility inside EBSs, we need to consider the issues that are involved, such as physical mobility (i.e. client is moving from one broker network to another broker network) must be handled. This mobility handling should be a transparent process. Since we are using a simulator, we have an API to specify this but a suggestion or two will

be provided to make the process transparent. Advanced semantics allow clients to have a flexible event notification system. We have implemented histograms for this purpose. The construction of histograms is based on the type of data we are interested in. Since in our case, we are dealing with constraints and based on those constraints the notifications are passed and histograms are built to store information about the range of data and the number of subscribers who are interested in those events by imposing a few constraints.

1.4 **Overview**

Chapter 2 will discuss event-based systems in detail as well as some related work. Chapter 3 concerns mobility in EBS and challenges for implementing mobility within the system. The approaches taken to solve these challenges are mentioned, and the future work for current implementation is explained in Chapter 4. Chapter 5 will overview advanced semantics we want to implement within this system. Chapter 6 will share the experimental results and their implications. Finally, Chapter 7 will conclude and discuss relevant future work.

Chapter 2

EVENT BASED SYSTEMS

2.1 Introduction to Event-Based Systems

In contrast to traditional request/reply mode of interaction for obtaining services, components in event-based systems are decoupled and interact with each other by generating and receiving event notifications, where an *event* is defined as change in the state of component of interest and notification contains data describing the event. Figure 2.1 represents a simple event-based system and how different components interact with each other.[Fiege et al., 2002] One other hand, Figure 2.1 represents the underlying event management system consisting of event brokers.[Zeidler and Fiege, 2003] In a simple case, we have *subscribers* who express their interest in some service (e.g. - Traffic delay on I-85 is more than 10 minutes) by means of *subscriptions* and we have some *publishers* who produce *notifications* after a certain event of interest occurred (e.g. - Traffic delay on I-85 is 15-20 minutes). *Advertisements* are issued by publishers declaring notifications that they are willing to send. A generated event notification is never addressed for a specific receiver but is propagated to any component that has

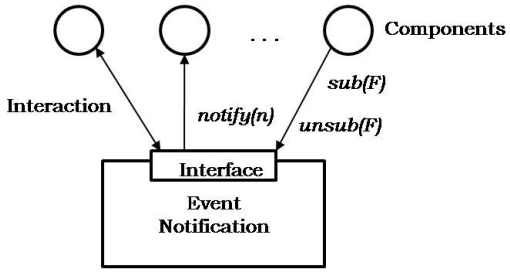


Figure 2.1: Simple Event-Based System

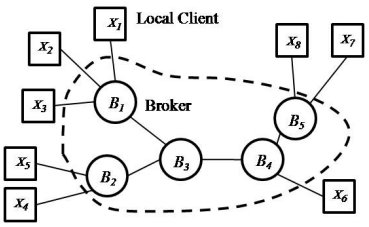


Figure 2.2: The Router Network
(Representing Event-Broker Network)

declared interest in receiving it. The generation and propagation are asynchronous operations. Due to this nature of operations, event-based systems are also known as publish/subscribe systems.

Event brokers within a network are responsible for handling this asynchronous, decoupled communication. They are connected to publishers and subscribers and act as event dispatchers who receive published notifications and propagate those notifications through the network and deliver them to interested subscribers.[Fiege et al., 2002, Carzaniga et al., 1998] The events are propagated to the subscribers via a network of event brokers.

2.2 Features

Some of the prominent features of event-based systems (a.k.a publish/subscribe systems or in short, pub/sub systems) are described below. These features make these systems candidates for wide area applications which require processing of large amounts of data without much interaction between client and server. These features can also complement a few requirements for todays growing mobile systems.

2.2.1 Asynchronous Communication

Event-based systems are asynchronous. Publishers and subscribers need not have their clock generators synchronized which means they can have variable bit rates. This asynchronous nature allows communication between heterogeneous components.

2.2.2 Strong Decoupling

Publishers and subscribers need not be connected with network at the same time. Publisher can publish messages and go offline without being concerned about subscribers. Subscribers should be online to receive any generated notifications, however some systems can provide disconnectedness, where subscribers will receive notifications when they come online.

2.2.3 Anonymity

The publishers and subscribers in event-based systems are usually not aware of each other. Subscribers subscribe for interested events and publishers may advertise what they are going to publish. Once a publisher publishes some event, it is routed through the network of brokers which match the events with subscriptions, and accordingly notify the subscribers. Thus, the publishers need not know addresses of subscribers in advance and vice versa.

2.2.4 Scalability

Naturally, asynchronous communication and decoupling implies scalable systems, since each components is concerned about its local environment and not the overall system. For example, a subscriber is concerned with his subscriptions and the event broker he is connected to. An event broker is concerned with publishers, subscribers and other event brokers directly connected with the broker. Computations are done at the brokers/servers specialized for such operations and only messages are passed over the system. This makes the system very scalable and able to handle massive number of heterogeneous components even with current communication channels.

2.2.5 Content-driven Interaction

All the communication happens according to the contents of the event advertisements, notifications and subscriptions. Through the contents of the message, subscribers specify their interest in specific events, publishers specify events they will be publishing for and the notifications are forwarded through the routing network. Thus, we have all the interaction between all the components by means of the contents of messages.

2.3 Types of Event-Based Systems

We can find three prominent types of event-based systems based on the filtering they use to process the data/information about an event. They are explained in brief as following.

2.3.1 Topic-based Filtering

Topic-based publish/subscribe system is similar to the notion of a group. Subscribing to a topic is equivalent to becoming a member of a group and publishing events related to that topic is equivalent to broadcasting the information to group members. It is static scheme, easy to understand and offers portability (strings used to divide event space). Hierarchies, wildcards are some of the variations. [Eugster et al., 2001]

2.3.2 Subject-based Filtering

Subject-based filtering uses string matching for notification selection. In this case, the publishers annotate each notification with a subject string which denotes a rooted path in the tree of subjects. For example, a publisher may want to publish new notification regarding Deltas stock exchange quotation under the subject /Exchange/USA/NYC/Technology/Delta. It gives information about a trade in NYC, USA, belonging to technology sector of the stock exchange. The clients who subscribe for /Exchange/USA/NYC/Technology/* to get all technology quotations will receive this notification. In general, arbitrary pattern matching can be applied on the subjects. This is a very simple approach but is inadequate to handle large amounts of data. The requirement of using a single path in the tree to classify a notification severely constrains expressiveness.[Oki et al., 1993]

2.3.3 Type-based Filtering

Type-based publish/subscribe system is a newly developed system which features encapsulation and type safety. This system improves upon the subject-based filtering, and uses path expressions and subtype inclusion tests to select otherwise opaque notifications.[Eugster et al., 2001] The subject tree is extended through multiple inheritance, to allow paths at different roots to connect to the same node(s).[Eugster et al., 2001, Oki et al., 1993]

2.3.4 Content-based Filtering

The expressiveness of notification service is determined by the language used for specifying subscriptions and the data model for transmitting notifications. The most flexible notification service for this purpose is content-based filtering. Filters are boolean functions on entire contents of the notification and are used to create subscriptions specifying the constraints on notifications.[Fiege et al., 2003] Then, used with the name/value pair data model, the subscription is quite easy to express. For example, *give notification when traffic on I-85 near downtown Atlanta is heavy* can be subscribed by the client with *service = traffic & location=downtown Atlanta & street=I-85 & traffic_delay >15 minutes*. When traffic near the specified location has a delay of 15 minutes or more, notification will be delivered to the client.

Since the expressiveness of content-based filters is impressive, and a combination of two or more filters can be used to combine information about two or more services together (for example, traffic information + weather updates), clients have a great deal of flexibility for expressing their interests. Therefore, we have chosen a combined broadcast content-based routing scheme [Carzaniga et al., 2004] for the study which is based on content-based filtering. Thus, whenever we use the expression event-based system, it is assumed that we are discussing a system using content-based filtering.

2.4 Challenges for Event-based Systems

Research in the field of event-based systems to this point, helped to develop systems with basic frameworks. There are several areas which are the focus of ongoing research. Some of them are discussed in this section in brief.

2.4.1 Security

Some of the security concerns in event-based systems are similar to those of distributed systems. Some differ from traditional middleware security because of publish-subscribe com-

munication semantics. Much of research has been done in this field and is still in progress. First, generic issues will be discussed, and then specific issues related to pub-sub communication will follow.[Wang et al., 2002]

Generic issues: *Authentication*, assuming the trust in contents of the message, refers to a sender and a receiver being sure about the originator of the message. *Information integrity* deals with the integrity of message by means of digital signature to ensure that contents of the message are not tampered with and the authenticity of the sender. *Service integrity* refers to malicious faults arising at servers introducing bogus information in the server. *User anonymity* is primitively covered by communication semantics in these systems by keeping only local information.

Specific issues: *Confidentiality* can be further divided into three types. Information confidentiality means publishers and subscribers want to keep information secret within the infrastructure by operations such as encryption. In subscription confidentiality, a subscriber expects infrastructure to keep his subscription secret. Publication confidentiality refers to keeping publications secret from illegitimate subscribers. *Subscription integrity* deals with the protection from unauthorized modifications since subscriptions form the basis for routing and forwarding. *Accountability* comes into the picture when a publisher wants to charge subscribers for information they are going to publish. *Availability* in event-based systems is most vulnerable to denial of service attacks by means of malicious publications and subscriptions to overload the system.

2.4.2 Fault Tolerance

Like any other system, event-based systems expect the system to work correctly in the presence of faults. If a publisher goes down, some events may not be reported to the subscribers on time. If a subscriber becomes faulty, the system should still deliver all the events of his interest for which notifications are generated while the subscriber was down after it is repaired. A

faulty event broker will be a major concern since it can result in loss of messages. Two conditions in the formal specification of event-based systems are *safety*, which ensures that nothing bad will happen and *liveness*, which stipulates that eventually something good will occur. To satisfy these correctness conditions, faults in the system must be masked. Recent research in this field points toward alternatives such as self-stabilization.[Mü et al., 2006]

2.4.3 Congestion Control

The assumption made by many that event-based systems messages are negligible in size and are not likely to saturate a network is not true in reality. Consider a simple situation in which there are not enough resources at the event broker to sustain the rate at which event messages such as published notifications, subscriptions and notifications to be routed are received. This can be termed as *event broker congestion*. *Network congestion* occurs when network bandwidth between event brokers is limited. Both conditions can lead to loss of messages which can further increase congestion by retransmission in case of guaranteed delivery semantics. Possible causes of congestion can be under provision of resources deployed in the network and a temporary need for more resources due to a sudden increase in messages.[Mü et al., 2006] Research done by P. R. Pietzuch and S. Bhola discusses a few options for scalable communication mechanism such as PDCC and SDCC [1][Pietzuch and Bhola, 2003]

2.4.4 Mobility

Event-based systems do not consider mobility by default. Some of the research dedicated toward guaranteed delivery semantics present partial solutions for mobility issues. *Physical mobility* includes handling disconnectedness when a component (moving or non-moving) looses its network connection due to circumstances such as faulty resources or out-of-network coverage area whereas location-based services having mobility awareness are

[1]PDCC is PHB-driven congestion control algorithm where PHB means publisher-hosting broker SDCC means SHB-driven congestion control algorithm where SHB means subscriber-hosting broker

included in *logical mobility*. Most of the existing applications in a static as well as dynamic environment are mostly concerned with physical mobility, while applications having mobility awareness, due to recent advances in mobile systems, are concerned mainly with logical mobility.[Zeidler and Fiege, 2003] This issue will be further discussed in the next chapter.

2.5 Related Work

Much work has been done in the field of event-based systems recently and progress is still continuing. Some of the notable publish-subscribe systems will be listed in the first part of this section, stating their filtering style, communication model, working and some features in brief. The contents are taken from a survey that compares a few event-based systems. [Liu and Plale, 2003] The second part will enlighten recent research within event based systems.

2.5.1 Examples of Publish-Subscribe Systems

JMS

The Java Message Service (JMS) is a vendor-agnostic messaging standard that allows applications to create, send, receive and understand messages. Just as JDBC provides APIs to application developers to access many different systems, JMS defines of a common set of interfaces that can be used by different Message-Oriented Middleware (MOM) vendors. It provides two types of messaging models, publish-subscribe and point-to-point queuing. In the publish-subscribe model, JMS uses topics as intermediaries, which include subject-based systems. One producer can send a message to many consumers through a virtual channel called a topic. It can be implemented as a Client-Server model, Peer-to-Peer model or both. in the JMS publish-subscribe messaging model, publishers have to create subscriptions for clients to subscribe while clients have to be active to receive message, but it also has an option for durable subscriptions, allowing clients to disconnect and later reconnect and collect messages that were published while they were disconnected.

Gryphon

Gryphon deals with the distribution of large volumes of data in real-time to thousands of clients distributed throughout a large public network. It is a content-based publish/subscribe system based on client-server model. Events are matched to subscribers through a matching tree that is constructed in the pre-processing phase. The nodes of the matching tree are expressions tha are evaluated over attributes, and the edges are traversed as a result of evaluation. While moving toward leaves from the root, conditions are evaluated at a higher level and refined at lower levels. Thus the leaves are individual subscriptions. The *Link matching algorithm* is used for multicasting. A full suite of security features for client authentication including – simple (telnet-like) password, mutual password authentication, asymmetric password certificate SSL authentication and symmetric certificate SSL are provided by Gryphon. Access Control Lists (ACLs) are used to limit the topics to which an authenticated client may publish messages or subscribe. The Gryphon system has been deployed by IBM over the Internet for real-time sports score distribution at the Grand Slam Tennis events, Ryder Cup, and for monitoring and reporting statistics at the Sydney Olympics.[IBM Gryphon, 1997]

Bayeux

Bayeux is a protocol developed for transporting asynchronous messages (primarily over HTTP), with low latency between a web server and web clients. Thus it is an efficient application-level multicast system. It is scalable and tolerant for failures in routers and network links. It is based on Tapestry – a decentralized peer-to-peer architecture in which each Tapestry node can assume the roles of server (which stores and serves objects), router (which forwards messages), and client (which serves as originator of requests). Some clients act as publishers and publish event messages to servers. The servers and routers acts as brokers which send/forward event messages to clients that act as subscribers . Its matching algorithm has similarity with *hashed-suffix mesh mechanisms*. While based on Tapestry,

Bayeux provides application level multicasting protocol to organize receivers into a multicast tree rooted at the source. [Liu and Plale, 2003, Russell et al., 2007]

NaradaBrokering

NaradaBrokering is a distributed event brokering system for wide area applications that require many distributed cooperating brokers. It is a content-based publish subscribe system. Within a cell, it has a hierarchical topology of servers for event dissemination while between the cells, it has peer to peer graphs. Again, the matching is achieved by constructing a matching tree from the content of subscriptions. When an event arrives, the matching tree is traversed to locate matched subscribers. NaradaBrokering also provides additional matching engines for SQL 92 queries based on the JMS specification and XML attribute-value pairs for topic subscription. Routing is based on shortest path computations. NaradaBrokering tries to provide a unified messaging environment that integrates Grid Services, JMS and JXTA. NaradaBrokering supports multiple underlying transport protocols, including TCP, UDP, RTP, SSL, and HTTP. The security framework is under development. [Liu and Plale, 2003]

XEvents/XMessages

XEvents/XMessages is a unique kind of event-based system in the sense that it is a hybrid subject-based and content-based publish-subscribe system and is roughly based on a client-server model. Clearly XMessage has characteristics of both systems. XMessages provides a framework for reliable application level messaging and XEvents is built on the top of XMessages framework. XMessages implementation provides a simple to use approach to send and receive XML-based application messages that use SOAP 1.1 and WSDL. The use of XML for data representation format gives XMessages flexibility in platform and language independence. It uses a lookup table to get the reference of XMessage channel for building the connection and then, it can use an SQL-like query to filter messages from the channel in which filtering is based on the content of messages. It uses a *filtering matching algorithm* in

which simpler predicates can be applied to select attributes of a message, and more complex predicates may result in the execution of an SQL-like query. The subscriber and publisher can interact with each other directly through an event channel which is the traditional form of communication in publish-subscribe systems or communicate with each other via third parties, or agents. XMessages is a reliable event service in the sense that when the network is down and target nodes are unreachable, the service will automatically keep trying until the network connection is up and running and maintains a log of messages not yet delivered, so messages are not lost if they cannot be delivered immediately.

