A

Thesis

on

Event Notification Service System for Mobile Users

by

Chit Htay Lwin

Submitted in partial fulfillment of the requirements for the award of Degree of Doctor of Philosophy

**Department of Computer & Information Sciences
School of Mathematics & Computer/Information Sciences
University of Hyderabad
Hyderabad - 500 046
INDIA**

June, 2005

Department of Computer & Information Sciences
School of Mathematics & Computer/Information Sciences
University of Hyderabad, Hyderabad - 500 046
INDIA

CERTIFICATE

This is to certify that this thesis work entitled **"Event Notification Service System for Mobile Users"** being submitted by **Mr. Chit Htay Lwin** (Regd. No. **01MCPC08)** in partial fulfillment of the requirements for the award of the degree of **Doctor of Philosophy (Computer Science)** of the University of Hyderabad, Hyderabad, is bonafide record of the work carried out by him under my supervision.

The research work submitted in this thesis is not submitted for the award of any other degree/diploma of any other Institute/University.

29/6/05

Prof. Arun Agarwal
(HEAD)

29/6/'05

Prof. Hrushikesha Mohanty
(Supervisor)

29/6/05

Prof. A. K. Pujari
(DEAN)

DECLARATION

I, Chit Htay Lwin, hereby declare that the work presented in this thesis entitled **“Event Notification Service System for Mobile** Users", has been carried out by me under the supervision and guidance of **Prof. Hrushikesha Mohanty,** Department of Computer and Information Sciences, School of Mathematics & Computer/Information Sciences, University of Hyderabad, Hyderabad, India.

This research work is submitted as per the ordinances of the University, in partial fulfillment of the requirements for the award of the degree of **Doctor of Philosophy (Computer Science)** of the University of Hyderabad, Hyderabad.

I declare, to the best of my knowledge, that either full or part of this thesis has not been or is being concurrently submitted for the award of any other degree/diploma of any other Institute/University.

29/6/05

Chit Htay Lwin
Regd. No. 01MCPC08

To my parents

Abstract

Client/Server based computing with synchronous request/reply is one of the popular architecture of distributed systems. This architecture is not well suited to implement information-driven applications like news delivery, stock quoting, air traffic control, and dissemination of auction bids due to limitations of traditional request/reply. These limitations include tight coupling between components, a lack of information filtering capabilities, and support for one-to-one communication semantics only.

An event notification service system serves users (consumers) with information that occur at producers. Consumers and producers could be co-located at a system or at different systems, sometimes even lying at large distances. The service is carried out in accordance with the interests of users. Event notification service system is widely recognized as being well suited to interconnecting components of mobile applications since it naturally accommodates a dynamically changing population of components and the dynamic reconfiguration of the connections among them.

In this thesis we propose a conceptual architecture of event notification service system for mobile computing environment. Traditionally event notification service systems are used for realizing communications among software components that run on distributed systems. Later at the advent of Internet, it is realized that such systems built on Internet can cater to several applications. But the concepts of implementing event notification service systems cannot be directly applied for mobile computing environment. This environment is characteristically different than Internet environment due to mobility of users, location awareness computing and limited resource constraints at mobile devices and wireless links. These limitations project several challenges for investigation. We have identified three major research problems viz. ordering of events, just-in-time delivery of events and resilient dissemination of events that arc essential to be addressed. For each of these problems a solution is proposed and its performance is analyzed with simulation results. Some of the results due to this work are reported in [42, 43, 44, 45, 46].

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Prof. Hrushikesha Mohanty, Department of Computer/Information Seiences, School of MCIS, University of Hyderabad, for his constant guidance, encouragement and invaluable advices throughout my research work. Without his support, it would not have been possible for me to see the final form of my thesis. He also helped me to get exposure in other Institutes and Universities of India.

I am greatly indebted to Prof. R. K. Ghosh, Department of CSE, IIT Kanpur, India, for advising me during the early stage of my studies and for providing suggestions, support and encouragement. I express many thanks to Prof. Goutam Chakraborty, Department of SIS, Iwate Perfectural University, Japan, for showing interest in my research work and for helping this research.

I express my sincere thanks to Dean, School of Mathematics and Computer/Information Sciences (MCIS), and Head, Department of Computer/Information Sciences (CIS), for providing me the necessary infrastructure to carry out this work. I am grateful to other faculty members of the Department of Computer/Information Sciences for their encouragement and help during the period of my research work in India. I also thank all supporting staff of Department of Computer/Information Sciences at University of Hyderabad. I also thank Ms. Wilaiporn Sriphaisal for proof reading of thesis draft.

My special thanks go to all my family members who have been waiting for the day 1 finish this degree for their support and encouragement throughout the years, and also to all my friends both in Myanmar and India.

I also would like to thank ICCR (Indian Council for Cultural Relations), Government of India, for financial support for this research work. Finally, I would like to thank my country, Myanmar, for being my pride throughout my work and for granting permission to pursue Ph. D degree in India.

Chit Htay Lwin

Table of Contents

for Mobile Users

List of Tables

List of Figures

Chapter 1

Introduction

Event Notification Service (ENS) system is a data dissemination technology which asynchronously notifies *consumers* whose interests match with the events published by *producers*. An event notification service system provides *time decoupling* (i.e., the interacting parties do not need to be with system at the same time) and *space decoupling* (i.e., the interacting parties do not need to know locations of each other) [22]. The loose coupling of producers and consumers is the primary advantage of ENS systems. Event notification service systems can be used in many applications such as air traffic control [41], dissemination of auction bids [7], process control systems [37], stock trading, weather/traffic reports, travel updates, alerting services, emergency notifications, banking, auctions, etc.