Siena

Siena is a content-based scalable event-notification service. It is a client-server model in a sense that two types of clients, publishers and subscribers, exchange messages through a Siena server, event broker. Publishers connect to nearest event broker to publish events they want to publish, and subscribers connect to the brokers for subscriptions, in which they specify the set of messages they are interested in receiving. Matching is accomplished at the broker with a *Binary Decision Diagram* which is based on the constraints specified in the subscriptions and evaluated based on contents of the messages. To use the system, a subscriber must implement a *notifiable* interface, so Siena delivers notifications to a subscriber by invoking the notify method on the subscriber object. Siena's routing paths for notifications are established at the time of subscription. A new subscription is stored and forwarded from the originating server to all servers in the network. This process forms a tree that connects subscriber with servers, and the notifications are then routed toward the subscriber following the reverse path of the tree.

JEDI, REBECA, Herms, Cambridge Event Architecture (CEA), Elvin, READY are examples of more event-based systems. We will discuss REBECA while dealing with mobility.

2.5.2 Recent Work in Event-based Systems

Event-based systems with publish-subscribe framework has become an important data communication paradigm in recent years or rather in last decade.[Chen et al., 2009] Though there is some research, focusing on the mobility in event-based system, but almost none regarding our advanced semantics, there is some research done and still going on other areas within event-based system. This particular section will discuss some of that recent work in event-based system.

Due to a large amount of data, achieving scalability but still maintaining high performance is an important issue in event based systems. In *Parallel Event Processing for Content-Based Publish/Subscribe Systems* , authors introduce a parallel matching engine which leverages current chip multi-processors (CMPs) to increase throughput and reduce notification-subscription matching time. They try to parallelize the matching engine by (i) assigning threads to events and executing the entire matching for their respective events in case of multiple events independent processing, (ii) assigning threads to specific parts of the matching process of a single event in case of single event collaborative technique and (iii) combining first two techniques hybrid technique.[Farroukh et al., 2009] Another technique for handling large amount of data is indexing. MICS (Multidimensional Indexing for Content Space) is one of such techniques. [Jafarpour et al., 2009] According to the authors *MICS creates a one dimensional representation for publications and subscriptions using Hilbert space filling curve*. MICS makes use of B+ Trees for organizing subscriptions which helps in determining covered/subsumed/none relationships. A matching algorithm also uses same to improve this process. Multiple techniques such as these two can be merged to achieve more efficiency in scaling the large amount of data.

Typically, all the subscriptions are equally important, but at the same time chances of subscriber receiving overwhelming number of notifications are high. A possible solu-

tions is ranking the subscriptions so that only notifications matching top ranked (most important) subscriptions are delivered to the subscriber when information flow is very high.[Drosou et al., 2009] offers such a solution by examining number of different policies (periodic, sliding window, and history-based). The approach is implemented and tested using Siena.

Another challenging issue in event-based systems is redundant messages as well as incomplete messages. Most of the existing event-based systems assumes that event data is clean, complete and accurate without presence of any noise. This may be true for electronically generated data by servers, sensors, etc. but may not be valid for event-based systems with human participants. The event data can be incomplete, insufficient or worse, malicious. Authors of [Chen et al., 2009] propose a novel algorithm *Agele* which aggregates event information scattered across multiple messages generated by different publishers, and eliminates redundant event messages. Agele's architecture is based upon a distributed broker overlay and the message brokers interact with one another in peer-to-peer fashion. Agele's biggest advantage is decentralized processing. Inaccuracy in the event data can occur due to several reasons such as communication errors, timing errors due to lack of global clock in distributed systems. [O'Keeffe and Bacon, 2010] proposes a solution providing an extension to a complex event language that allows programmers to specify various detection policies. These policies can help the detectors to tolerate a variety of errors to a certain acceptable degree.

Event-based systems are much more effective in utilizing sensor data. Sometimes, single sensor data is sufficient for an application, but for complex events applications are concerned with aggregation of data from different sensors. Some of the recent studies acknowledge this problem. Distributed Constraint Processing (DCP) by [Mao et al., 2008] is one example of the proposed solutions. Quality of Service a.k.a. QoS, is one of the important issues

in event-based system.[Mahambre and Bellur, 2008] proposes an adaptive approach for ensuring reliability (as QoS guarantee) in event-based middleware. This study focuses on providing reliability by setting up routes for different packets that guarantee requested reliability by a subscriber on an event broker network. Route determination is done in an adaptive fashion as per proposed Adaptive Reliability (AR) algorithm. This algorithm tries to minimize number of messages transmitted during route-establishment, guarantees reliable event notification delivery to subscribers and use reliability-estimates to refine route quality adaptively. Authors of [Keeney et al., 2008] concentrate on extending Siena Content-based Network by increasing flexibility of subscriptions and publications. The authors call the new network with these features as Knowledge-based Network. Two extensions are added to the system. The first one provides ontological concepts as an additional message attribute type, onto which subsumption relationships, equivalence, type queries and arbitrary ontological subscription filters can be applied. The second extension provides bag-types to be used allowing bag equivalence, sub-bag and super-bag relationships to be used in subscription filters.

Some studies do not focus on contributing toward *adding new features* to the pub/sub systems but on making use of pub-sub style to achieve different goals.[Hu et al., 2008] using pub-sub framework for distributed automatic composition of different services in large scale systems. In this mechanism, authors propose service inputs to be modeled as subscriptions, outputs as advertisements, and service interfaces mapped as publish/subscribe messages which are used in such a ways that matching can be used to evaluate service compatibility. In today's dynamic environment, receiving notifications about runtime information (e.g. sport events, stock exchange, etc.) is important. [Michlmayr et al., 2008] argues that SOA runtime environment should be able to address this issue. Authors first introduce event notification support in SOA runtime environment, which includes event types, participants, representations, as well as ranking, correlation, subscription, and notification mechanisms. Complex

Event Processing (CEP) is a concept in which the goal is to find composed events. CEP has many useful applications in areas ranging from agile business and enterprise processes management, financial market applications to active Web and service oriented computation. Ad-hoc semantics demands use of formal logical semantics for event-based systems. Many logic-based approaches for CEP based on formal semantics fail because they are not able to compute complex events in data-driven fashion. The authors of [Anicic et al., 2009] propose an approach which enables both logic-based and data driven complex event detection.

Some research is also dedicated toward *adding new features* to the event-based system. Most of them are either very loosely supported or not supported at all. One of them is *predictive publish/subscribe matching*. Authors of [Muthusamy et al., 2010] propose an added ability to predict the likelihood of a subscription matching at some time in future. As authors stated that composite subscriptions consisting of temporal and logical operators are efficiently represented by a set of finite state machines and rules. The proposed algorithm utilizes Markov model according to application's event workload, and predicts the probability of a subscription that will match within certain time period in the future event stream.

The research in the emerging field of event-based systems is not constrained by different studies mentioned above. There is a vast number of subfields within event-based systems. Some of them are mentioned in this chapter along with brief introduction to the related work. Some research is still going on to improve available functionality, for example, our work in providing mobility in combined-broadcast and content based systems. There are certainly new areas to expand which have not came into picture yet or are being worked on. Our advanced semantics is one of such new areas of interest which will be discussed soon in upcoming chapters.

Combined Broadcast Content-Based (CBCB) system which is used for this study and Siena overlap with each other in the in terms of their working i.e. filters, routing, notifications, subscriptions, etc. They are developed by the same group of people and are based on the same notion (i.e. combination of broadcast protocol and content-based protocol). All the systems mentioned above have specific characteristics but none of them handles mobility and the advanced semantics mentioned before. Thus we choose CBCB Simulation to implement mobility and observer its effects, as well as implement statistics maintaining histograms to support flexibility in event notification.

Chapter 3

MOBILITY IN EVENT-BASED SYSTEMS

Simple event based systems by default does not provide support for mobility but recent research shows that it is a powerful tool which can be used to support mobility in a pervasive environment.[Cugola and Jacobsen, 2002, Mascolo et al., 2002b, Meier and Cahill, 2002]

3.1 Mobile Systems and Event-based Systems - Overlapping

Mobile systems are distributed systems in which a subset of components/nodes or all of them are mobile. Recent advances in wireless networking technologies and the growing success of mobile computing devices, such as laptop computers, third generation mobile phones, personal digital assistants, watches and similar electronic devices are enabling new classes of applications that present challenging problems to designers. The range of mobile computing applications includes but is not limited to wireless networks, location-based services, sensor networks, and ad hoc networking. A combination of two or more of these applications and/or combination with other sources of data can give more power for information dissemination. For example, location-based information can be combined with weather reports, information on traffic jams, or availability of free parking spaces. In such cases, the system can propose routes that avoid places in which traffic is high or weather conditions are unpleasant, or the system can direct the driver to the nearest free parking space.[Mascolo et al., 2002b, Cugola and Jacobsen, 2002]

Some of the features of event based systems match perfectly with the requirements of mobile systems. Temporary loss of connectivity, moving from one location to other, low battery power, slow CPU speeds, small memory, frequent and unannounced changes in environment, such as highly variable network bandwidth, and overloaded resources are some of the challenges faced by mobile systems. These challenges demand decoupled and asynchronous communication between various heterogeneous components of the dynamic topology network. Middleware that utilizes an event-based communication model is well suited to address the requirements of the mobile computing domain as it requires a less tightly coupled communication relationship between application components compared to the traditional client-server communication model.[Mascolo et al., 2002b, Meier and Cahill, 2002]

3.2 Requirements Vs. Features

There are many challenges faced by a mobile systems. We can point out intermittent connections, low computing power, and dynamic context as most prominent. This section will discuss requirements of mobile networking along with features of event-based systems complementing those.[Mascolo et al., 2002b]

3.2.1 Intermittent connections Vs. Asynchronous communication

Today, mobile devices such as low powered mobile phones/PDAs or even high powered laptops are frequently connected to the network mainly for accessing data (weather reports, stock exchange information, etc.) or requesting some service (guidance for driving by utilizing information about current traffic situation, etc.). During these connection periods, available bandwidth is usually much lower than what we find in fixed systems. As a client moves, bandwidth varies. For example, consider your laptop moving within a wireless router's range. As you go away from router, connection speed you experience degrades. As you come closer to the router, connection speed boosts up. Sometimes this available bandwidth drops down to zero and shows no network coverage. Often, it can be observed that a client requires some

service, but the server that delivers it is not connected. In order to allow communication between such components, an asynchronous form of communication is necessarily. Components in an event-based system are decoupled and use asynchronous communication, so they are perfect for a mobile system.[Mascolo et al., 2002b]

3.2.2 Low powered mobile devices Vs. Use of event brokers

Mobile applications usually operate on resource-scarce devices with low computational power, slower CPUs, less memory, low battery power, etc. Therefore, computations should be performed on resourceful devices. Event brokers can be used for this purpose. Thus, client devices will only be concerned with generating and receiving notifications, but propagation of those notifications can be optimized by event brokers that form an overlay network. For example, cell phone towers can be used as event brokers which handle the communication between all the cellular devices within the cell as well as the communication between other towers. We can place high powered devices with these towers for computational purposes.[Mascolo et al., 2002b]

3.2.3 Dynamic context Vs. Physical and Logical connections

Mobile systems execute in highly dynamic environments. Variability of network bandwidth and heterogeneity of mobile devices have been already discussed above. When a device is moving, it is physically disconnected from one server and connected to another. Some of the services require location-based awareness within middleware. For example, finding a free parking space within a two-block radius distance of the current location of a moving car. In such a case, a device is physically connected to the same server, but logically it is moving. Such kind of awareness is desirable in the middleware. Event-based systems can handle physical as well as logical connections.[Fiege et al., 2002]

3.3 Issues for mobility in event based systems

Though event-based systems are good for mobile systems, they do not incorporate mobile systems by default. The analysis of basic issues for providing mobility in event-based systems leads to two orthogonal forms of mobility – physical mobility and logical mobility. *Physical mobility* is mainly concerned with unwanted effects of moving components in existing applications, whereas *logical mobility* is concerned with the advanced and upcoming applications which have location-awareness.[Zeidler and Fiege, 2003] Many of the existing applications in event-based systems prefer mobility support with location transparency in middleware, so they do not have to deal with changes in the existing interface(s). For example, one of the main concerns with the mobility in existing event-based systems is disconnectedness. If a client moves from vicinity of one event broker to another's, all of its subscriptions has to be transferred to new broker and it must be ensured that all the notifications after moving client will be delivered to the new broker and not the old one, possibly avoiding duplications. Also, while providing mobility in the middleware, the quality of service must not degrade substantially. It is obvious that changes have to be made in middleware to support mobility which may increase the cost of normal operations.

Due to recent advances in wireless technologies, future applications might not want to be concerned with complete location transparency and may have location-awareness. In other words, applications have to deal with unwanted phenomenon such as disconnectedness as well as they have to facilitate location-awareness for location-based services (for example finding a free parking spot within a two-block radius). Notifications can be used to convey location changes manually, but it will be convenient for the applications if infrastructure could adapt to location changes automatically and then middleware could handle the necessarily changes in routing the notifications accordingly. Today, extending pub/sub systems to handle location-awareness is a promising issue.

3.4 Related Work in Mobility

Even though research in mobility in event-based systems is increasing due to a need to process a massive amounts of data and provide information to low resource devices, there are not many pub/sub systems supporting mobility. Some of the research done by other people such as Huang and Garcia-Molina [Huang and Garcia-Molina, 2004] discuss a few possible ways to add support for mobility, but their work does not provide any concrete solutions. According to [Podnar and Lovrek, 2004], mobility support for an event-based system provided in Elvin [Sutton et al., 2001] is one of the first implementations. The solution suggested by the authors is to put a proxy server between the original server and the moving subscriber. Much like all the centralized systems, this approach creates a performance bottleneck. JEDI provides a specific mechanism for mobility that uses explicit operations such as moveIn and moveOut. Operations such as these two are an application level implementations and the middleware does not provide much support for them. Siena which is based on the content-based routing scheme, has a similar approach. Siena uses client proxies and a special client library to manage subscriptions on behalf of a subscriber and the move-out and move-in procedure.[Caporuscio et al., 2002] Some of the event-based systems such as [Yoneki, 2003] using JMS as basic framework implement mobility support by using lightweight JMS-compliant API for Java-enabled mobile terminals. But these implementations offer very little support for routing optimizations. Mobility support in REBECA [Zeidler and Fiege, 2003] has implemented support for physical mobility, and our implementation is loosely based on same notion. A possible approach for logical mobility using location dependent filters is discussed in [Fiege et al., 2003]. We are not considering logical mobility in the CBCB Routing scheme.

3.5 Events in Publish/Subscribe System

For understanding the process of moving a predicate through a network, one should have understanding of basic events in a pub/sub system. Thus, the first subsection will elaborate the process of subscription, the second subsection will explain the process of unsubscription, and finally, the third subsection will provide insight into the process of moving a subscription.

3.5.1 Subscription

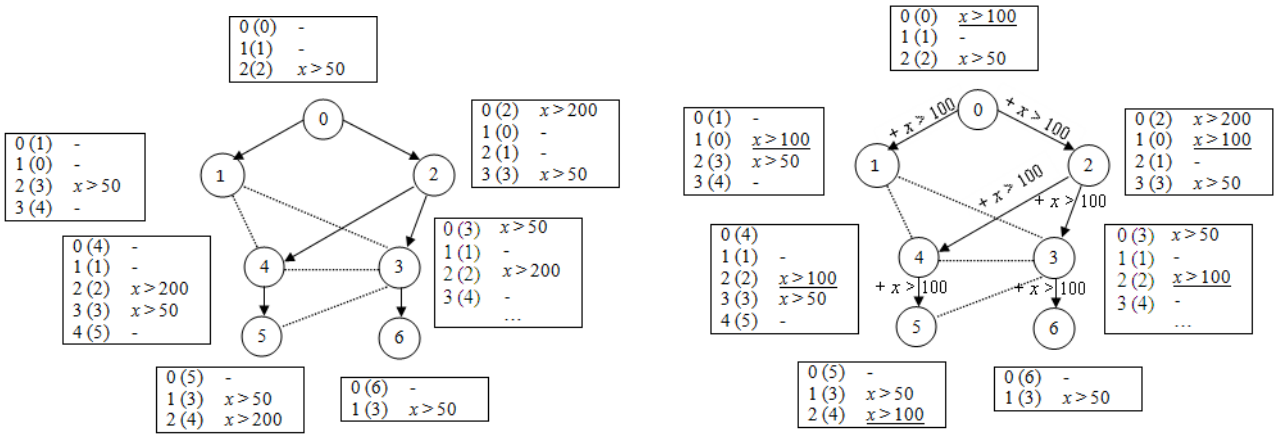
The process of a client expressing some interest(s) for receiving notification(s) related to some event is known as *subscription*. This section will clarify what happens in a content-based pub/sub system when a subscriber subscribes for some event. In content-based systems, interests are expressed by means of predicates. Some predicates might cover others, for example $x > 20$ covers $x > 50$. A simple approach would be a broker delivering each predicate to all of its the neighbors. This is also known as flooding, and has the worst performance of all the approaches. An improvement on this will be to check new predicates against all the predicates in the routing table for all the neighbors. If a match is found, there will be no need to propagate a predicate to that particular neighbor. This can still be improved by taking advantage of covering predicates. This approach considers the following cases whenever a new subscription arrives at a broker either from a subscriber or from a neighboring broker.