With the advent of the Internet, it became possible to build large-scale distributed applications because of the existence of a global, packet-based communication infrastructure. The architecture of current large-scale and networked computer-systems is dominated by synchronous client/server platforms (e.g., the World Wide Web, Corba [54], J2EE [78]). In client/server systems, a client requests information or functionality from a server and then waits until the server has responded with a reply as shown in Figure 1.1. A client is blocked after it has issued a request, until the corresponding reply arrives. Moreover, the client object must know the identity of the server and can only interact with a single server at a time, which limits scalability. Remote Procedure Call (RPC) is a mechanism that facilitates a request/reply interaction between two distributed processes [73]. This is similar to the traditional mechanism of procedure calls [33] found in high-level programming languages. The fundamental difference is that the calling procedure

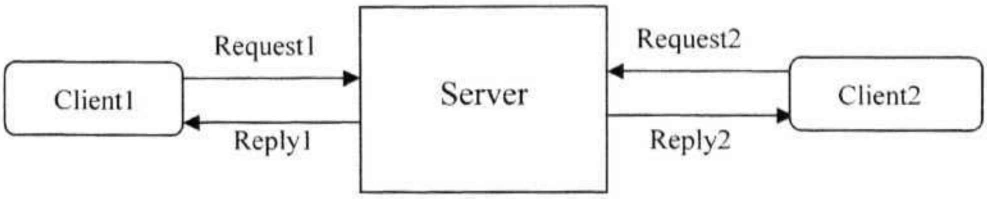


Figure 1.1: Client/server system

executes in one computing machine, and the called procedure executes in another, and data is exchanged between the two communicating parties.

For example, in a stock trading program, stock quote updates should reach to destinations with minimum latency without losses of quote updates. These requirements can only be met using a high polling frequency. Hence, data freshness is inconsistent with a low frequency on data fetching. Moreover, synchronous polling blocks the client until the reply arrives. This means that multiple polling requests (e.g., for different stocks) either have to be executed sequentially or error-prone multi-threaded polling must be used. The same would be true if multiple data sources (e.g., different stock exchanges) were involved. Polling leads to resource waste because it unnecessarily saturates the servers, the network, and the clients [29]. Polling is also inappropriate for applications that run on mobile devices (e.g., PDAs). These devices are especially susceptible to resource waste because of their limited processing power and network bandwidth. Event notification service system is a more scalable paradigm that addresses the shortcomings of request/reply communication by supporting many-to-many interaction among entities.

1.1 Event Notification Service System and Mobile Environment

A general architecture of event notification service system is provided in Figure 1.2. The figure describes basic functionalities of the system. A distributed event notification service system is composed of interconnected event servers connected to consumers and producers over a communication network. The event servers are connected with hierarchical, acyclic peer-to-peer and general peer-to-peer architecture. Consumers *subscribe* their interests to

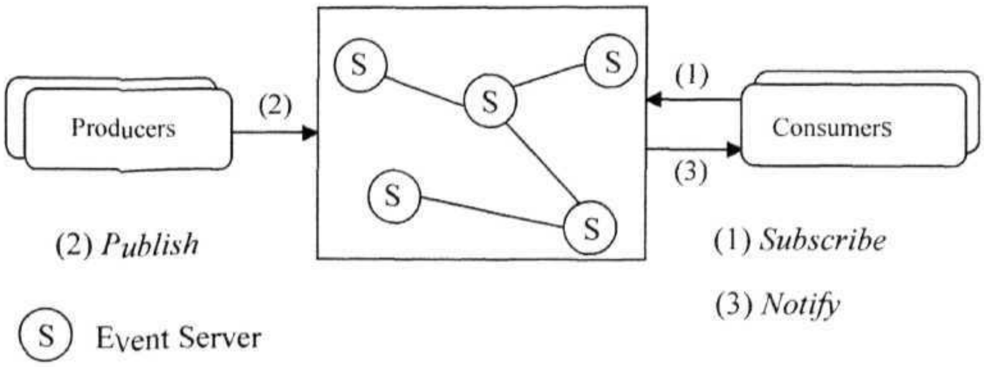


Figure 1.2: Distributed event notification service system

receive relevant events (notifications) and producers *publish* events. A consumer may *unsubscribe* to cancel its requested interest. *Subscriptions* can be considered as filters that help event notification service system to notify an event to consumers only if the event matches with consumers' subscriptions. Event notification service systems should be scalable distributed systems to disseminate event information like sensor readings, traffic status, new product information, etc. The consumers receive notifications, for example via hand-held devices, Internet pages, or email. The classification of several event notification service systems and their research issues are presented in our work [45, 46].

Advances in wireless network technology and the increasing number of handheld mobile devices make it possible to offer many services to mobile users. Emerging mobile and ubiquitous computing applications typically comprise a number of interconnected components distributed over large geographical areas. These components require to communicate and such communications are asynchronous in nature because of mobility of components. The impacts due to mobility of system components are contained at the algorithm and middleware levels. Algorithms from the area of traditional computing do not address the complexities of the distributed mobile environment. New algorithms, which adapt to the dynamic behavior of mobile computing environment, are to be developed. Several event notification service systems for the mobility of clients have been proposed in [10, 27. 79]. Generally, in these systems, during the time of disconnection notified events are stored at the event server at which the user registered last. On reconnection, the stored events are forwarded to the user's new location.

As an event notification service system follows multicast communication protocol, it allows a producer to send the same event to many interested consumers with only one publishing operation. It can cope with dynamically changing operational environment where producers and consumers frequently disconnect and reconnect. Similarly traditional mechanisms for passing messages in distributed systems needs to be adaptive for doing so in mobile computing environment. In this work we study the problem of sending events to a large number of mobile users. As senders and receivers are not required to be in synchrony, an event notification service system can be scalable and can adapt to changes in environment. The event notification service system can be used in real-life applications such as stock exchanges, real-time avionic mission computing systems, weather and traffic reports, travel updates, alerting services, emergency notifications, and trading services for financial market. As Event Notification Service system is extremely useful for communication service in many applications, it becomes necessary to extend this service system to a mobile environment.

1.2 Research Problems

This thesis first focuses in designing the architecture of event notification service system for wireless network with base station which is widely used in mobile environment and then investigates the following three challenges.

1.2.1 Ordering of Events

A distributed event notification service system is asynchronous system consisting of several subsystems namely, producer clients, consumer clients and event servers without having common memory and global clock. These subsystems communicate each other by sending messages with unpredictable transmission delays causing unordered propagation of messages. The ordering of events is more important for event notification service system in a mobile computing environment as frequent disconnections and changing of event servers of clients. The causal ordering is defined in Lamport`s *happened-before* relationship [38]. In existing causal ordering algorithms [2, 39, 65, 74, 86], a mobile support station (MSS) that sends messages knows the destinations of messages. Hence a causal ordering algorithm is only executed at the destination MSS. In distributed event notification service systems, as the routing of messages depends on their contents, each event server is only aware of its neighboring event servers. Hence these existing causal ordering algorithms cannot be used for event notification service systems. We propose a causal ordering algorithm [42] in event notification service systems for mobile users to receive events with correct sequence and to avoid loss and duplication of events.

1.2.2 Just-in-time Delivery of Events

In event notification service systems for wireless network, a mobile consumer may disconnect and reconnect to different event servers. Hence, mobility causes that a new event server of a mobile consumer causes the following delays to receive notifications in handoff process.

(1) The previous event server transfers subscriptions of the mobile host to a new event server.

(2) The new event server re-subscribes above subscriptions to establish new paths of notifications.

(3) If the previous event server receives new notifications for the mobile host, it needs to transfer them to new event server during executing handoff.

The handoff (normal handoff) module which is developed in proposed causal ordering algorithm has the above delays. This delay may not be tolerable for many deadline-bound applications like stock trading, auctioning and monitoring of traffics, etc. A mobile user expects to get notifications as soon as it reaches a new location. We term this kind of event delivery as just-in-time delivery of events to deal with data that is valid for limited time period. This problem has not been studied yet and here we propose a concept of location-based pre-handoff [43] that uses regional route map for predicting arrival cells based users' mobility.

1.2.3 Resilient Dissemination of Events

Fault-tolerance is an important feature in a large-scale event notification service system as link or node failures are expected to be more frequent in wide-area networks than in local-area networks. Therefore, fault-tolerance is critical for smooth operation of a large-event notification service system. An event notification service system may experience link failures, event server failures and consumer/producer failures.

Replications of *rendezvous* nodes are described in [8, 61] for *rendezvous* node failures. This strategy provides fault tolerance over *rendezvous* nodes by making use of replicas. In case of link failures, producers/consumers can only continue to publish/receive events in partitioned networks they are located. Reconfigurations that involve the *removal* of a link and the *insertion* of a new one, thus keeping the dispatching tree connected are described in [6, 17, 60]. The main idea is that if an event server fails, its neighbors makes new connection without failed event server. It is based on unsubscribing (to remove a link) and re-subscribing (to insert a new link). Reconfiguration technique follows exclusion of faulted event server and hence the event server facilities are not available to others. We propose a region-based architecture and a fault-tolerance strategy [44] to minimize the impact of failures of event servers and links in event notification service system.

1.3 Contribution of this Thesis

The contributions of this work can be categorized into three parts with respect to the research problems defined earlier. The event notification service system is extended for mobile applications addressing these three problems described in Section 1.2.

Causal Ordering. For event ordering research problem described in Section 1.2.1, we propose a causal ordering algorithm [42] which is based on *vector clock* (i.e., vector of sequence number). The proposed algorithm consists of two modules: the *static* module for normal operation and *handoff* module for mobile hosts migrated to a new location. Each event server maintains vectors of sending sequence number and receiving sequence number for both subscriptions and notifications. An event server appends a related vector of sequence number to every message (subscription/notification) it sends. On receiving notifications or subscriptions, an event server executes causal ordering algorithm before executing other processes. Causal ordering algorithm determines if the notifications/ subscriptions are deliverable to other processes of the event server. Each mobile host also stores the last sequence number of subscriptions/notifications it subscribed/published to its current event server to avoid loss and duplications of subscriptions/notifications during handoff. We prove that the *safety* and *liveness* properties of causal ordering algorithm. In analysis of the causal ordering algorithm, low *message overhead* and *handoff complexity* show that the proposed algorithm is scalable and suitable for mobile environment. Experiments of simulation demonstrate that the causal ordering algorithm causes relatively low cost in terms of latency for delivery of notifications for different network sizes and publication rates in event notification service system.

Location-based Pre-handoff. To solve the research problem of just-in-time delivery of events for mobile users (Section 1.2.2), we propose location-based pre-handoff algorithm [43] that uses regional route map for predicting arrival cells based users" mobility. Each event server maintains mobile hosts location table (MLT), and local border cell table (BCT) besides the routing table (RT). A MLT is a collection of tuples {<mobile_host_ID>, <cell_ID>). In MLT, while a mobile host is moving, its cell ID varies. BCT is a collection of tuples {<border_cell>, <border_event_servers>}. i.e., BCT = {<CB>, <BES>}. Border event servers of a border cell are the event servers located in the adjacent location areas of the border cell.

The main idea of location-based pre-handoff is to create virtual consumers at a set of expected locations (event servers) of a mobile consumer in order to set up flows of notifications before the mobile consumer reaches any of those event servers. An event server uses MLT and BCT in order to determine the border event servers on which pre-handoff processes have to be executed on behalf of consumers who are moving in its border cells. When a mobile consumer gets connected to a new event server on which a **virtual** consumer is already running, it finds that its subscriptions exist and routing paths of notifications are already established. In analysis for the location-based pre-handoff algorithm, we observe the *time complexity*, *message complexity* and *storage complexity*.

The effectiveness and message complexity cost of the location-based pre-handoff are also studied by simulation experiments. Experiments demonstrate that location-based pre-handoff algorithm has a relatively low cost of message complexity.

Region-based Architecture and Fault-Tolerance Algorithm. For resilient dissemination of events described in Section 1.2.3, we propose a region-based architecture and a fault-tolerance strategy on this architecture [44]. A region-based architecture is designed for a fault-tolerant event notification service system. The network of event notification service system is divided into a few big regions which have separate networks composed of event servers. Each region contains multiple event servers and has a region leader event server (RES) that is responsible for communicating other event servers located in different regions. Hence, the region-based architecture consists of three layers: physical network, event server network and RES network. In each region, a central event server (i.e., center node of a graph) is selected as region leader (RES) to minimize the latency of delivering notifications published from other regions. Dissemination of events is differentiated into 2-tiers: inter-region and intra-region. The main idea is that subscriptions of consumers of a region are not forwarded to other regions and notifications are forwarded to reach region leaders of all regions. On receiving a new notification, each RES forwards the same to its neighboring RESs. Hence, the region-based architecture reduces sizes of routing tables at event servers, thus the latency for delivery of notifications to consumers also decreases.