1. A new predicate is covered by some other predicate present at the current broker (This is possible when a subscription is issued by a subscriber at the current broker). The predicate is then simply dropped.

2. A new predicate covers one or more predicates at the current node/broker. In this case, all the covered predicates are removed, and the new predicate is added. The predicate is then propagated to all the neighbors.

3. A new predicate is neither covering any other predicate at current broker, nor is it being covered by any predicate. In this case, the new predicate is added to the predicate list of the current broker and propagated to all the neighbors.

The solid arrows form the edges in the broadcast tree for the currently subscribing node and dotted lines represent remaining connections in the network. The initial state assumes



(a) Initial State of CBCB Network

(b) After Propagation

Figure 3.1: Subscription process (Propagation of RA)

that a subscriber at node 3 has subscribed for $x > 50$ and another subscriber at node 2 has subscribed for $x > 200$. When a subscriber at node 0 subscribes for $p = x > 100$, p is propagated to its neighboring nodes 1 & 2. Since they do not have any predicates at interface for node 0, predicate p is added in the routing table at that interface. This predicate is then propagated to nodes 3 & 4 by node 2 (since they are children of node 2 in the broadcast tree rooted at 0). According to condition 2 specified above, we have $x > 100$ covering $x > 200$ and hence is replaced at both children. Node 3 then propagates the RA to its child node 6 where the predicate at interface 3 is covering the new predicate $x > 200$. Thus, p is simply dropped (Current example doesn't have any children at node 6 thus there is nothing to propagate to. But even if the node 6 was supposed to have children, still would not have propagated p).

3.5.2 Unsubscription

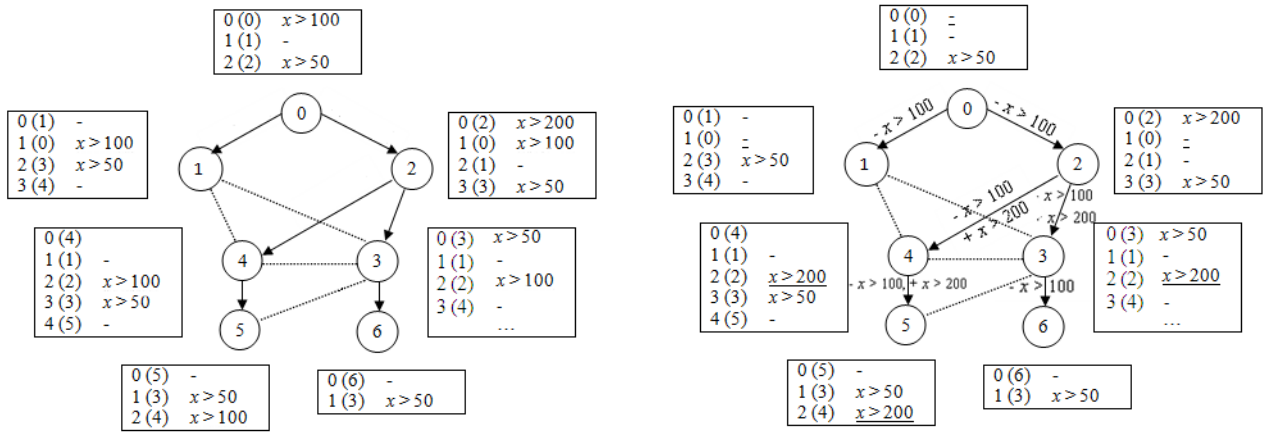
The process of a client removing his previously expressed interest(s) related to some event is known as *Unsubscription*. When a subscriber unsubscribes for some event, certain actions are necessary for ensuring correctness of the content-based pub/sub system. In one approach, when a subscriber unsubscribes for something, that subscription is simply removed without considering any other predicates previously present which were covered by the predicate that is being unsubscribed. Thus, if a subscriber first subscribes for predicates p_1, p_2,..., p_i and then for p_n which covers p_1 *to* p_i,, and then unsubscribes for p_n, there is no subscription present against that subscriber anymore assuming that when a subscriber subscribes for a predicate covering some previous subscriptions, he no longer needs older predicates. In another approach, when a subscriber unsubscribes for any covering subscription, all the previously covered predicates are resubscribed. In other words, in previous scenario, when subscriber unsubscribes for p_n, system will simultaneously resubscribe for predicates p_1, p_2,..., p_i. The first approach seems valid, and it can be implemented at event brokers where subscriber is actually connected. In case of event brokers, second approach is desired since each event broker acts as a subscriber to its neighbors but in actuality, the broker has multiple subscribers connected to it. This approach considers the following cases whenever a new subscription arrives at a broker either from a subscriber or from a neighboring broker.

1. A predicate that is being unsubscribed is covered by some other predicate that is present at the current broker (This is possible when a subscription is issued by a subscriber at the current broker). Unsubscription of this predicate is then simply dropped.

2. A predicate that is being unsubscribed covers one or more predicates at the current node. In this case, all the covered predicates are resubscribed, and the new predicate is removed. The predicate being unsubscribed is then propagated to all the neighbors along with the covered predicates to be subscribed.

3. A predicate that is being unsubscribed is neither covering any other predicate at the current broker nor is it being covered by any predicate. In this case, the predicate that is being unsubscribed is removed from the predicate list of the current broker and propagated for unsubscription to all the neighbors.

The solid arrows form the edges in the broadcast tree for currently unsubscribing node while dotted lines represent the remaining connections in the network. The initial state is



(a) Initial State of CBCB Network (b) After Propagation

Figure 3.2: UnSubscription process (Propagation of US)

assumed as shown in figure 3.2a which is same as figure 3.1 (b). When a subscriber at node 0 unsubscribes for $p = x > 100$, it is propagated to its neighboring nodes 1 & 2. These US packets do not contain any information about resubscribing for other predicates because there are no predicates being covered by p at node 0. Now, both of them have the same predicate at the interface for node 0 in the routing table, so predicate p is deleted from the routing table as well as from the issuer-predicate table. The predicate $x > 200$ issued by node 2 was covered by p and hence its resubscribing information is added to the US packets, and they are propagated to the children of 2 in the broadcast tree rooted at 0. According to condition 2 specified above, we have $x > 50$ covering $x > 200$, hence, there is no need to resubscribe for anything at node 6. After receiving US packet from parent 3, the node has nothing to unsubscribe for and drops the packet.

3.5.3 **Move Subscription**

This process handles mobility in event-based systems. A subscriber usually moves from one place to another. It is possible that he moves out of the range of one event broker and may or may not enter another's. If he is moving within range of other broker, it is necessary to move all of his relevant subscriptions from the previous broker he was connected to. Some subscriptions may be based upon locations and may not need to be moved if a subscriber moves to other location. Others which we may call *relevant subscriptions* have to be moved. Often this differentiation may not be required, and event-based systems may choose to simply move all the subscriptions from one broker to another. This approach may suffer by a heavy communication cost if the subscriber moves back and forth between few event brokers. In such a case, a sound optimization will be to keep copy of the subscriptions at all event brokers available at the places subscriber frequently visits.

Consider the simplest case of moving a subscriber from one broker to another without any kind of optimization. This action will involve (i) resubscribing for all the subscriptions made by the subscriber at the broker he is leaving at the broker he is moving to and (ii) unsubscribing for the same at old broker. Few approaches are possible for this process. One will be a subscriber notifying the event broker that he is going to move at, and he carries all of his subscriptions with him while moving. After moving to new broker, he can subscribe for those again at this broker and ask the previous broker to unsubscribe for the same. In real world, a subscriber may not know about range of event brokers (For example, cell phone towers: we never bother where the boundaries are for current cell or whether we are going to enter another cell. As a matter of fact often user does not even know about such things). Thus, it is highly unlikely that a user will know whether he is moving from the range of current event broker to another. Hence another approach suggests that a subscriber should have some kind of ID for the event broker he is currently connected to and for himself, so as soon as he connects to a different broker, the system will acquire information about both

of these IDs. After getting required information, this new broker will contact the old broker asking for all the subscriptions for this subscriber, will subscribe for them and ask old broker to unsubscribe for them. This process will include following steps.

1. After a subscriber is entered within the range of a new broker, the broker obtains information about the subscriber and previous event broker he was connected to. If there is no previous broker, then there are no subscriptions to be moved.

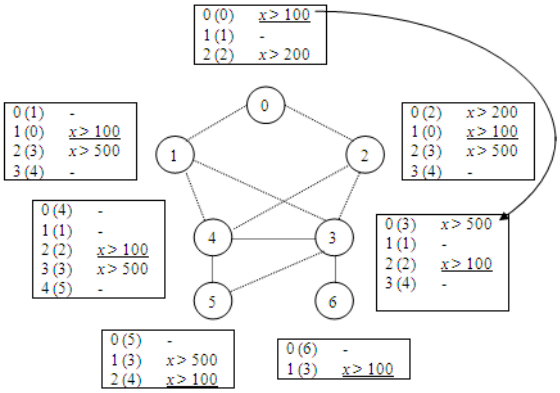
2. Contact the previous broker and ask for the subscriptions from the current subscriber. It is unlikely not to have a single subscription at this previous node unless the subscriber decided to unsubscribe for all just before moving.

3. Subscribe for all the subscriptions received from the previous broker.

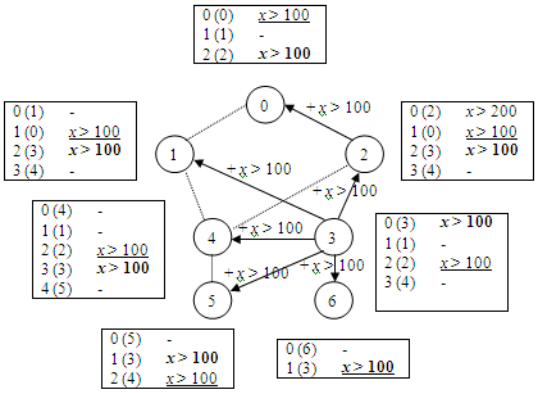
4. Ask the previous broker to unsubscribe for those subscriptions.

The solid arrows form the edges in the broadcast tree for currently unsubscribing node while the dotted lines represents remaining connections in the network. Figure 3.3 shows the process of moving a subscription. The initial state is assumed as shown in figure 3.3a which indicates that predicate $p = x > 100$ will be moved from node 0 to node 3. The predicate at node 0 is indicated by underlining wherever it is propagated. When the broker at node 3 realizes that predicate $p = x > 100$ is being moved from node 0, the broker subscribes for p. This broker already has a subscription $p' = x > 500$ but which is now covered by subscription being moved. Moved subscription at node 3 is indicated by boldface wherever it is propagated (figure 3.3b). After subscribing for p, the new broker tells the old broker to unsubscribe p. For simplicity, figure 3.3c depicts the unsubscription after complete subscription has finished. In reality, both of these operations might happen simultaneously.

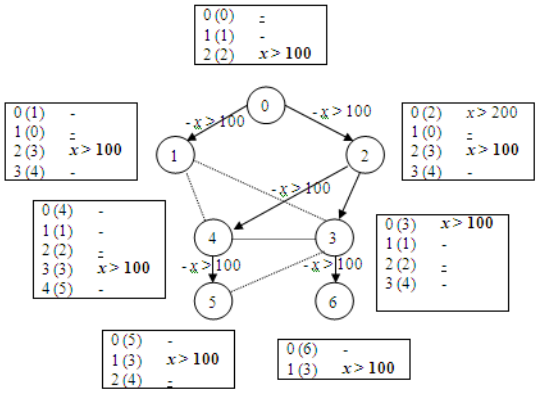
We have discussed mobility requirements, the ability of event based systems to handle those requirements; issues involved in introducing mobility in event-based systems; and



(a) Initial State of CBCB Network



(b) While Moving



(c) After Moving

Figure 3.3: Move Subscription process (Propagation of US)

related work in the field. The next chapter will discuss the CBCB Routing scheme and implementation of mobility in CBCB networking.

Chapter 4

IMPLEMENTING MOBILITY IN EVENT-BASED SYSTEMS

In the following sections, we will discuss **Combined Broadcast and Content-Based** (CBCB) **routing** as well as CBCB-simulation infrastructure.[Carzaniga et al., 2004]

4.1 CBCB Routing Scheme

In CBCB routing, the content-based layer is superimposed on traditional broadcast layer. The advantage of this scheme is that each message in the broadcast layer is handled as a broadcast message but at the same time, the content-based layer prunes the broadcast distribution paths, limiting propagation of messages to those nodes which have expressed their interests by advertising predicates that match these messages. Thus the notification service forms an *overlay network* in the underlying system. The virtual overlay network runs some processes on different nodes in the underlying physical network. Processes communicate with each other by means of message exchange in the overlay network based on contents and actual propagation of those messages through the underlying network of physical nodes.[Carzaniga et al., 2004] The overlay network is usually formed between event brokers, but subscribers and publishers are connected to the brokers. Current CBCB Simulation, does not provide distinction between these components which means, a node can act as a subscriber, as a publisher, or both, and also as event brokers that perform all the computations necessary to propagate messages.

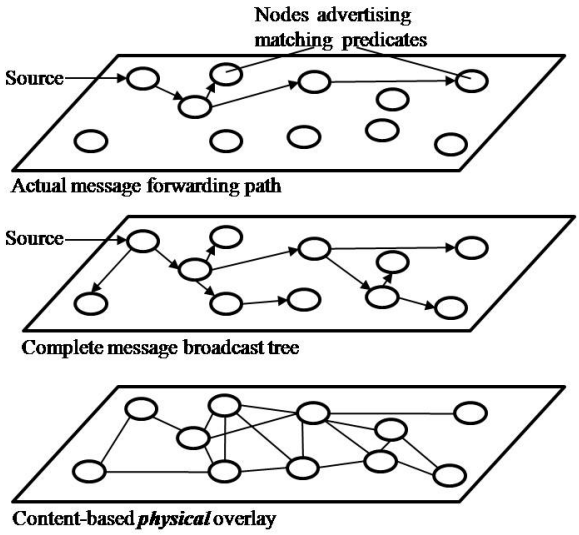


Figure 4.1: Formation of virtual overlay network above real network

4.2 Predicates and Notifications

In this content-based communication model, subscribers declare their interests using *selection predicates* and publishers simply publish *messages* that describe occurrences of the event. The "*message content is structured as a set of attribute/value pairs, and a selection predicate is a logical disjunction of conjunctions of elementary constraints over the values of individual attributes*". For example, a message may have content, such as

[id="falcon02010", event_name="Crash", vehicle_type="Honda Accord", street="I-85", position_latitude=33.79135, position_longitude=-84.39088]

which would match a selection predicate, such as

[id=* $\wedge$ event_name="Crash" $\wedge$ vehicle_type=* $\wedge$ street="I-85" $\vee$ street="GA-316"]

In this case, a subscriber is asking for crash reports on I-85 or GA-316 from anyone for any kind of vehicle. A publisher, with id falcon02010, publishes a crash event for a Honda Accord on I-85 along with the actual coordinates of the place of the event. Figure 4.2 shows the predicate structure used in CBCB scheme. A predicate consists of a set of filters, and a filter is made of a set of constraints. A constraint has a variable, an operator, and a value.

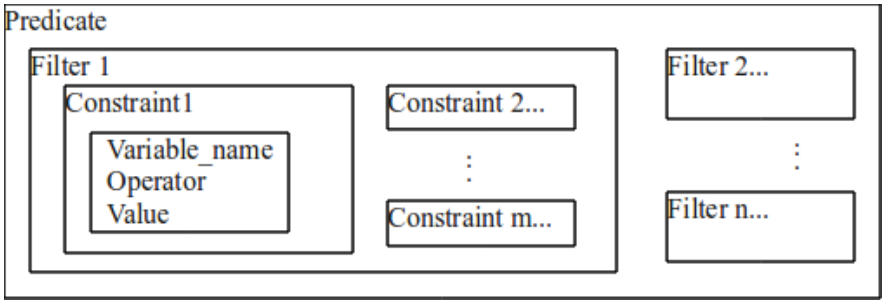


Figure 4.2: Structure of a Predicate in CBCB Simulation

Conjunction in a predicate is obtained by representing multiple constraints together in a filter, and disjunction is represented as collection of filters in a Predicate.