For resilient dissemination of events, a fault-tolerance strategy is presented on the region-based architecture to minimize the impact of faults occurring in event notification service system. The fault-tolerance strategy is based on primary-backup replication models described in [55, 58, 71]. In the fault-tolerance strategy, routing tables of region leaders are replicated at their backup event servers. Hence each backup event server maintains its routing table and a replicated routing table of its region leader. We term the replicated routing table as virtual routing table (VRT). In normal condition, a region leader enables its routing table (RT) and its neighboring region leaders are connected to it. If a region leader RESj or one link between the RESj and its neighboring region leaders fails, the backup event server of RESj enables VRT of RESj to execute the functions of the RES,.

Neighboring event servers of RES, also connect to backup event server instead of RESj. Hence the fault-tolerance strategy makes the system resilient to failures of RESs and links between them. As a notification can reach to every region, the fault-tolerance strategy ensures that a failure in a region has no effect on other regions. Fault-tolerance strategy is analyzed to observe the *time complexity, message complexity* and *storage complexity*. The effectiveness on routing table sizes in proposed region-based architecture is demonstrated with simulations.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows:

Chapter 2 first provides a survey of the background that is necessary to understand the notion of an event notification service system. The example applications, subscription styles and routing strategies are described in Section 2.2. Though there are several event notification service systems, they have commonality in research issues and these research issues are presented in Section 2.3. In Section 2.4. related technologies of event notification service systems are surveyed and classified based on their architectures, subscription styles and support for mobility. Then, some results of important research issues of event filtering, event ordering and security are described in Section 2.5. The event filtering algorithm which is a main feature of each event notification service system can be categorized as three types: Predicate indexing based algorithms, tree based algorithms and XML based algorithms. After that, existing virtual clocks (sequence number of events) and global clock (physical clock) techniques are described to investigate and use for the ordering of events in event notification service system. Finally, some security solutions like access control mechanism and mutual trust between consumers and producers solutions are presented.

Chapter 3 presents a design of event notification service system for mobile computing environment. Firstly it describes event notification service systems that can support mobility. It also identifies the basic requirements for every event notification service system and additional requirements depending on application domains and environments. Then the types of mobility and characteristics of mobile/wireless networks and mobile devices are discussed. In mobile computing environment, the mobile networks can be classified as *wireless network with base stations* (e.g., GSM cellular network) and *ad-hoc network* without base stations (e.g., Bluetooth). This chapter focuses on event notification service system for wireless network with base stations. After that, a conceptual architecture of an event server of event notification service system for mobile network is presented. The services of causal ordering of events (Chapter 4) and just-in-time delivery of events (Chapter 5) are based on this architecture.

Chapter 4 first presents the requirement of causal ordering of events in event notification service systems. Then related technologies for causal ordering of events in mobile environment are described. After that, causal ordering of events for event notification service systems are defined with *subscription ordering* and *notification ordering.* We propose a causal ordering algorithm for mobile environment and it consists *of static* module and *han doff* module. An event server executes static module on receiving a notification/subscription. Handoff module is executed if a mobile host disconnects the current event server and connects to a new event server. The correctness proof of the algorithm is proved by *safety* and *liveness* properties and the algorithm is analyzed to observe message overhead and handoff complexity. Finally, the performance of causal ordering algorithm is evaluated with simulations.

Chapter 5 presents a service for just-in-time delivery of events for event notification service systems in wireless network. Firstly, it describes a model of mobile cellular environment which is a collection of geometric areas called cells. The system model is based on a conceptual architecture presented in Chapter 3. Then regional route map and location tables of mobile hosts are described. Each event server maintains mobile hosts location table (MLT), and local border cell table (BCT) besides the routing table (RT). After that, the proposed location-based pre-handoff algorithm is described with the help of regional route map and location tables mobile hosts. The algorithm ensures that mobile consumers can receive matched notifications immediately after connecting to a new event server. Analysis of the proposed location-based pre-handoff algorithm is presented with the cost of time complexity, message complexity and storage complexity. Finally, the message complexity of the location-based pre-handoff is studied by simulation experiments.

Chapter 6 first describes the related technologies for fault-tolerance in event notification service systems. Then types of failure experienced in event notification service systems are discussed. After that, a fault-tolerant region-based architecture for a large-scale event notification service system is presented. Region-based architecture minimizes the size of routing tables and reduces latency of notification delivery to consumers. We then present a fault-tolerance strategy on region-based architecture so that event notification service system is resilient to failures of event servers and links and ensures dissemination of events to consumers. In the fault-tolerance algorithm, we present two parts: processes executed at region leaders and processes executed at backup event servers. Then time complexity, message complexity and storage complexity of the fault-tolerance algorithm are analyzed. The chapter finishes with a simulation to demonstrate the effectiveness of region-based architecture on routing table sizes at event servers.

Chapter 7 gives a brief conclusion, summarizing the work described in this thesis, and some directions for future work.

Chapter 2

Background and New Challenges[1]

This chapter surveys basic architectures, routing strategies and features of a variety of event notification service systems. We review a number of event notification service systems by presenting background information such as terminologies and by describing their architectures and supported features. It also identifies the basic issues of research interests associated to such systems. Some research results of important research interests of event notification service systems are also discussed.

2.1 Introduction

The purpose of this chapter is to provide the necessary background required to understand the concept of event notification service system. We review a number of event notification service systems by presenting background information such as terminologies and by describing their architectures and supported features. We examine the issues involved in realizing event notification service systems for different environments and describe how these issues have been resolved. Several event notification service systems have been proposed in the literature [4, 8, 13, 18, 31, 56, 61, 87] over different platforms.

On first glance client-server (C/S) model may seem ideal to implement event notification service systems where producers and consumers are clients to servers that receive/disseminate events. But the model may not be suitable for the purpose as the communication in C/S model is typically synchronous and one-to-one (not one-to-many). The service that the event notification service system provides to consumers should not only be asynchronous but should ideally be anonymous. It is possible to develop such a system making use of recent development in computing and communication technologies.

[1] Some work of this chapter is presented and published in (1) *Proceedings of International Conference on Information Technology (CIT 2003)*, Bhubaneswar, India, pages 314-319, December 2003 [45] and (2) *Proceedings of International Workshop on Mobile Commerce in association with International Conference on Emerging Technologies (ICET '03)*, Bhubaneswar, India, pages 71-101, December 2003 [46].