4.3 Application Programming Interfaces

We will discuss the few APIs already available in CBCBSim as well as new APIs provided for introducing mobility.

set_predicate: This API is used for content-based address of a node allowing to subscribe. Originally, the API was supporting one time subscription i.e. once a subscription is done, new filters/predicates would not be added. The idea behind this was *content-based address* in which each node has address as the contents of the predicates it is subscribing for. If a new subscription is performed at the same node, the new subscription was overwriting the previous subscription and thus, was changing the address of the node. This API been modified to support multiple subscriptions at a node at any time. Effectively content-based address notion is removed to some extent i.e. a content-based address can now be changed by adding more subscriptions, but we are not considering the notion for our problem.

send_message: Notifications will be generated through this API. Once a notification is generated, it will be propagated throughout the network using CBCB routing scheme until that notification reaches every node showing interest in receiving such notification. In other words, contents of message satisfies the conditions specified in the subscription done by the node(s).

remove_predicate: As the name suggests, this API does exactly the opposite of what a set_predicate does. Originally, this interface was not available but since the previous subscription was being overwritten, that process was as good as unsubscribing for previous subscription and subscribing for a new one. But what if a subscriber does not want to subscribe for a new subscription? Also, consider a subscriber who subscribes for events with a large number of filters and wants to add some more or remove some of its filters, subscribing for other duplicate filters will simply increase communication cost. Thus we have added this API. Right now, since there is no distinction between a subscriber and an event broker, we are assuming multiple subscribers each with distinct predicates connected to a node and this API is for that node acting as a subscriber on behalf of all those subscribers. To allow assumption of multiple subscribers at a node, currently, the predicate ID also acts as the subscriber ID. Thus subscriber is allowed to unsubscribe for the whole predicate and not the part of predicate.

move_predicate: This particular API is very specific to the problem we are dealing with i.e. introducing mobility in the CBCB routing scheme. We added this API to simulate moving (physical mobility) of a subscriber from one node to other. This API is designed by specifying from which node to which a subscriber is moving, along with the predicate ID for specific subscriber.

Node which could act as any of subscriber, publisher and event-broker will now act simply as a event broker with multiple subscribers/publishers assumed to be connected to one node. Though they are not physically present, logically each subscriber has separate ID given according to their subscription's predicate ID. It is possible to simulate different subscribers/publishers in the future by creating and adding those objects and keeping track of their IDs at the broker node they are connected to. This can allow multiple subscriptions by a single subscriber and/or multiple publications by a single publisher.

4.4 Basic Definitions and Structures

A *selection(p)* can be defined as set of all messages that have contents which satisfy a predicate *p*. A *covering* relation between predicates p1 and p2 can be defined as p1 *covers* p2 if selection(p2) $\subseteq$ selection(p1). Now onwards, p2 $\prec$ p1 will indicate that p2 is covered by p1, as given in [Carzaniga et al., 2004].

In CBCBSim, each node has a content-based routing table and a content-based forward table. A *routing table* is used for managing subscriptions while a *forward table* is used for notification propagation. The forwarding table is used for associating predicates to both external interfaces and local application. For convenience, 0th interface always belongs to the local application and remaining interfaces belong to neighboring nodes. The routing table and the forward table are conceptually the same (if the router table is updated anytime, the forward table is updated accordingly).

The remaining part of this section will discuss the structures of different packets (messages) used for the APIs discussed above. Originally only RA was given, but we have modified that API as well as provided new structures such as US, MI, MR, and MU for introducing mobility.

Receiver Advertisement (RA): These are issued by nodes whenever there is a new subscription. The structure of an RA packet is shown in figure 4.3. An RA packet contains an Issuer ID to verify identity of the issuer, and a predicate ID, which has different roles. In our case, it acts as a subscriber ID (for mobility). While unsubscribing, this ID is used to validate predicates. These RA packets are propagated through the network until either the nodes with covering predicates or the leaf nodes in the broadcast tree are reached.

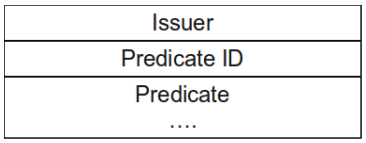


Figure 4.3: A Receiver Advertisement (Modified)

UnSubscription (US): These packets are issued by nodes whenever a subscriber wants to remove his subscription. Structure of a US packet is shown in figure 4.4. Similar to RA, an US packet also contains an issuer and a predicate ID which is used for validating the predicate. Predicate to be removed (*Remove Predicate*) consists of all the filters which needs to be removed from the corresponding predicate at the interface of the routing table of the current node. *Remove Plus Predicate* contains filters which need not be removed since they are covered. The sole purpose of propagating these predicates is to validate predicate which is being unsubscribed. *Set Predicates* contain the new predicates to be subscribed along with their issuers as a result of unsubscription of the current predicate. These predicates are required when the predicate being unsubscribed is covering some other predicate(s).

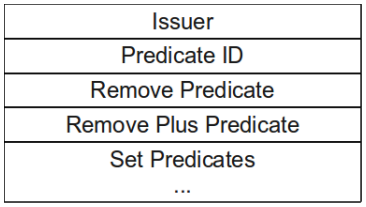


Figure 4.4: An UnSubcription Packet

Mobile Inquiry (MI): Figure 4.5 shows the structure of MI packets. These are issued by a node in which the subscriber subscribing for predicate with ID "Predicate ID" is moving to the "To Node". It contains ID for process running "From Node". This packet tells the "To Node" to do an inquiry at "From Node" to get the predicate with given ID. The idea behind this action is, when a subscriber moves from one broker to other, he should not have to subscribe again for his relevant subscription(s). Hence, when he moves to other node, subscriber will make the node aware of its presence by providing his ID (which is ID of the predicate in our case). The node will fetch that subscriber's subscriptions automatically by contacting its former broker.

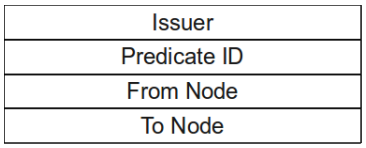


Figure 4.5: A Mobile Inquiry Packet

Mobile Response (MR): This packet is issued by the "From Node" responding to a Mobile Inquiry, telling the "To Node" to subscribe for the "Predi-

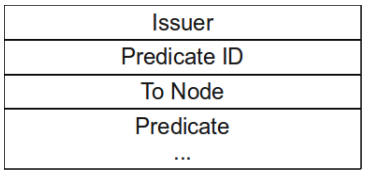


Figure 4.6: A Mobile Response Packet

cate" having a "Predicate ID". When the packet is propagated to destination event broker i.e. "To Node", that node immediately issues a subscription which is propagated in the same manner as a normal subscription (using RA packets) except that this subscription is not done through the API for subscription i.e. *set_predicate*. Along with this subscription, the "To Node" also issues unsubscription instructions for former event broker for current subscriber. Figure 4.6 shows the structure of MR packets.

Mobile Unsubscription (MU): This packet is issued by the node to which the subscriber is moved, telling "From Node" to unsubscribe for predicate(s) being moved. When this packet reaches the destination node, it immediately issues an unsubscription packet which is propagated in the same manner as a normal unsubscription, but this node does not use the unsubscription API *remove_predicate*. Figure 4.7 represents the structure of this packet.

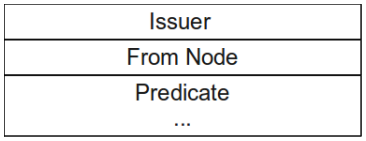


Figure 4.7: A Mobile Unsubscription Packet

Message Event (ME): This packet is the only untouched packet in the original CBCB routing scheme. As figure 4.8 shows, the packet consists of an "Issuer" verifying validity of the issuer of the message. A "Message" is sent along with its "Message ID" for logging purposes. The contents of the message are checked against the conditions specified by the filter(s) in the predicate(s) of interfaces while propagating packet. If conditions aren't satisfied by any of those predicates, then the message is simply dropped.

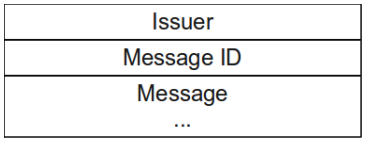


Figure 4.8: A Message Event Packet

4.5 Subscription and Unsubscription

RA and US are propagated through combined broadcast and content-based protocol.

RA Processing and Propagation

This has three main steps as explained below.[Carzaniga et al., 2004]

1. Predicate in the RA packet p_{iRA} received from interface i is checked against the predicate p_i in the routing table available for this interface. If p_{iRA} is already covered by p_i, it is simply dropped by the router.

2. If p_{iRA} is not covered by p_i, we add it to the p_i i.e. $p_i \leftarrow p_i \vee p_{iRA}$. Otherwise, if p_{iRA} is covering p_i then we replace p_i by p_{iRA} i.e. $p_i \leftarrow p_{iRA}$.

3. Using the routing table, next hop links in the broadcast tree rooted at issuer are computed and the RA packets are propagated to it.

This process, also known as *subscription*, is depicted by an example in figure 3.1.

US Processing and Propagation

This process has three main steps as explained below.

1. The predicate in the US packet p_{iUS} received from interface i is checked against the predicate p_i available for this interface in the issuer-predicate table. If the predicate p_{iUS} is not present in hp_i, it is simply dropped by the router.

2. p_{iUS} is then checked against predicate p_i in the routing table. If it is already covered by p_i, we still propagate the packet for correctness, indicating that the predicate need not be unsubscribed but the issuer-predicate table should be updated.

   i. If p_{iUS} is present but not covering any subpredicates in p_i, then we delete p_{iUS} from p_i. Issuer-predicate table is updated.

   ii. Otherwise, if p_{iUS} is covering some of the subpredicates in p_i then, we delete p_{iUS} from p_i and find the list of predicates covered by p_{iUS} along with their issuer. US packet is updated with this new information. The issuer-predicate table is updated.

3. Using this routing table, the next hop links in the broadcast tree rooted at the issuer are computed, and the US packets are propagated to the connected nodes.

This process, also known as *unsubscription*, is depicted by an example in figure 3.2.

The Find_Covered_Predicates algorithm shown in three different parts (algorithms 4.1, 4.2 and 4.3) portrays proposed implementation. The first part handles unsubscription requests for null predicates as well as for predicates which are covered by other predicates. Part two and three handle unsubscription requests for predicates which are covering some other predicates. The second part identifies which predicates/filters have to be resubscribed as a result of unsubscription and the third part prepares list of predicates/filters along with their issuers and interfaces which have to add those filters to their routing tables after unsubscription.

Algorithm 4.1 Find_Covered_Predicates Part 1

```
1: procedure FIND_COVERED_PREDICATES(unsubscribe_predicate, issuer)
2:     if unsubscribe_predicate.size() = 0 then
3:         allCovered ← TRUE
4:         return allCovered
5:     end if
6:     unsub_filter_set ← unsubscribe_predicate.getFilters()
7:     ifid ← 0                ▷ router_table is a table of interfaces vs. predicates against them
8:     while ifid < router_table.size() AND unsub_filter_set.size() > 0 do
9:         current_predicate ← router_table[ifid]
10:        for all Filters in unsub_filter_set do
11:            if current_predicate COVERS current_unsub_filter then
12:                REMOVE current_unsub_filter from unsub_filter_set
13:            end if
14:        end for
15:    end while
16:    if unsub_filter_set.size() = 0 then
17:        allCovered ← TRUE
18:        return allCovered
19:    end if
```

Part 1 initially checks for a null predicate for unsubscription. If the predicate is not null, then this part separates the unsubscription predicate into filters. All the filters in this set will be checked against all predicates present in the routing table at current node. If a filter

is covered by some predicate, then that filter is removed from the set. In the end, if the set becomes null, all the filters are covered which implies that the whole unsubscription predicate is covered at the current node, so there is no need for resubscribing for any predicate(s). If the set is not null, the algorithm will move forward to find out if there are any predicates which should be resubscribed.

Algorithm 4.2 Find_Covered_Predicates Part 2

```
20:  i ← 0                                  ▷ predicate_table is a table of issuer vs. predicates
21:  while i < predicate_table.size() AND unsub_filter_set.size() > 0 do
22:      current_predicate ← predicate_table[i]
23:      current_filter_set ← current_predicate
24:      for all Filters in current_filter_set do
25:          if uncovered_unsubscribe_predicate COVERS current_filter then
26:              ADD current_filter, current_issuer to filter_issuer_List
27:          end if
28:      end for
29:  end while
```

Part 2 will make use of the *predicate_table* to find out whether uncovered filters of unsubscription predicate are covering some other predicates/filters. For each issuer in the predicate table, all the filters issued by him are checked for coverage against *uncovered_unsubscribe_predicate* which is a predicate that consists of all the unsubscription filters which are not covered by any predicates in the *routing_table*. The filters which are found to be covered are added to a 'List' of issuers and corresponding filters.

Part 3 determines which predicates/filters should be added to the routing tables of the neighbors and have to be propagated to them along with the unsubscription. It is highly probable that some filters in the 'List' prepared in previous part will cover some of the other filters and/or some filters will neither cover nor be covered by others. Keeping these probabilities in mind, the list of filters along with their issuers is prepared against each neighbor of the current node and will be propagated to them.

Algorithm 4.3 Find_Covered_Predicates Part 3

```
30:     for all Neighbors in neighbor_nodes_set do
31:         for all Filters in filter_issuer_List do
32:             ▷ Neighbor need not resubscribe for his own subscription
33:             if current_issuer ≠ current_neighbor then
34:                 if successor_filter_set.size() = 0 then
35:                     ADD current_filter to successor_filter_set
36:                     ADD current_filter, current_issuer to successor_issuer_List
37:                 else if current_filter COVERS successor_filter_set then
38:                     REMOVE all filters from successor_filter_set
39:                     ADD current_filter to successor_filter_set
40:                     ADD current_filter, current_issuer to successor_issuer_List
41:                 else if successor_filter_set DOES_NOT_COVER current_filter then
42:                     ADD current_filter to successor_filter_set
43:                     ADD current_filter, current_issuer to successor_issuer_List
44:                 end if
45:             end if
46:         end for
47:         new_subscriptions[current_neighborID] ← success_issuer_List
48:     end for
49:     allCovered ← FALSE
50:     return allCovered, new_subscriptions
51: end procedure
```

4.6 Implementing Mobility in CBCBSim

We have already seen two types of mobility - physical and logical. In CBCBSim, we are considering only physical mobility since there is no support for simulating sensor devices needed for determining logical mobility. Introducing this support is out of the scope of the current study. Hence we are only considering moving of a subscriber from one event-broker node to other. Keeping in mind that subscribers and publishers are not simulated yet but assumed to be connected to the nodes acting as event-broker and every subscriber has an ID based on the ID of the predicate(s) it is subscribing for, the mobility inside cbcbsim is introduced as follows. Event brokers on the other hand are fixed (For example, consider

event broker application running on cell phone towers). Also, mobility of publishers is not considered since we are not going to use SR/UR protocol.[Carzaniga et al., 2004]

The process of moving a subscriber is depicted by an example in figure 4.9. A subscriber

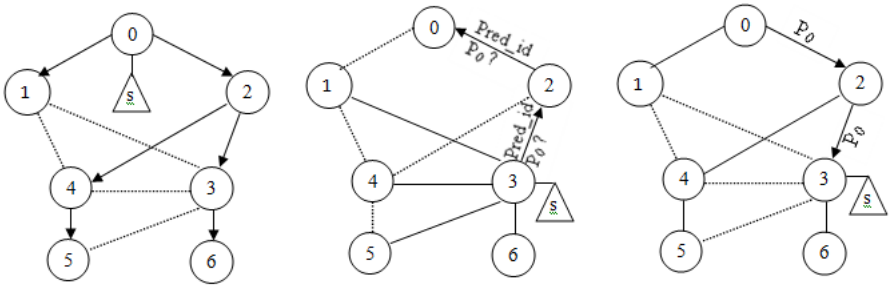


Figure 4.9: Initial Stages of Moving a Subscriber

s is moving from node 0 to node 3. After moving, he issues a packet MI inquiring about the predicate(s) it has subscribed for at node 0. Upon receiving the inquiry, node 0 sends the corresponding predicate within an MR packet along the reverse path. Once the node 3 obtains the predicate, it issues subscription packet for that predicate along with the MU packet to node 0 and instructs that node to unsubscribe for the predicate. After receiving MU, node 0 issues unsubscription for the predicate. Formally, this process can be explained as follows.