Systems with different names viz. publish/subscribe system [87], notification service [54], message queuing system [56], message brokering system [76], content-based messaging [4], event-based middleware [61] and event-based system [18, 26] have much commonality with event notification service system. These systems even use different terminologies to describe a concept that is common to event notification service system. For example, producer is variously known as producers [27, 87], publishers [8, 80], suppliers [54, 31] and objects of interests [13].

Based on subscription style, the reported event notification service systems can be classified into several categories namely, channel-based systems, subject-based systems and content-based systems. Event delivery that follows content-based mechanism [4, 13, 31, 54, 76, 87] is adaptable to customer needs that vary dynamically. Because of this flexibility, the content-based information retrieval has greater research interest.

In the following, first the basic terminologies, subscription styles and routing strategies of event notification service systems are described in Section 2.2. After that, research issues of event notification service systems are presented in Section 2.3. In Section 2.4, a survey on several event notification service systems reported in literature are described and classified. Then, some research results of event filtering, event ordering and security issues which are important features of event notification service systems are discussed how these issues have been resolved in Section 2.5.

2.2 Event Notification Service Systems

The architectures of event notification service systems can be classified as centralized and distributed architectures. A distributed event notification service system is composed of interconnected event servers connected to producers and consumers over a communication network. The event servers are connected with three basic architectures [13, 14]: hierarchical, acyclic peer-to-peer, and general peer-to-peer. A hierarchical architecture usually represented as a directed tree and each event server only forwards subscriptions to its master. An event server can have many incoming connections of “client” servers but only one outgoing connection of “master” server. An acyclic peer-to-peer architecture allows bidirectional flow of subscriptions and notifications. An event server and its neighbors communicate with each other symmetrically as peers. Like the acyclic peer-to-peer architecture, a general peer-to-peer architecture allows bidirectional communication between two event servers but it can have multiple paths between event servers. The choice of system architectures depends on application types. In next section, we discuss about some interesting applications and important concepts of event notification service systems, further developers have to address the challenges encountered while building such systems. For this purpose we identify these challenges and discuss on the research issues in subsequent sections.

2.2.1 Applications

In a distributed system, processes interact by exchange of messages. The sending of a message can be viewed as an event (notification) by the sender process and the recipient process that acts the message is essentially receiver of the event. This way the occurrence of an event in a process (sender) is asynchronously notified to any other process (receiver) that has declared some interest in it. Clearly, these interactions can be implemented using an event notification service system paradigm. A variety of event notification service systems have been implemented in different application domains and different environments (e.g., real-time, mobile and wide-area network). To fulfill the needs of their application domains, event models differ in their architectures and in the features they support. The event notification service system can be used in many real-life applications such as publishing stock exchanges, real-time avionics mission computing systems, weather and traffic reports, travel updates, alerting services, emergency notifications, and trading services for financial market [4, 13].

An event notification service system is a good candidate for a variety of e-commerce and m-commerce applications [25 j. Some of these applications are as follows:

(1) Business to Consumer (B2C) applications (e.g., on line book store, booking and purchase of airline tickets)

(2) Consumer to Consumer (C2C) applications (a consumer can negotiate with other consumers: e.g., auction)

For example, in an *auction* system if a consumer (including mobile user) is interested in buying a ticket (i.e., buyer) he will subscribe for events that advertise the availability of tickets. Conversely, a producer that wants to sell a ticket (i.e., seller) emits an event of the availability of tickets. If the event matches the subscription, the event notification service system will notify to the consumer. When the consumer receives this notification, he can send a bid for the ticket to the system.

In *Slock Exchange,* selected stock values are delivered only to the interested consumers as notifications. For example, Elvin event notification service system [4] provides *Tickertape* application that notifies stock prices to consumers based on their choices. Unlike traditional internet browsing system that requires users' active participations, the *Digital Library* service systems (which alert passive users on arrival of publications of their interests) have been implemented using the principles of event notification service system in [61].

In [13], it describes simulation studies of event notification service systems with respect to topologies and concludes as follows. The hierarchical should be used where low densities of consumers subscribe/unsubscribe very frequently. The acyclic peer-to-peer is more suitable where the total number of notifications published by producers exceeds the number of notifications interested by consumers. In *stock price dissemination*, the number of notifications published is very high and consumers subscribe/unsubscribe frequently. Hence hierarchical may be suitable in stock applications. In the application domain of *Digital Library*, the number of notification published is low and the number of subscribing/unsubscribing is not frequent. Hence peer-to-peer should be used in services of *Digital Library.*

2.2.2 Notification

On occurrence of an event, or a combination of events, an event notification service system dispatches a notification to all the consumers who subscribed for that event or the combination of events. A consumer can have multiple active subscriptions. After a consumer has issued a *subscription,* the event notification service is responsible for delivering all future matching *notifications* that are published until the consumer cancels subscription to those. In event notification service systems, we can mainly identify two classes of event models as described in [18].

(1) *Tuple-based:* The notifications are defined as a set of strings with a syntax and semantics. For example, tuple with three strings {<Product name> <Release version> <Website>} models an event about digital product.

(2) *Record-based:* These notifications are defined as a set of typed fields characterized by a name and a value. For example,

```
Struct NewRelease{
  string ProductName = "Usefulstuff"
  integer ProductRelease = 4
  string DownloadURL = "http://..."
}
```

is a record-based notification composed of three typed fields.

2.2.3 Subscription Styles

A subscription style defines the form of the selection expression. When subscribing, a consumer expresses its interests in receiving certain messages. The subscription approaches can be classified as follows:

(1) *Channel-based:* Event notification service systems using *channel-based* style require producers to nominate *channels* (i.e., predefined groups) from which consumers may receive their messages. For example, channel = 314, channel = "CNN" and user = "ABC". The first generation of event notification service systems used *channel-based* subscription style. For example, CORBA event service which is implemented before CORBA notification service uses channel-based subscription [54].

(2) *Subject/topic-based:* Subscriptions are specified by indicating the subject of interest by using filter expressions. Subscribing to a subject *T* can be viewed as

becoming a member to a group. Publishing a notification for subject *T* causes a multicast to the consumers of that group. This notification scheme is implemented by using well-established commercial product such as TIB/Rendezvous [80] and JEDI [18]. In these systems, subjects are arranged in hierarchies. A subject tree includes sub-subjects. A consumer's choice of a subject also implies related sub-subjects. All entries of a subject tree are equally visible to all consumers.

(3) *Content-based:* Besides, subject attributes, a consumer can provide constraints on those attributes for specifying the choice more precisely. Event notification service system prepares a filter on the basis of these constraints. For example, a filter for a stock subscription could conjunction of constraints be as shown below:

```
string class > * finance/exchange/
string exchange = NYSE
string symbol = DIS
float  change > 0
float  change < 1.5
```

(4) *Hybrid subscription:* In *Hybrid* subscription style, a subscription is specified by both *subject-based* and *content-based.* A consumer specifies subjects that it is interested in, and also supplies a filter expression that operates on the subjects' attributes. This subscription style is used in Hermes [61]. It bridges the semantic gap between events and programming language types. Event types are organized into event type hierarchies similar to class hierarchies in an object-oriented language.

2.2.4 Routing Strategies

The routing strategies of event notification service systems are classified as follows:

(1) *Notification forwarding:* This strategy broadcasts events to all the servers present in an event notification service system. Event matching is performed at each local server against registered subscriptions received at that server to send notifications for corresponding consumers. This strategy suffers from the drawback that all

notifications are forwarded to all servers, irrespective of that fact whether or not these servers have any consumers interested in published notifications.

(2) *Subscription forwarding*: In this strategy, consumers broadcast subscriptions to all servers. When a producer publishes an event, its local server matches subscriptions to the event and on matching the event is routed to consumers who issued matching subscriptions. The advantage of subscription forwarding is that a shortest path is used to route notifications back to only the local servers of interested consumers. Some event notification service systems [13, 87] use *covering* method as described in Section 2.4.1 to reduce the routing entries and communication traffic.

(3) *Advertisement forwarding:* Producers broadcasts advertisements to all servers to inform the event notification service about the kind of events to be published. When a consumer issues a subscription, its local server matches advertisements to the subscription. If the subscription matches an advertisement, the subscription is routed to corresponding producers. Then notifications published are forwarded along the reverse path of subscriptions. Hence the server forwards a subscription only if there is at least a notification for the subscription. Moreover, if a subscription does not match any advertisement at local server, the event notification service can inform the corresponding consumer that the service does not have any information for this subscription. But in hierarchical architecture, advertisement forwarding cannot be used because advertisements are only forwarded to master server like subscriptions. Hence subscriptions are necessary to reach until root to finish matching with all advertisements.

(4) *Type- and attribute-based routing:* Both second and third strategies need to broadcast subscriptions or advertisements and store them at all servers in the network. These global broadcasts are not scalable and will lead to inconsistent system state when network partitions occur [61]. The solution to the problem is by declaring servers for particular types. Each such server, called *rendezvous* node is attached to a particular type of subscriptions and notifications. First, a producer creates a *rendezvous* node that manages a particular event type. After that,

producers send advertisements to that *rendezvous* node. In order to subscribe a particular event type, a consumer routes its subscription to the *rendezvous* node associated to the event. In this strategy, both advertisements and subscriptions are routed to a *rendezvous* node and they do not need to broadcast to all servers. *Rendezvous* nodes function as meeting points for subscriptions and advertisements.

2.3 Research Issues

Though there have been several implementations of event notification service systems in different environments like local area network, Internet and wireless network with mobile users, etc., still there is a commonality in research issues. In this section we will discuss on features that require to be investigated further for better service.

2.3.1 Real-time Issue

Real-time applications require the real-time constraints (delivery deadlines) defined on the event delivery behavior and duration. Applications like warehouse monitoring, auctions, reservation systems, traffic information systems, flight status tracking, logistics systems, etc., consist of a potentially large number of clients spread all over the world demanding timely information delivery. A user may be willing to receive sports and news information with delay by a few minutes. But may not tolerate delays or out of sequence quotes in stock prices. In order to provide real-time services, techniques like prioritizing events and defining event delivery deadlines are followed. For these applications, an approach is proposed in [43] for mobile users. It guarantees timely delivery of published events to subscribing mobile users by creating virtual consumers at possible future locations decided according to the current locations of mobile users in a cellular environment. CORBA Notification Service [54] provides real-time service by assigning priority and timeout (lifetime) properties to events. But this solution depends on a centralized mediator. Time synchronization among servers is required for real time applications in wide area networks. Hence this issue should be studied for distributed event notification service systems.

2.3.2 Mobility

Portable computing devices such as notebook computers, personal digital assistants (PDAs) and cellular phones along with wireless communication technology enable people to carry computational power with them as they keep changing their physical locations. Due to the decoupling characteristic of event notification service system, the routing of events only depend on the contents of events but not on the identities of users. As and when a new mobile consumer joins, event notification service needs to find paths to the consumer for delivery of notifications and these paths are based on contents of the consumer's interest. There are only a few event notification service systems [13, 18, 79, 87] which support consumer's mobility.

The first event notification service system which supported mobility is JEDI [18]. JEDI has a feature which supports the disconnections and reconnections of components to the system. Event servers manage temporary storages for the duration of the disconnection of clients. Elvin [79] uses a "proxy" to support persistency of events sent to disconnected clients as described in Section 2.4.2. But the limitation of Elvin is that each client has to reconnect to the same proxy from where it disconnected earlier. SIENA has been extended to support mobility [10, 11]. A mobile client connects to the existing event notification service system by using wireless medium such as GPRS (General Packet Radio Service) wireless network. In [87] an algorithm is proposed that extends REBECA service to accommodate mobile clients by maintaining a buffer for all notifications that are not yet delivered for a consumer. To efficiently support mobility, [15, 27] describe an approach to replicate virtual clients at several places to deal with the uncertainty of client movement. Based on virtual client concept, [43] also proposed a solution a mobile host on arrival at a new location can receive notification as soon as it joins. Issues due to mobility could be complex in the case of wireless networks. Locating a nomadic user and provisioning just-in-time message delivery service are of research interests.

2.3.3 Notification Storage

Notifications for a disconnected user have to be stored in the system for delivery in future when the user gets connected. But such provisioning may demand a large storage for saving notifications when either a user remains disconnected for a longer period or the number of users increases or both. In order to minimize the storage requirements, each notification is assigned with time-to-live stamp (expiry time). In periodic interval the expired are purged in [54]. A good compromise between processing overhead and space overhead could be to judiciously estimate the periodic interval on the basis of the frequency of event generation as well as users mobility.