1. After subscriber s moves to new broker B_n, the new broker issues the MI_s packet asking for the predicate p_s subscribed by s at old broker B_o.

2. B_o issues the packet MR_s in response to MI_s containing p_s. This packet MR_s is propagated to B_n along the reverse path.

3. B_n receives packet MR_s and subscribes for p_s and also issues MU_s telling B_o to unsubscribe for p_s.

4. B_o unsubscribes for p_s after receiving MU_s

The idea behind this process is, after connecting with a new broker, the subscriber will tell the broker his ID (which is the same as the predicate ID right now) and the ID of the

old broker it was connected to. Since a broker node in CBCBSim knows only about itself and its neighbors, the new broker will issue an MI packet searching for the old broker and obtain the predicates subscribed by s. This way, the subscriber, which has limited resources, does not have to save its predicates locally. Now, there are two approaches to instruct the old broker to unsubscribe for the predicates being moved.

1. Unsubscribe the predicate as soon as the MI is received because it is confirmed that the subscriber has moved to the new location and there is no need to receive notifications related those predicates unless either the new broker is a descendant of the old broker for the broadcast tree rooted at the node issuing ME, or the message contents are satisfying subscriptions by other subscribers at the old broker itself/descendants of the old broker. This approach will suffer the problem of losing some messages.

2. The old broker does not unsubscribe for predicate(s) being moved immediately after receiving the MI to avoid loss of messages. Instead, the moment it receives the MI, it starts buffering the incoming message packets (ME) because the subscriber has moved to a new location. The ME(s) are buffered until the old broker receives an MU packet indicating that the new broker has subscribed for moving the predicate(s) for moving the subscriber, and from now onwards it will receive the corresponding messages. The buffered packets are then issued to the network. This approach can suffer from the problem of duplicates but will guarantee the delivery of the messages.

In current implementation, we are following the second approach to some extent. Currently we are not supporting the buffering mechanism in CBCBSim and are allowing loss of some messages. The reason behind this is that the original implementation is suffering from a race condition, i.e. messages issued while propagating the new subscription can get lost. This is caused by network latency. For example, consider a subscription $x > 100$ is issued at time t units. Assuming latency per link as 1 time unit, neighbors of the current node n_i will receive this information at t +1. Their neighbors will receive it at $t + 2$ and so on until it reaches

nodes which are either have the subscriptions at those node(s) covering this subscription, or are leaves of the broadcast tree rooted at n_i. If a message satisfying the condition $x > 100$ such as $x = 200$ is issued at some node n_j, between time t to $t + k$, where node n_j is k hops away from n_i, and $k > 0$, it will not be delivered to node n_i. Due to this race condition, even if we buffer the ME(s) till MU is received at old broker, we can not avoid loss of messages. A possible solution to avoid the race condition is buffering ME(s) till time T where $T =$ the amount of time required to propagate a packet from one end vertex of the graph diameter[1] to another and then propagate those packets. This can obviously introduce a delay in providing event information which can be harmful for time sensitive event notifications (For example, a medical emergency). Also, we can propagate twice, i.e. propagate immediately as well as buffer and then re-propagate after time T. This solution will introduce duplicates and can have other side effects such as congestion, increase in communication cost, etc.

We have discussed the CBCB Routing scheme and implementation of mobility in CBCB networking.In the next chapter, we will discuss the advanced semantics in event-based systems and the histogram implementation to support them.

[1] *Graph diameter* is also known as the "*longest shortest path*" in the graph. In the case of an unweighted graph, we can define it as the largest number of vertices which must be traversed in order to travel from one vertex to another in the graph without backtracking or looping.

Chapter 5

ADVANCED SEMANTICS USING HISTOGRAMS

Remember the example of medical services during a disaster given in section 1.1.2 where one of the needs for flexible event notification systems is highlighted. This chapter, in particular, will explain need for these advanced semantics, related work, and implementation of histograms as well as provide some insight into utilizing these histograms for flexible notifications.

5.1 Flexible Event Notifications

Consider a thousand people in Atlanta or nearby cities subscribing for the traffic event "Crash" within an approximately 20-mile radius around their current position (refer to the example given in section 4.2). Now if some crash event has occurred near downtown Atlanta then "*all*" the subscribers within 20 miles distance from that position should receive the notification. Now consider the military medical service example where all the doctors advertise their medical support by subscribing for making their medical expertise available. For example, a subscription can be:

N_i = [id=* $\wedge$ soldier="Yes" $\wedge$ injury_level>8 $\wedge$ body_parts=* $\wedge$ location="Bagdad"]

Here if any soldier in Bagdad having a severe injury (on the scale 1 to 10 with 10 being most severe) to one or more body_parts issues a notification, it should reach to the doctor subscribing using this predicate. During war, all an injured soldier has to do is to issue a notification indicating he needs medical attention. Now, as we have explained in the example, only 3 out of 50 should get the notification to avoid the chaos. In other words, The event brokers need to propagate this notification such that exactly 3 out of 50 doctors who

are available will get this notification.

When we analyze both the scenarios in content-based publish-subscribe systems, we have some subscribers and some notifications describing the occurrence of the related event(s), but the way the notifications should be propagated is different. In the first scenario, we need to deliver notification to all the subscribers for whom message contents are satisfying the conditions. In second scenario, we need to deliver notification to some of the subscribers out of all the subscribers for whom conditions are being satisfied. Another challenge in this case will be picking out the 3 doctors who will receive these notifications. A bad system might deliver notifications to the same 3 doctors again and again while some doctors remain idle. To avoid this situation, the system should be fair. It is safe to say that the second scenario requires the system to deliver notification to a group of subscribers (can be random or specific, depending upon the application requirements) out of all subscribers. The event-based system needs flexibility in delivering the notification. Also, a soldier may require assistance from the same doctor he has received treatment from previously since that doctor may have more detailed knowledge about the soldier than other doctors. So there is a need for specific doctor.

In such scenarios, "*manycasting*" seams to be the ideal approach, and we also have faircasting/bestcasting/anycasting techniques which can be useful in one way or other depending on scenario under consideration. The chances of a subscriber S_i of receiving some event notification may depend upon the other subscriber S_j receiving that event notification. Also, it is possible that the chances of S_i receiving an event notification regarding event E_x are higher than receiving E_y. This imposes some challenges. The first is how much knowledge of network, subscriptions each broker has to acquire in order to support the above semantics. This can vary depending on many factors like the number of subscribers in the area, subscription patterns, usage, etc. The second challenge will be to design efficient protocols that

will let a broker acquire the required knowledge. Designing such kinds of algorithm is not a simple task. The next challenge will be designing an algorithm assisting each broker to make independent decisions based on the information acquired. This will be another difficult task because of the distributed nature of the pub-sub system. The fourth challenge is designing efficient data structures so that we can maintain and access necessary information about the subscriptions at each broker to be able to perform quick routing. Data access should be faster so that time-critical notifications (e.g. – stock quotes, medical attention, etc) can be processed effectively.

5.1.1 **Manycasting**

In simple networks, *manycast* is a group communication scheme allowing communication with an arbitrary number of group members. Anycast and multicast are both special cases of manycast where anycast means only one member of the group while multicast means all the members. [Carter et al., 2003] In the case of an event-based system, a group can be defined as a set of subscribers, for whom, the message contents are satisfying the subscriptions. For example, $x = 300$ is satisfying subscriptions such as (i) $x > 100$, $x > 200$, $x < 500$, etc. but not as (ii) $x > 1000$, $x < 50$, $x = 100$, etc. Thus, the current manycast group will be subscribers from (i).

Formally, we can define manycast in event-based systems as delivering notification to the k subscribers where m subscribers are eligible to receive notification from a network of n subscribers, where $k \leq m \leq n$. When $k = 1$, and $m \leq n$, we are looking at *anycasting*. When $k = m \leq n$, we are looking at *multicasting*. And $k < m \leq n$ & $k > 1$, is a general case of manycasting.

Introducing manycast in pub/sub systems is not a simple task. It faces many challenges, especially when we allow mobility. Similar to mundane and fixed networking, temporary disconnection of one or more subscribers or the worst – event brokers, can introduce errors

in delivering notification to exactly k subscribers. Thus, when a notification is supposed to be delivered to k subscribers, in reality, it should actually be delivered to $k+T_h$ subscribers where T_h is some threshold value for allowed error margin. Since there will always be some amount of disconnectedness/duplicates in mobility enabled event-based systems, we need to have this threshold.

There is not much work to support manycasting in event-based systems but there is some research in ad-hoc networking. [Carter et al., 2003] explains some of the approaches possible to support manycasting in such networks. Since our system is content-based publish/subscribe system, none of these approaches are useful. One should not get confused between manycasting or multicasting in event-based systems with the protocol given in [Banavar et al., 1999] which is a multicast protocol for content-based routing.

5.2 Histograms

A basic requirement for supporting these advanced semantics is storing, maintaining and accessing statistics about the subscriptions at each broker using efficient data structure. During this study, we implemented a structure called a *histogram* which will be explained in this section.

5.2.1 Histogram Construction and Management

Construction of a histogram is based on the routing scheme used. In general, we need to maintain statistics for particular ranges of values. Thus each entry in the histogram will have [*Lower_bound*, *Upper_bound*) = *Count* format for each attribute. Since different interfaces may have different predicate sets,

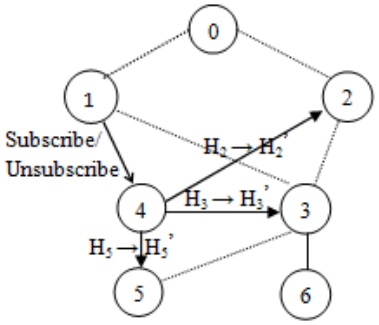


Figure 5.1: Updating Histograms

attribute-range-statistics will be different along each outgoing branch from the current event broker. Whenever a subscription arrives at broker, it updates its histogram which involves creating new attribute entries, creating new ranges, splitting ranges or just updating existing range statistics. In figure 5.1, broker 1 propagates either a subscription or unsubscription. Broker 4 updates its histograms along outgoing links to brokers 2, 3 and 5 respectively based on the predicate and type of operation, i.e. subscription or unsubscription.

5.2.2 Structure of Histograms in CBCB Routing scheme

The structure of histograms is designed according to the structure of predicates in the CBCB routing scheme. A predicate consists of a name/value pair with an operator. It can be depicted as [*variable_name operator value*]. Different predicates will have different values and different operators for a variable. Thus, there will be several ranges of different values for a variable. A predicate can have more than one variable, and/or different predicates may contain different variables. Also, in the routing table, there will be different interfaces for each neighbor. To propagate notification along the link to the neighbor that has a subscription satisfied by the contents of the message in this notification, we have to maintain statistics for each interface.

We design the histograms considering all these situations as follows. Current implementation only considers range-predicates with $<$, $>$ and $=$ operators.

1. For each variable there will be a few ranges of values. A predicate will fall into one of these ranges. We keep information about how many predicates fall into one range $\Rightarrow$ a *map* to keep statistics for each range and another *map* to store and access different variables for an interface.

2. For each interface, there might be more than one variable, each with its own ranges $\Rightarrow$ a *map* to store and access variables belongs to each interface.

3. Special attention is paid to the *equal* predicates since a range-predicate may belong to more than one range but equal predicates fall into exactly one range ⇒ This is kept in similar fashion where we have a map to keep track of different interfaces, a map for different variables for each predicate, and a map for keeping count for different values.

In the end we have a structure of the histogram as shown in figure 5.2. It represents a

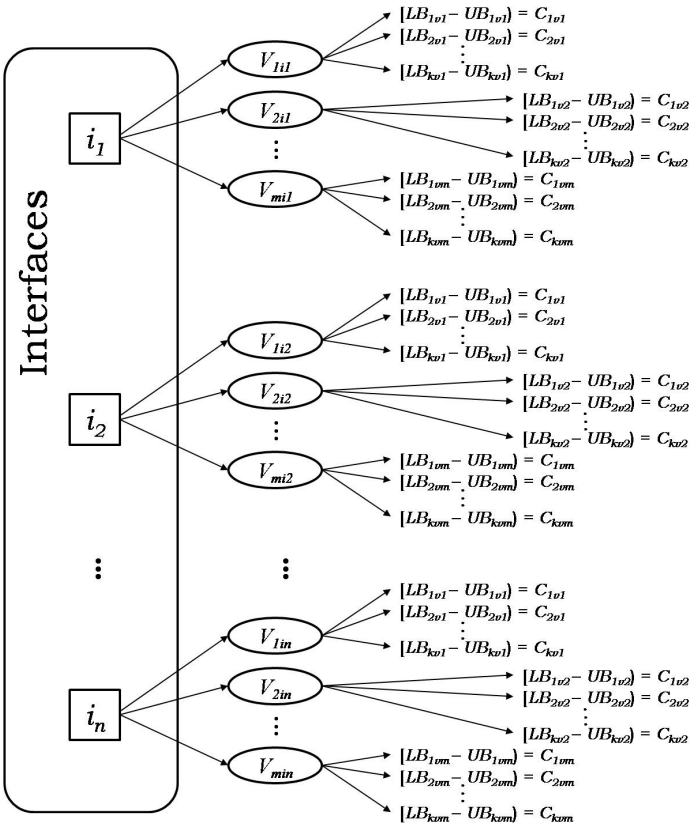


Figure 5.2: Structure of Histogram in CBCB Routing scheme

histogram for an event broker for n outgoing links to n different interfaces. Each interface has many attributes/variables with each having different ranges of values. Ranges are represented by (lb_i - ub_i) and 'c_i' represents the count of predicates falling into this range.

5.2.3 **Example**

Figure 5.3 demonstrates an example for the CBCB routing scheme which has an implementation of histograms. This figure helps to understand the notion of histograms diagram-

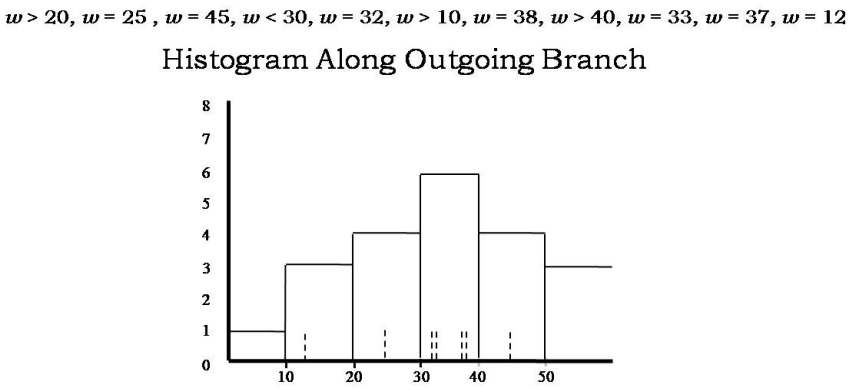


Figure 5.3: Histogram Example

matically. It is just a conceptual diagram representing a histogram along some outgoing link from an event broker after receiving a set of subscriptions. We have a set of incoming subscriptions for attribute 'w' as shown at the top of figure. The black lines represents spikes in the chart due to "equal to" predicates.

5.3 Approach for Advanced Semantics

Providing advanced semantics in a precise manner will be very expensive and involve a very high communication cost and a significant amount of local resources to store and process related data. The proposed approach here utilizing multidimensional range histograms will provide an *approximate semantic guarantee* which can help realize these semantics in an efficient as well as scalable fashion. At the same time, guarantees can be relaxed in a controlled and flexible manner such that the application will have a choice between better precision and higher communication cost. For instance, a subscriber in approximate-faircasting receives a matching event with probability $\frac{k}{m} \pm \epsilon$ where k indicates the number of desired subscribers, m indicates the total number of subscribers matching the event and ϵ is the approximation factor. The smaller the ϵ, the higher the communication cost of the application.