Events are dependent on time in many of applications such as stock trading system, auction system, reservation system, sports reporting and traffic condition monitoring systems. For these applications, if a mobile consumer is disconnected for long time, it causes a large numbers of notifications to queue up for delivery at its event server. When the consumer reconnects to the service, many of these events may be outdated. Hence, by using deadlines of events the buffer size for notification store can be minimized. Moreover, it also reduces the communication overhead between event servers and consumers as the out-of-date notifications are not delivered to the consumers. But it becomes necessary to synchronize the physical clocks of event servers by a clock synchronization protocol like NTP (Network Time Protocol).

2.3.4 Security

In a wide-area network, an event notification service system must handle information dissemination across distinct authoritative domains, heterogeneous platforms and a large, dynamic population of producers and consumers. In traditional systems, security is mostly based on knowing the identities of involved parties, but it is not possible in event notification service systems as they are decoupled in nature. Little research has been done on the security issues of event notification service systems. In current event notification service systems, malicious producers can easily insert bogus notifications which can flood the whole network and malicious consumers can subscribe fake subscriptions causing wastage to bandwidth [50]. Hence, an authorization subjects are identified by credentials in [50j. Producers publish the event together with their credentials to its local event server and the event server decides to allow the publishing by checking the relevant rules of policy list. Similarly, consumers subscribe together with their credentials to the service. In general, an event notification service system must deal with varying security needs allowing diverse policies and mechanisms to be implemented within its infrastructure. The security requirements for an event notification service system can be divided into the requirements for a particular application involving producers and consumers, and for the infrastructure of services [84]. The research interest in security includes *confidentiality, integrity and availability* for applications. Security of the infrastructure primarily includes system *integrity* and *availability.* In some cases, existing approaches can be adopted to achieve these goals, and often with minor modification. Elvin [79] and REBECA [28] protocols have considered these requirements of security in event notification service system. Elvin has its own security layer to encrypt the transmission keys between clients and servers to ensure security. Some research results of security issue are described in Section 2.5.3.

2.3.5 Filtering

In event notification service systems, each event server executes filtering to match notifications (events) and subscriptions for delivering of notifications to consumers or forwarding of notifications to other event servers. As the number of events grows in the network, it becomes important to enable consumers to express their interests more precisely. On the other hand, increasing the accuracy of filters increases its complexity. The complexity of filter directly impacts on the performance of an event notification service. Many event notification service systems such as SIENA, Gryphon and Elvin use content-based subscription style to specify consumers* interests precisely. Content-based routing reduces the size *of* routing tables by using covering of subscriptions. The matching technique should be scalable to handle a large number of subscriptions. Some information filtering algorithms are described in Section 2.5.1. As the performance of an event notification service system mainly depends on efficiency of event filtering, hence investigators tend to put extra effort for developing better event filtering algorithm.

2.3.6 Event Ordering

Because of the randomness in routing of packets over a network (due to several network traffic parameters), the order in transmission of two consecutive messages from a source to a destination, may not maintained. This means that the order in which events are generated is not the same order in which a recipient receives them. The out of sequence arrival of events pertaining to a specific application may not be desirable. This anomaly can be eliminated by implementing ordering of events before delivering. To maintain consistency property in delivery of events, an ordering mechanism for processing of events should be defined. Some methods of event ordering are presented in Section 2.5.2.

In [47] a Generalized Event Monitoring Language (GEM) was proposed for monitoring of communication networks and distributed system. GEM is used to specify the operation of event monitors. It discusses the effect of communication delays on composite event detection and presents an approach for dealing with out-of-order event arrivals at event monitors. In GEM, the user can specify temporal constraints to deal with delays. As GEM uses physical time of event arrivals, it assumes that a well-synchronized global clock exists among components. J EDI [18] describes that events are delivered according to a causal ordering policy in its implementation. It argues that the events generated by a sender are delivered to all the interested recipients in the order these have been published. The ordering requirements of subscriptions and notifications in event notification service systems are defined in [42]. It includes an algorithm which uses *vector clocks* (vectors of sequence numbers) to maintain the correct sequence of notifications and subscriptions.

2.3.7 Scalability and Expressiveness

The *scalability* is a crucial requirement for internet-scale distributed applications. *Scalability* of a system is concerned with the issue required for upgrading/downgrading the system making it adaptable to higher/lower demands on system functionality. For example, one should be able to upgrade an event notification service system when there is an increase in producers/consumers/the number of notifications. Hence the architecture of the event notification service system has to be distributed because any centralized architecture/service becomes bottleneck for upward scalability.

Expressiveness refers to the ability of the event notification service system to provide a powerful data model for specifying events and subscriptions. A subscription language is expressive if it provides basic selection predicates and the ability of combining predicates for the selection of one single event at a time. In practice, scalability and expressiveness are conflicting goals that must be traded off [13, 14]. This issue should be studied for systems in different environments such as LAN/WAN, Internet and wireless communication.

2.3.8 Interoperability

Event notification service system should be language and platform independent to facilitate interoperability between heterogeneous servers connected over network. It should be able to operate in dynamic environment where a variety of fixed and mobile devices join and leave the system at run-time. Interoperability among different devices and infrastructures can be achieved by defining common conceptual abstractions under which they interact in a defined way. Thus, interoperability is an important research issue.

2.4 Related Work

2.4.1 SIENA

SIENA (Scalable Internet Event Notification Architecture) has been designed to support event-based communication in wide-area networks such as the Internet [12. 13. 14]. It is developed by a research group of Politecnico di Milano.

Architecture

SIENA has a logically centralized component that propagates events to the consumers. This logically centralized component is implemented as a set of event servers cooperating with each other to provide event notification service system over a wide-area network. Servers cooperate to each other to select events and deliver them to consumers across a wide area network. SIENA servers arc arranged either in hierarchical or peer-to-peer topology. In case of hierarchical server organization, an event notification needs to traverse to all the nodes of a tree thus causing message traffic congestion for nodes of higher levels. Moreover, every server may potentially become a single point of failure for the whole network. A failure in one server disconnects all the subnets reachable from its parent server and all the subnets (children of parent).