The faircasting, multicasting and the bestcasting discussed earlier will be obtained by maintaining subscription statistics at each broker using a set of approximate multidimensional histograms. Currently, the focus will be on faircasting, but same strategies can be applied to remaining techniques. As per discussion of the structure of a histogram in the previous section, the histogram will indicate an approximate number of subscriptions which match with various combinations of attribute ranges pertaining to that topic. Each broker will maintain one histogram along with each outgoing path which will provide subscription statistics needed along that path. In the case of faircasting, a broker, after receiving an event during routing, will use this set of histograms and find a number of matching subscriptions along with each outgoing path. Based on this information, the broker can determine a set of neighbors to which it should forward the event message and forwards notifications accordingly. The k is decreased w.r.t number of notifications forwarded along each path. Every broker along the path executes this algorithm and once k reaches zero, no further notifications are required and the message is simply dropped. An example of execution is shown in following figure 5.4

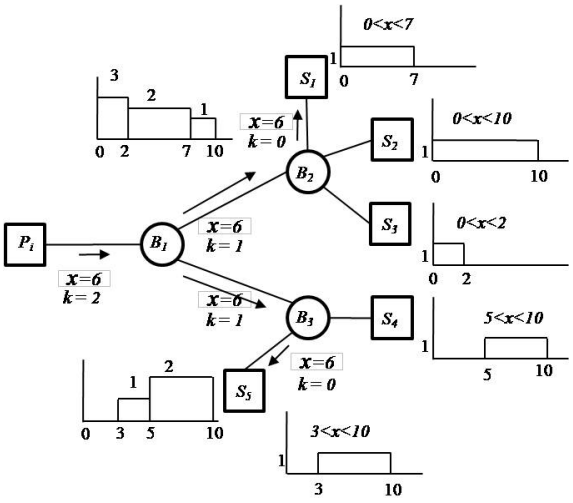


Figure 5.4: Histogram Based Faircasting

The figure illustrates faircasting with $k = 2$. which means that only 2 subscribers should be able to receive the notification $x = 6$ published by P. There are no subscribers at broker

B_1 which determines that path along B_2, which has 2 subscribers and the path along B_3, which also has 2 subscribers matching this event. B_1 forwards the event along both the paths after setting the value of $k = 1$ along both forwards. At both brokers, B_2 and B_3, two subscribers each are found subscribing for this event. In this case both of them forward notification to any of the subscribers which makes $k = 0$ and no further propagation is required. It is possible not to forward the event to any of these subscriber but along some other outgoing edges to maintain fairness.

Each broker builds histograms of various event topics reflecting subscriptions of its direct subscribers. Figure 5.4 illustrates histogram construction assuming only one event topic x. Lazy update propagation will be adopted to mitigate the high cost of communication in dynamic environments.

Chapter 6

EXPERIMENTAL RESULTS

This chapter discusses experimental setup for and results obtained from various experiments conducted. The tests compare throughput in terms of packets, bytes and time units against different number of nodes as well as optimized and unoptimized implementations.

6.1 Experimental Setup

A number of tests were performed based on the parameters given below:

1. Optimized and unoptimized implementations
2. Number of nodes
3. Number of moving predicates
4. Position of the node where predicate is being moved
5. Position of the node where predicate is being covered

The measure of the distance between two nodes is the number of hops, where 1 hop is covered by a packet in 1 time unit. In the real world, it depends on various factors such as medium, actual distance between nodes, bandwidth, network delay, etc. Accordingly, the performance of the system will be affected. For experimental purposes, we are only considering number of hops, simulated time and constant bandwidth and/or delay in the network.

Different tests are conducted by varying one or more of these parameters. First of all, the same sets of tests are conducted for both optimized and unoptimized implementations. One

set of tests contains the same number of nodes (i.e. 1000 nodes) but a varying number of moving predicates from 100 to 1000 irrespective of the node positions. All the other sets keep the number of moving predicates constant i.e. 1 and vary the number of nodes. Also, different cases were tested based on position of nodes under consideration, such as neighboring nodes, nodes at a distance equal to the diameter of the graph and nodes at average distance. The position of related nodes for an experiment provides different cases, for example, a predicate is covered by nodes at any of the positions mentioned above. Also, a predicate can be move from one node to a node at any of these positions. Based on this criteria, nine different cases are considered for some of the experiments. These cases are explained below with the help of an example.

Consider the following paths for example specifying edges forming it:

Path 1: <865,649><649,150><150,123><123,59><59,17><17,38><38,202><202,449>

Path 2: <865,419><419,351><351,338><338,31><31,19><19,39><39,307><307,362>

Both of these paths give the following nodes at different positions under consideration with node 865 as a source (node where the original predicate is subscribed).

	short	average	long
Path 1	649	59	449
Path 2	419	31	362

(i) Short_covered_short_move: A predicate at node 865 is covered by a neighboring node 649 and is being moved to either node 649 itself or node 419.

(ii) Short_covered_average_move: A predicate at node 865 is covered by a neighboring node 649 and is being moved to either node 59 or node 31.

(iii) Short_covered_long_move: A predicate at node 865 is covered by a neighboring node 649 and is being moved to either node 449 or node 362.

(iv) Average_covered_short_move: A predicate at node 865 is covered by node 59 (average distance) and is being moved to either node 649 or node 419.

(v) Average_covered_average_move: A predicate at node 865 is covered by node 59 and is

being moved to either node 59 itself or node 31.

(vi) Average_covered_long_move: A predicate at node 865 is covered by node 59 and is being moved to either node 449 or node 362.

(vii) Long_covered_short_move: A predicate at node 865 is covered by node 449 (graph diameter distance) and is being moved to either node 649 or node 419.

(viii) Long_covered_average_move: A predicate at node 865 is covered by node 449 and is being moved to either node 59 or node 31.

(ix) Long_covered_long_move: A predicate at node 865 is covered by node 449 and is being moved to either node 449 itself or node 362.

It should be kept in mind that these are the cases that are considered for experiments but we can have covering nodes or destinations at any distance from the shortest to the longest mentioned above. The longest distance equals the graph diameter.

It can be observed that this example explains two different cases within each case itself where the predicate is moved on the same path as that of the covering node or a different one. Furthermore, these cases explain covered predicate concepts but there can be covering predicates present at different nodes too. Those are not considered separately while conducting experiments because if it is not covered by some other predicate, then it is as good as moving a non-covered predicate. If it is covered by some other predicate on other node, then it is one of the cases mentioned above.

6.2 **Experimental Results**

This section will discuss The results obtained though the tests conducted in different cases by representing it with charts and/or tables.

6.2.1 **Constant Number of Nodes**

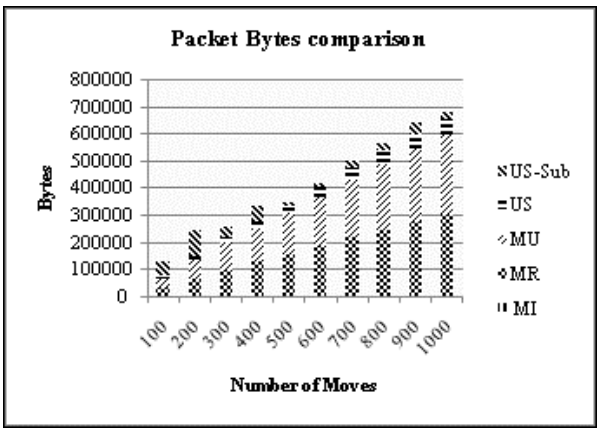
the number of nodes is kept constant i.e. 1000 and the number of moves is varied from 100 to 1000 by incrementing using steps of 100. Since this particular test aims at

obtaining the average throughput of the system, other variables are not necessarily kept constant. The experiment is conducted by generating a database of few thousands of moving predicates and then copying the required number of predicates into different test cases. All the logs obtained are parsed using a simple program written in Java to get results for further analysis. This particular database is generated using the following parameters:

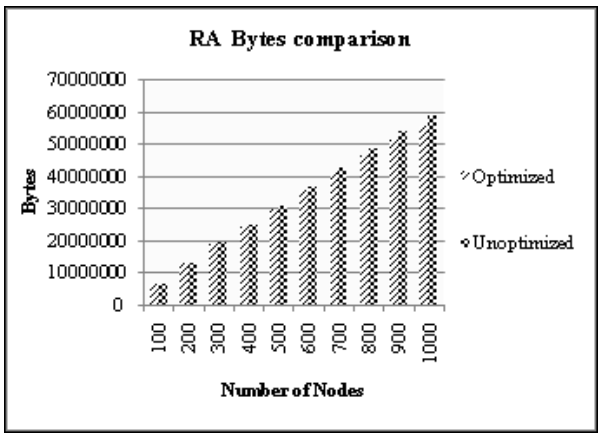
(i) Attributes: Minimum 1, Maximum 5
(ii) Constraints: Minimum 1, Maximum 5
(iii) Filters: Minimum 1, Maximum 5
(iv) Node count = 1000
(v) Publisher count = 500
(vi) Subscriber count = 1000
(vii) Simulation length = 10 minutes
(viii) Simulation time unit = 1 sec
(ix) set_predicate: Mean time to event = 2 (sec)
(x) Seed = 472810
(xi) move_predicate: Mean time to event = 10 (sec)

Comparing Different Packets w.r.t. Predicate Bytes

As expected, the number of predicate bytes required increases linearly along with the number of predicates being moved irrespective of the position of related nodes. Since the an MI packet contains only the predicate ID and not the actual predicate, no predicate bytes are present in Figure. It is propagated from destination to the source investigating the source. Using predicate ID, corresponding predicate, if present, is found at source, and MR packet issued toward the destination along the reverse path. These bytes are shown using checkers which show a linear increase with the number of moves. Once the predicate in question is being retrieved by the destination node, it issues RA packets starting the subscription process and the MU packet toward the source asking it to unsubscribe. These MU packets are always equal to MR packets since both travel along the same path in opposite directions and both have similar linear characteristics. Although the bytes required by US packets for both unsubscription and subscription predicates vary for different numbers of moves, these bytes have linear characteristics overall. These may not be sharp as in case of MR and MU, but they are linear. Figure 6.1a represents these cases whereas Figure 6.1b represents RA predicate bytes in both optimized and unoptimized cases (Note that they are linear and



(a) Without RA predicate bytes



(b) RA predicate bytes Comparison

Figure 6.1: Comparing Different Packets w.r.t. Predicate Bytes

almost same in both the cases). Since the number of bytes required by RA packets is much higher, these bytes are represented by a different chart.

Comparing Number of Packets Required

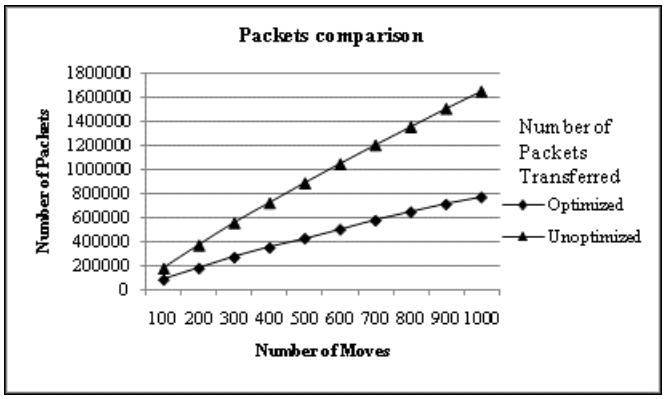
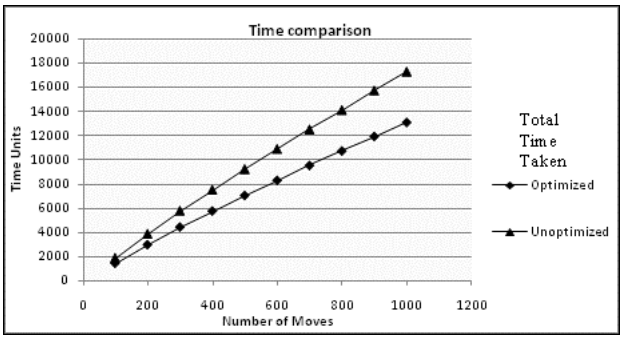


Figure 6.2: Comparing Number of Packets

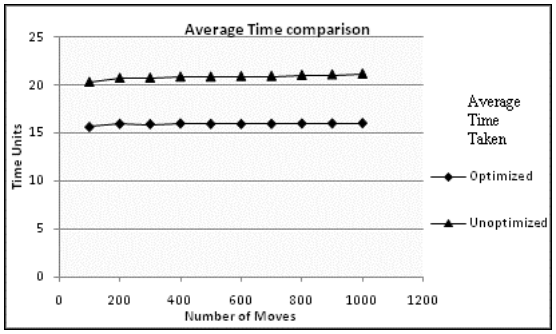
Figure 6.2 shows that number of packets required for the different number of moves increases linearly with an increase in the number of moves. Additionally, it can be seen that unoptimized implementation demands significantly more packets than optimized implementation. The line representing the unoptimized version has a much higher slope than that of

the optimized version, which leads to the conclusion that greater the number of moves, comparatively, the fewer number of packets that are required for propagating necessary changes.

Comparing Time Units Required



(a) Total Time for All Moves Comparison



(b) Average Time Per Move Comparison

Figure 6.3: Comparing Time requirements

The time required to propagate all the moves and necessary changes throughout the network is represented by the figures in Table 6.1. The total time required for both optimized and unoptimized versions (Figure 6.3a) increases linearly with the number of moves, and clearly, it can be seen that it is higher for the unoptimized version. If we consider average time required per move represented in Figure 6.3b, it is seen to be constant for both versions as desired.

Comparing Unsubscription Bytes

	Bytes	
Number of Moves	US-Optimized	US-Unoptimized
100	7501	6851000
200	14482	13286000
300	20176	19786000
400	25753	25246000
500	30914	30849000
600	37063	36933000
700	43329	43069000
800	49088	48776000
900	54795	54431000
1000	59553	59384000

Table 6.1: US Bytes for All Moves

Table 6.1 and Figure 6.4 both represent unsubscription bytes required for moving given number of predicates. The graph shows that number of bytes required for unsubscription packets by optimized implementation are significantly less than that of unoptimized version. Table is a reference given to corresponding readings since unsubscription bytes

from chart appears very close to zero in Figure. We can easily conclude that optimized version provides huge improvement over unoptimized in case of unsubscription of predicates.

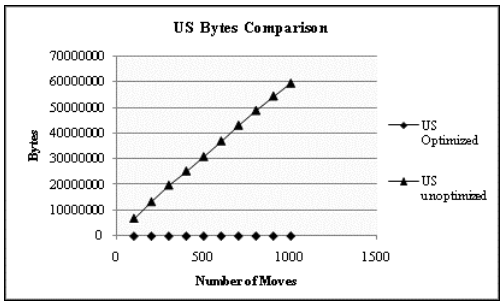


Figure 6.4: US Bytes for All Moves

Further, referring to Figures 6.1a, 6.1b and 6.4 we can infer that optimized implementation provides a huge improvement over the unoptimized when all the operations involved in moving of predicates are considered.

6.2.2 Constant Number of Nodes and Moves, Different Cases

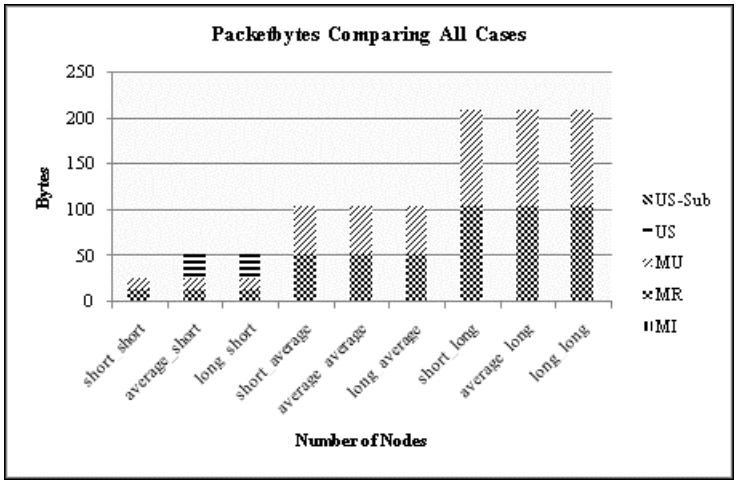


Figure 6.5: Comparing All Cases w.r.t. Different Packet Bytes

In this experiment, a topology of 1000 nodes is selected, and 1 predicate is moved in order to consider different cases. Figure 6.5 does not consider RA bytes since the number of RA predicate bytes is significantly higher than that of the others. In all the other packets, it is seen that the cases did not matter much since MI does not carry any predicate bytes;

it is zero. MR and MU packets carry equal number of predicate bytes since they contain the predicate being moved, the first one as a response to a mobile query and latter one for instructing unsubscription at the source node. The bytes needed increase linearly with the distance between source and destination nodes. Other charts show that changing the number of nodes does not affect these two types of packets and the number of bytes completely depends on the distance. The number of US bytes required are seen only in two cases where a predicate at a node covered by a node further than the distance between neighbors is moved to a neighbor node. In this case unsubscription packets are dispatched to the neighbors of the source node. Hence, the number of unsubscription bytes depends on the number of neighbors and inherently on predicate size (number of filters, constraints, attributes, types of attributes and values). These US packets are required to indicate that the subscription is moved from the source node, and predicates may or may not associate again with the same interface at the neighbors. All the neighbors then make changes if necessary and since the interface remains same for their neighbors, no further unsubscription bytes are required. In special cases where subscription bytes are present within US packets (due to covered-covering relationships), US packet may again be propagated through the network to make necessary changes in the issuer-predicate table and/or routing tables. These packets may not contain unsubscription bytes.

6.2.3 Constant Number of Moves (1)

The number of moving predicates is kept constant and the number of nodes are varied from 100 to 1000, incrementing in steps of 100. This test aims to observe the performance of the system affected by moving predicates in terms of number of packets required, predicate bytes required, as well as time improvement w.r.t. change in network size. This experiment is conducted by finding out various nodes at distance equal to shortest, graph diameter and average distance, and placing various predicates at these nodes considering the predicate being moved and covering predicates. Simple predicates are used where number of attributes,

attribute lengths, constraints and filters are 1. Values are all integers which are 4 bytes and only comparison operations such as $<$, $>$, and $=$ are considered. Thus, a predicate usually consists of 13 bytes in these tests.

During these experiments, we have graph diameters of length 7 hops for nodes 100, 200 and 300, and 8 hops for remaining nodes until 1000. This has to be done because of the topologies generated using BRITE [Medina et al., 2001] and the desired nodes selection for observing results.

Boston university Representative Internet Topology gEnerato (BRITE)

BRITE is a network topology generator. BRITE is described by the authors as a universal topology generator having representativeness (synthetic topologies reflecting actual Internet topologies w.r.t. hierarchical structures, nodes degree distribution etc.), inclusiveness (many generation models such as flat AS, flat Router and hierarchical topologies) and interoperability (allows importing topologies from other models), flexibility (wide range of different size networks), extensibility (user can easily extend capabilities), and few more features. Hence, this topology generator is chosen for experiments. Also, the workload generator, provided by CBCBSim, works with topologies generated by BRITE.

Working of Brite: Broadly it can be stated as four step process:

1. Place the nodes in the plane
2. Interconnect those nodes
3. Assign attributes to topological components – links: delay and bandwidth; router nodes: AS Ids, etc
4. Output topology to a required format

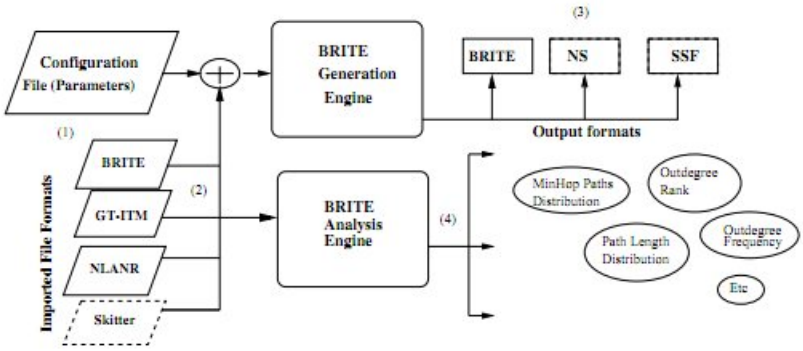
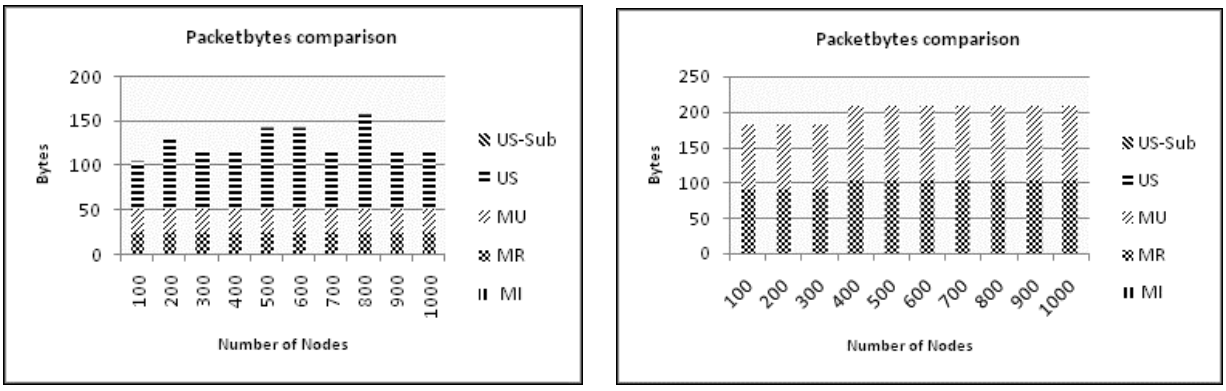


Figure 6.6: Schematic Structure of BRITE

This process does not explain generation of all the models but conceptually it is reflecting what is happening. Given the necessary inputs (files/parameters), BRITE generates different outputs as per requirements. Figure 6.6 shows schematic representation of this process.

Comparing Different Packets w.r.t. Predicate Bytes



(a) Short Covered Short Moved

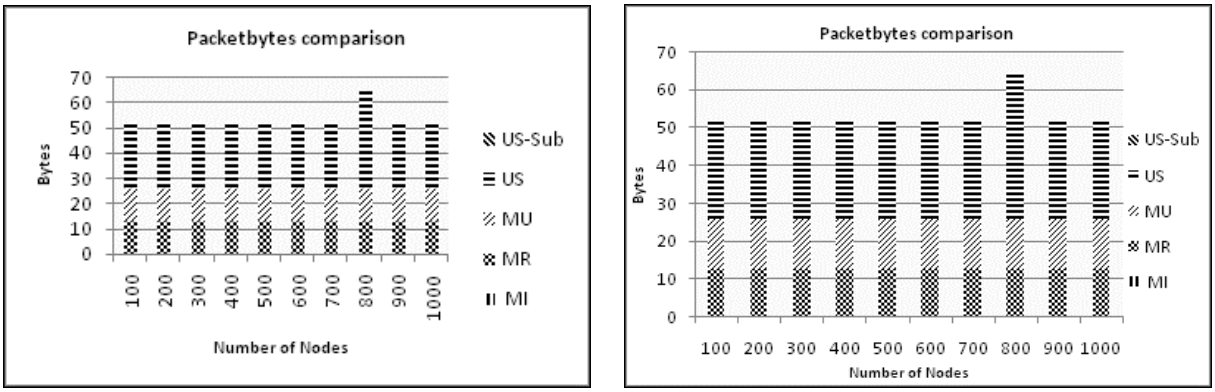
(b) Long Covered Long Moved

Figure 6.7: Comparing Different Packets w.r.t. Predicate Bytes - Simple Cases (Without RA predicate bytes)

We have already seen in the previous section, that *the number of predicate bytes required increases linearly along with number of predicates being moved* irrespective of the position of related nodes. MI packets have no predicate bytes due to absence of predicates in it. From Figures 6.7a and 6.7b it is observed that only MR and MU packets contain predicate

bytes, and they increase linearly with an increase in the distance between two nodes. This conclusion is solidified by observing charts for moving predicates to average distance. It is quite obvious since the algorithm used for finding source of predicate being moved propagate packets recursively from source to destination using parent-children relationship in the case of MR packets and vice versa in the case of MI and MU packets. The packets cannot be directly delivered between two nodes unless they are neighbors since the system aims at keeping only local information at any node and no global information.

In most of the cases, it is observed that no unsubscription bytes are required to be propagated except when a predicate at a node is moved to its neighbor, as seen in Figures 6.8b, 6.8a and 6.8b. The column of 800 nodes is little higher than others due to the fact that the source has 3 neighbors rather than 2 like all other cases. We have already seen that the number of unsubscription bytes is directly proportional to number of neighbors.



(a) Long Covered Short Moved (b) Average Covered Short Moved

Figure 6.8: Comparing Different Packets w.r.t. Predicate Bytes - Mixed Cases (Without RA predicate bytes)

Short_Covered_Short_Moved is a special case, in which, a sudden increase and decrease in the unsubscription bytes is seen. This can be explained with the help of Figure 6.9.

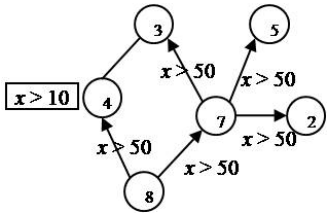


Figure 6.9: Short Covered Short Move - Special Case

Consider an example of a network where a predicate p1: $x > 50$ is subscribed at node 8 and another predicate

p2: $x > 10$ is subscribed at node 4 and p1 is being moved from node 8 to node 4. Now, node 4 is connected to node 3, which also connected to node 7.

When the predicate is moved from node 8 to node 4, it is unsubscribed at node 8, which asks its neighbors to adjust their routing tables. In an ideal situation, node 4 and 7 should simply drop the US packet since p2 is covering p1 at 4, and node 7 should have received p2 from node 4 through node 8. However, it is possible that node 7 received p2 from node 3 instead of node 7, resulting in two separate entries for p1 and p2 against two different interfaces.

Node 8: $x > 50$

Node 3: $x > 10$

Node 2: -

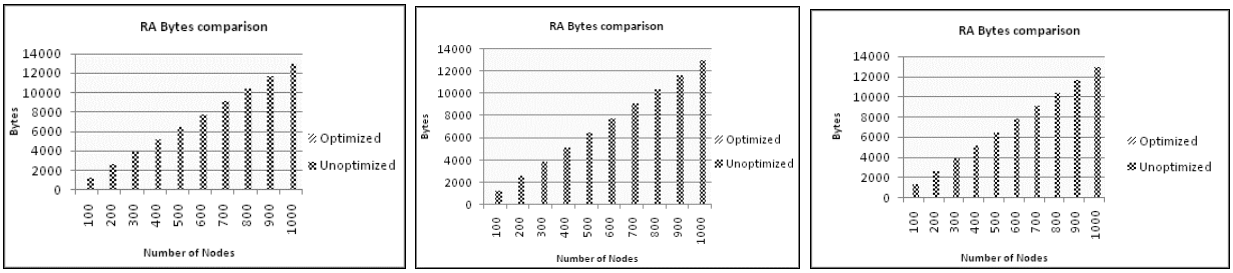
Node 5: -

This results in further propagation of US packets to node 7's neighbors.

In real world situations, these kinds of anomalies are quite common but can be controlled as per convenience, and system performance may be affected accordingly.

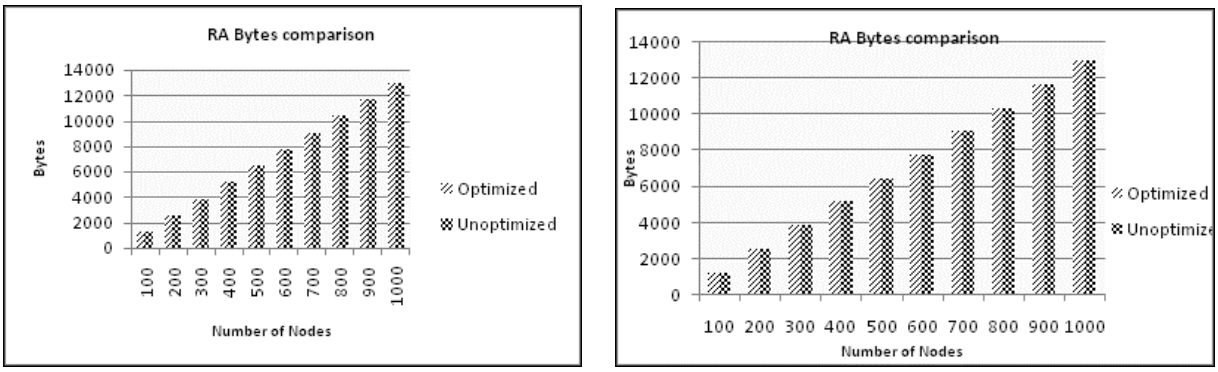
The following charts in this remaining subsection represent RA predicate bytes required for a different number of nodes. Evidently, it can be said that the number of RA packet bytes are directly proportional to the number of nodes in the system. If only subscriptions are allowed, then its possible to reduce these numbers but since the system now supports unsubscription and mobility correctly, it is required that RA packets to be propagated throughout the network most of the time.

Figures 6.10a, 6.10b and 6.10a represent simple cases where a predicate is moved to the node covering that predicate. In optimized implementation, there is no need to propagate RA packets through the network since they are already covered. An unoptimized version, on other hand, requires propagation to maintain correctness of unsubscription and mobility.



(a) Short Covered Short Moved (b) Average Covered Average Moved (c) Long Covered Long Moved

Figure 6.10: Comparing RA Packets w.r.t. Predicate Bytes - Simple Cases



(a) Long Covered Short Moved (b) Short Covered Average Moved

Figure 6.11: Comparing RA Packets w.r.t. Predicate Bytes - Mixed Cases

In other cases, irrespective of implementation RA packets are propagated throughout the network for correctness, as shown in by Figures 6.11a and 6.11b.

Comparing Number of Packets Required

Figure 6.12 shows that the number of packets required for moving a single predicate increases linearly with the number of nodes. Thus, we can conclude that the number of packets required to move a predicate through the network is directly proportional to the number of nodes in the system. Although, it is not quite clear from the graph, optimized implementation require a much small number of packets as compared to unoptimized

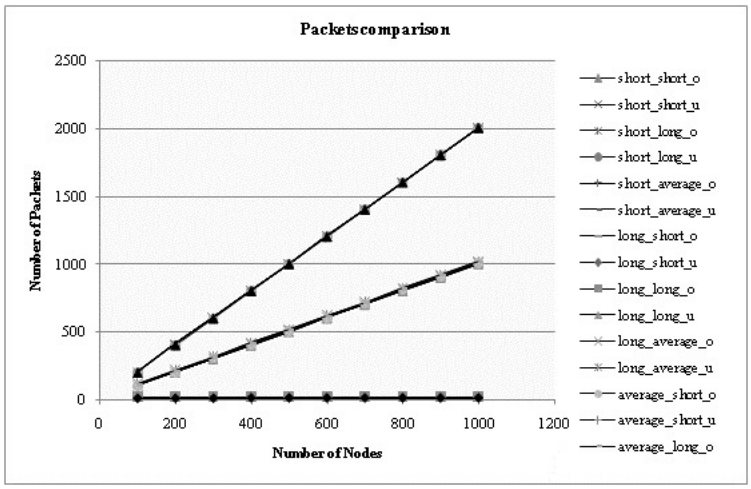


Figure 6.12: Comparing Number of Packets Required for Single Move

implementation. The lines with the highest slope represent packets required for moving a predicate within given a network using an unoptimized version, whereas lines with the smallest slope (almost horizontal) represent two cases of the optimized version, namely average_covered_average_move and long_covered_long_move. All the lines between these two represent the remaining cases of the optimized program.

Comparing Time Units Required

Figure 6.13 shows time requirements in one of the cases (long_covered_long_move) for both version of implementation. It can be observed that time requirement for nodes 100, 200 and 300 are constant, and it increases by one step and is constant for further nodes (400-1000). The reason for this nature of the graph is networks with nodes 100-300 having a graph diameter of 7 and network with nodes 400-1000 having a diameter of 8. Clearly, it shows that the time required to move a predicate is directly proportional to the diameter of the graph and not number of nodes in the network.

The time improvement obtained using optimized implementation for propagating one moving predicate and related changes throughout the network is represented by Table 6.2.

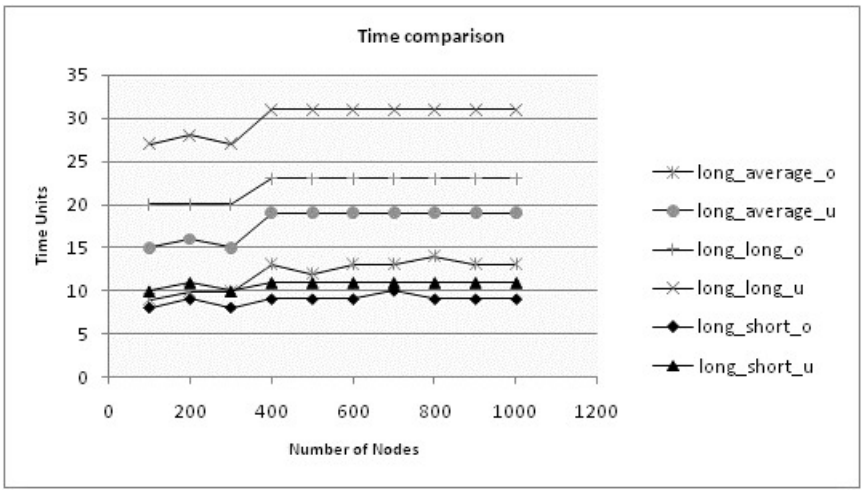


Figure 6.13: Time Comparison (Long Covered Cases)

Following formula is used to calculate time improvement:

$$\frac{(T_u - T_o)}{T_u} * 100$$

Where T_u = Time required to propagate changes in the unoptimized version

T_o = Time required to propagate changes in the optimized version

It is obvious that the optimized version has improved the required time to reflect propagation of a predicate being moved. In some cases (e.g. - average/long_covered_short_move) it may not be significant (9%-20%), but it is huge (30%-50%) in a few cases(such as short/long/average_covered_average_move). The improvement is moderate, i.e. approximately 25%, in the case of moving predicates through higher distances. Another important thing to notice is number of nodes does not matter for time improvement since it is almost constant for all of them when diameters of the graphs are almost equal (7 hops for 100-300 nodes and 8 for 400-1000 nodes in experiments). In some cases, we can observe the abrupt drop in figures, which is the result of different topologies of the network.

Nodes	Case					
	short_short	avg_short	long_short	short_avg	avg_avg	long_avg
100	50	20	20	40	46.67	40.00
200	54.55	18.18	18.18	37.50	50.00	37.50
300	50	20	20	33.33	46.67	33.33
400	54.55	18.18	18.18	31.58	42.11	31.58
500	54.55	18.18	18.18	31.58	42.11	36.84
600	54.55	18.18	18.18	31.58	42.11	31.58
700	54.55	9.09	9.09	31.58	42.11	31.58
800	54.55	18.18	18.18	26.32	42.11	26.32
900	54.55	9.09	18.18	26.32	42.11	31.58
1000	54.55	18.18	18.18	31.58	42.11	31.58

Nodes	Case		
	short_long	avg_long	long_long
100	25.93	25.93	25.93
200	28.57	28.57	28.57
300	25.93	25.93	25.93
400	25.81	25.81	25.81
500	25.81	25.81	25.81
600	25.81	25.81	25.81
700	25.81	25.81	25.81
800	25.81	25.81	25.81
900	25.81	25.81	25.81
1000	25.81	25.81	25.81

Table 6.2: Time Improvements

6.3 Conclusion

Many experiments and their results are discussed in this chapter. Beyond any doubt, optimized implementation provides a huge improvement over unoptimized version. In both the cases, the number of packets and predicate bytes used by MI, MR and MU packets used to introduce mobility are directly proportional to the distance between source and destination, and number of moves, and not on number of nodes in the network. Number of RA packets and RA predicate bytes, on other hand, depend on number of nodes as well as on the number of moves but not on the distance. Total number of packets required to propagate all the moves in a network are directly proportional to the number of operations, and total number of packets required to move one predicate in a different network is directly proportional to the number of nodes. Average time taken to propagate all the moves in a given network is constant, and the total time increases linearly with number of moves. Also, time required to move one predicate does not depend on the number of nodes in the network but on the diameter. The unsubscription bytes or the US packets required for necessary changes depend on the situation such as predicate at a node is covered by another predicate at a distance higher than its neighbor and it is moved to the neighbor. Thus, the number of US packets/unsubscription bytes is not constant w.r.t. the number of moves or number of nodes or diameter. Hence, the network graph diameter affects the number of packets, predicate bytes and time required to propagate changes. The number of nodes affects mainly RA packets and predicate byes used by them. Distance between source and destination affects MI, MR, MU and US packets. US packets also depend on the situation with covered and/or covering relationships. Factors such as attributes, filters, constraints, and values directly affect the size of the predicate in terms of bytes and thus affect the total number of bytes, which is reflected in total communication cost in terms of bytes along with other factors discussed above.

Chapter 7

CONCLUSION AND FUTURE WORK

This chapter will sum up the study and explains a few more ways to extend it further.

7.1 Conclusion

The purpose of this work was to study mobility in event-based systems, specifically, content-based systems, which are very flexible and powerful in terms of expressiveness, scalability, and performance. The importance of and the related issues in introducing mobility in event-based systems is discussed in the Chapters 1 and 3. Chapter 2 discussed event-based systems, associated challenges, and related work in the field of a few content-based systems, for example, Gryphon, Bayeux, Siena, NaradaBrokering, XMessages, and JMS. We have then seen a few mobile networking issues, and how features of event-based systems complement them. Application scenario for traffic management is a good example for mobility in content-based systems. This mobility is introduced and tested during this study using package CBCBSIM based on Combined Broadcast Content-Based routing scheme. Chapter 4 has the discussion about implementation of these concepts. The CBCBSIM is then directed toward supporting advanced semantics. Histograms are constructed for this purpose. These histograms are currently handling all the subscriptions, unsubscriptions, and movements of the subscribers accurately. The previous chapter had discussion regarding some of the results obtained by conducting various experiments on optimized version of mobility and unoptimized version supporting Histograms. These tests help to understand the roles of various factors affecting the system performance.

7.2 Future Work

During this study, histograms are constructed for CBCBSIM package but not used while propagating actual notifications. The future work might want to implement advanced semantics to provide flexible event notification system. Advanced semantics such as *manycasting*, *faircasting* and *bestcasting* can be supported using the histograms. Also, we have seen a trade-off in terms of the performance while introducing histogram feature into the package. This increased the cost of communication during subscribing, unsubscribing, or mobilizing the subscribers. Increase in cost due to these operations might be inevitable but can be reduced in different ways. For example, instead of sending all the predicates to all the nodes, developer may choose to propagate the histograms. He might want to aggregate few changes and send them out once, instead of propagating changes as soon as they have occurred. Thus, trade-off between accuracy and communication cost can be handled in such a way that the inaccuracy will not hurt the overall performance of the system by large amount. There are a few race conditions observed for delivering notifications while moving the subscriber. If those race condition are handled properly, a better performance can be obtained.

Bibliography

[Anicic et al., 2009] Anicic, D., Fodor, P., Stuhmer, R., and Stojanovic, N. (2009). Event-driven approach for logic-based complex event processing. In *Proceedings of the 2009 International Conference on Computational Science and Engineering - Volume 01*, CSE '09, pages 56–63, Washington, DC, USA. IEEE Computer Society.

[Bacon, 1999] Bacon, J. (1999). Report on the eighth acm sigops european workshop. *SIGOPS Oper. Syst. Rev.*, 33:6–17.

[Banavar et al., 1999] Banavar, G., Ch, T., Mukherjee, B., Nagarajarao, J., Strom, R. E., and Sturman, D. C. (1999). An efficient multicast protocol for content-based publish-subscribe systems. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, pages 262–, Washington, DC, USA. IEEE Computer Society.

[Caporuscio et al., 2002] Caporuscio, M., Inverardi, P., and Pelliccione, P. (2002). Formal analysis of clients mobility in the siena publish/subscribe middleware. Technical report, Department of Computer Science, University of LAquila.

[Carter et al., 2003] Carter, C., Yi, S., Ratanchandani, P., and Kravets, R. (2003). Manycast: exploring the space between anycast and multicast in ad hoc networks. In *Proceedings of the 9th annual international conference on Mobile computing and networking*, MobiCom '03, pages 273–285, New York, NY, USA. ACM.

[Carzaniga et al., 1998] Carzaniga, A., Di Nitto, E., Rosenblum, D. S., and Wolf, A. L. (1998). Issues in supporting event-based architectural styles. In *Proceedings of the third international workshop on Software architecture*, ISAW '98, pages 17–20, New York, NY, USA. ACM.

[Carzaniga et al., 2004] Carzaniga, A., Rutherford, M. J., and Wolf, A. L. (2004). A routing scheme for content-based networking. In *Proceedings of IEEE INFOCOM 2004*, Hong Kong, China.

[Chen et al., 2009] Chen, J., Ramaswamy, L., and Lowenthal, D. (2009). Towards efficient event aggregation in a decentralized publish-subscribe system. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 18:1–18:11, New York, NY, USA. ACM.

[Cugola and Jacobsen, 2002] Cugola, G. and Jacobsen, H.-A. (2002). Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6:25–33.

[Drosou et al., 2009] Drosou, M., Stefanidis, K., and Pitoura, E. (2009). Preference-aware publish/subscribe delivery with diversity. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 6:1–6:12, New York, NY, USA. ACM.

[Eugster et al., 2001] Eugster, P. T., Guerraoui, R., and Damm, C. H. (2001). On objects and events. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 254–269, New York, NY, USA. ACM.

[Eugster et al., 2003] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131.

[Farroukh et al., 2009] Farroukh, A., Ferzli, E., Tajuddin, N., and Jacobsen, H.-A. (2009). Parallel event processing for content-based publish/subscribe systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 8:1–8:4, New York, NY, USA. ACM.

[Fiege et al., 2002] Fiege, L., Mü hl, G., and Gä rtner, F. C. (2002). Modular event-based systems. *Knowl. Eng. Rev.*, 17:359–388.

[Fiege et al., 2003] Fiege, L., Gä rtner, F. C., Kasten, O., and Zeidler, A. (2003). Supporting mobility in content-based publish/subscribe middleware. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 103–122, New York, NY, USA. Springer-Verlag New York, Inc.

[Gazzotti et al., 2003] Gazzotti, M., Mamei, M., and Zambonelli, F. (2003). A programmable event-based middleware for pervasive mobile agent organizations. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:517.

[Hu et al., 2008] Hu, S., Muthusamy, V., Li, G., and Jacobsen, H.-A. (2008). Distributed automatic service composition in large-scale systems. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 233–244, New York, NY, USA. ACM.

[Huang and Garcia-Molina, 2004] Huang, Y. and Garcia-Molina, H. (2004). Publish/subscribe in a mobile environment. *Wirel. Netw.*, 10:643–652.

[IBM Gryphon, 1997] Buttner, G. (2001). The gryphon project. `http://www.research.ibm.com/distributedmessaging/gryphon.html`

[Jafarpour et al., 2009] Jafarpour, H., Mehrotra, S., Venkatasubramanian, N., and Montanari, M. (2009). Mics: an efficient content space representation model for publish/subscribe systems. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 7:1–7:12, New York, NY, USA. ACM.

[Keeney et al., 2008] Keeney, J., Roblek, D., Jones, D., Lewis, D., and O'Sullivan, D. (2008). Extending siena to support more expressive and flexible subscriptions. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 35–46, New York, NY, USA. ACM.

[Li et al., 2007] Li, M., Yan, T., Ganesan, D., Lyons, E., Shenoy, P., Venkataramani, A., and Zink, M. (2007). Multi-user data sharing in radar sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 247–260, New York, NY, USA. ACM.

[Liu and Plale, 2003] Liu, Y. and Plale, B. (2003). Survey of publish subscribe event systems. Technical report, Indiana University.

[Mü et al., 2006] Mü hl, G., Fiege, L., and Pietzuch, P. (2006). *Distributed Event-Based Systems* . Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Mü hl, 2001] Mü hl, G. (2001). Generic constraints for content-based publish/subscribe. In *Proceedings of the 9th International Conference on Cooperative Information Systems*, CooplS '01, pages 211–225, London, UK. Springer-Verlag.

[Mahambre and Bellur, 2008] Mahambre, S. P. and Bellur, U. (2008). An adaptive approach for ensuring reliability in event based middleware. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 157–168, New York, NY, USA. ACM.

[Mao et al., 2008] Mao, J., Jannotti, J., Akdere, M., and Cetintemel, U. (2008). Event-based constraints for sensornet programming. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 103–113, New York, NY, USA. ACM.

[Mascolo et al., 2002a] Mascolo, C., Capra, L., and Emmerich, W. (2002a). Middleware for mobile computing (a survey). In Gregori, E., Anastasi, G., and Basagni, S., editors, *Neworking 2002 Tutorial Papers*, volume 2497 of *lncs*, pages 20–58. springer.

[Mascolo et al., 2002b] Mascolo, C., Capra, L., and Emmerich, W. (2002b). *Mobile computing middleware*, pages 20–58. Springer-Verlag New York, Inc., New York, NY, USA.

[Medina et al., 2001] Medina, A., Lakhina, A., Matta, I., and Byers, J. (2001). Brite: An approach to universal topology generation. In *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '01, pages 346–, Washington, DC, USA. IEEE Computer Society.

[Meier and Cahill, 2002] Meier, R. and Cahill, V. (2002). Steam: Event-based middleware for wireless ad hoc network. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCSW '02, pages 639–644, Washington, DC, USA. IEEE Computer Society.

[Michlmayr et al., 2008] Michlmayr, A., Rosenberg, F., Leitner, P., and Dustdar, S. (2008). Advanced event processing and notifications in service runtime environments. In *Proceedings of the second international conference on Distributed event-based systems*, DEBS '08, pages 115–125, New York, NY, USA. ACM.

[Muthusamy et al., 2010] Muthusamy, V., Liu, H., and Jacobsen, H.-A. (2010). Predictive publish/subscribe matching. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 14–25, New York, NY, USA. ACM.

[O'Keeffe and Bacon, 2010] O'Keeffe, D. and Bacon, J. (2010). Reliable complex event detection for pervasive computing. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 73–84, New York, NY, USA. ACM.

[Oki et al., 1993] Oki, B., Pfluegl, M., Siegel, A., and Skeen, D. (1993). The information bus: an architecture for extensible distributed systems. *SIGOPS Oper. Syst. Rev.*, 27:58–68.

[Pietzuch and Bhola, 2003] Pietzuch, P. R. and Bhola, S. (2003). Congestion control in a reliable scalable message-oriented middleware. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 202–221, New York, NY, USA. Springer-Verlag New York, Inc.

[Podnar and Lovrek, 2004] Podnar, I. and Lovrek, I. (2004). Supporting mobility with persistent notifications in publish/subscribe systems. In *In Proc. of the 3rd Int. Workshop on Distributed EventBased Systems*, pages 80–85. ACM Press.

[Ramaswamy et al., 2009] Ramaswamy Lakshmish, Deepak P., Polavarapu Ramana, Gunasekera Kutila, Garg Dinesh,Visweswariah Karthik,Kalyanaraman Shivkumar (2009). CAESAR: A Context-Aware, Social Recommender System for Low-End Mobile Devices. In *Proceedings of Mobile Data Management*, pages 338–347

[Repantis et al., 2006] Repantis, T., Gu, X., and Kalogeraki, V. (2006). Synergy: Sharingaware component composition for distributed stream processing systems. In *In Middleware*, pages 322–341. Springer-Verlag.

[Russell et al., 2007] Russell, A., Wilkins, G., Davis, D., and Nesbitt, M. (2007). The bayeux specification. `http://svn.cometd.com/trunk/bayeux/bayeux.html` .

[Sutton et al., 2001] Sutton, P., Arkins, R., and Segall, B. (2001). Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, CCGRID '01, pages 277–, Washington, DC, USA. IEEE Computer Society.

[Wang et al., 2002] Wang, C., Carzaniga, A., Evans, D., and Wolf, A. (2002). Security issues and requirements for internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02) – Volume 9*, HICSS '02, pages 303–, Washington, DC, USA. IEEE Computer Society.

[Yoneki, 2003] Yoneki, E. (2003). Pronto: Mobile gateway with publish-subscribe paradigm over wireless network. *IEEE Distributed Systems Online*, 4:–.

[Yoon et al., 2006] Yoon, J., Noble, B. D., Liu, M., and Kim, M. (2006). Building realistic mobility models from coarse-grained traces. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, MobiSys '06, pages 177–190, New York, NY, USA. ACM.

[Zeidler and Fiege, 2003] Zeidler, A. and Fiege, L. (2003). Mobility support with rebeca. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, pages 354–, Washington, DC, USA. IEEE Computer Society.

[Zhong et al., 2008] Zhenyu Zhong, Lakshmish Ramaswamy, Kang Li (2008). ALPACAS: A Large-Scale Privacy-Aware Collaborative Anti-Spam System In *proceedings of INFOCOM*, pages 556–564.