The implementation of event notification service system in wide-area network requires hybrid architecture because of different requirements at different levels of the network. Thus, SIENA provides hybrid of both hierarchical and acyclic peer-to-peer architectures that offer the opportunity to tailor the server/server topologies and protocols in such a way that locality information can be used in application building. The services provided by the system are: (1) *Advertise* (2) *Subscribe* (3) *Publish* and (4) *Notify*.

Delivery of Notifications

In SIENA, the notification model is a record-based structure which consists of a set of typed attributes. Each individual attribute has a type, name and value, and a notification is a collection of attributes and associated values. When a filter is used in a subscription, multiple constraints for the same attribute are interpreted as a conjunction. For example, a subscription which is expressed with

```
string   Stock = "DIS",
integer    Gain > 10,
integer    Gain < 20
```

matches with a notification/notifications publishing DIS stock gain is greater than 10 but less than 20.

The propagation of events is regulated by mechanisms of advertisement, subscription and publication. SIENA has two routing strategies: a subscription forwarding and advertisement forwarding which is described in Section 2.2.4. The routing algorithm of hierarchical architecture is simpler because subscriptions and advertisements are merely propagated along unique paths to the root of the hierarchy. The routing algorithms for peer-to-peer architectures attempt to reduce communication, storage, and computation costs by pruning propagation trees over a network of servers. When a server receives a new subscription, it checks whether that subscription S_{new} is covered by previous forwarded subscriptions S_{old} (i.e., $S_{\text{new}} \subseteq S_{\text{old}}$). Hence, the server knows a set of servers that need to be forwarded the subscription and it forwarded the subscription only those servers.

Ordering of Events

SIENA defines total ordering on events by using global timestamp. It processes events in a sequence defined by ordering. But SIENA cannot support race conditions among causal events because of asynchronous delays in communication of these events. For example, a consumer may send an unsubscribe request after some notifications have already sent to it [14J. This situation violates total ordering of events and thus SIENA does not provide solutions to such race conditions. SIENA assumes that the event service is able to examine events in the right time order and this assumption requires that the event service buffer notifications and shuffle them in the correct temporal sequence within a finite time. Hence, SIENA needs to use a synchronization protocol to synchronize local clocks of event servers.

Mobility

In SIENA, the computing objects like processes, threads, files and servers etc. are identified by its unique name and its location. These objects may change their locations thus causing difficulty in maintaining referential/execution consistency. SIENA designed and evaluated a support service for mobility in its event notification service as described in [10, 11]. This approach is based on a client proxy that acts as an interface to the event notification service system while the client is disconnected. The client calls the *move-out* function on its local mobility interface before disconnection and it causes the local server to transfer stored subscriptions of the client to proxy. When the client reconnects to a new destination, it uses the *move-in* function to instruct its mobility interface to contact a local proxy. Then the

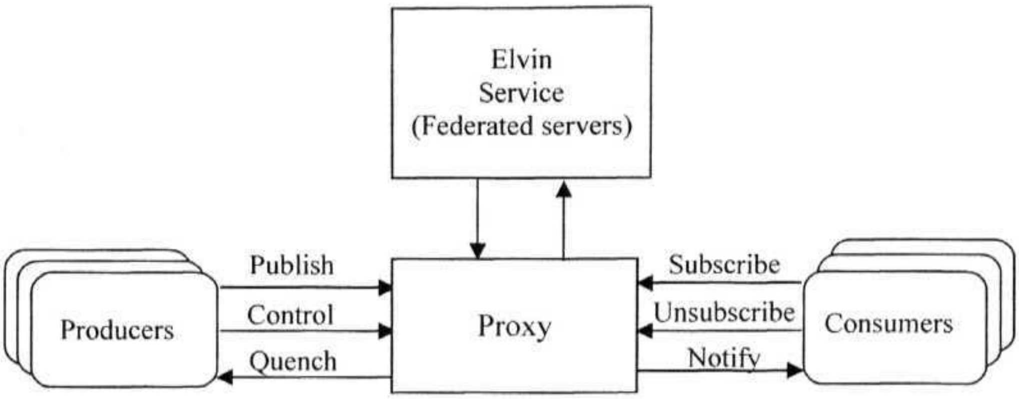


Figure 2.1: Elvin protocol architecture

local current proxy and the remote previous proxy engage in a protocol that results in the transfer of all the subscriptions and all the buffered messages.

2.4.2 Elvin

Elvin event notification service system [4, 70, 79] has been developed at the University of Queensland (Australia). Elvin-3, which provides a means of content-based messaging, was implemented in 1993 and then Elvin-4 has been developed to allow federation of Elvin servers located in both local and wide area networks.

Architecture

Elvin-3 uses client-server architecture for delivering notifications. In Elvin-3 the clients establish sessions with a server and then able to publish notifications for delivery or subscribe to receive notifications. Clients can act as both producers and consumers of information within the same session. A client must maintain a connection to its server to keep its subscriptions active. If the connection is lost, the registered subscriptions are freed by the server and all information about that client is destroyed. In order to take care of this problem, Elvin-4 uses proxy servers between servers and clients. The Elvin-4 "proxy" server works by maintaining a permanent connection to the server and continue subscribing and also receiving notifications on the behalf of the disconnected consumers. Elvin-4 server is implemented in C for UNIX platforms, and client-side libraries are available for C, TCL, Smalltalk, Python, Lisp and Java.

Delivery of Notifications

Elvin-4 is designed to cater notification services to mobile hosts which may get intermittently disconnected to the network. Elvin uses content-based subscription approach similar to SIENA. It allows consumers to specify filters over the subscription database. Quenching is a facility of Elvin to reduce notification traffic by preventing unwanted notifications. Quenching means that the system informs producers on consumers' interests and then producers send notifications that are in demand. In addition, Elvin provides disconnection services to hosts by storing notifications at proxy servers to forward those on reconnection of hosts (Figure 2.1). A user may use different devices to receive notifications and to subscribe its interest. Elvin can handle this situation by allowing participations of several devices (clientIDs) for a user in an application session (SessionID).

Mobility

A consumer needs to maintain a connection to its Elvin server to keep its subscriptions active. If the connection is lost, notifications may be lost. Hence, Elvin proxy maintains a permanent connection to Elvin servers and clients connect directly to proxy. The proxy delivers notifications received from the server to connected consumers and stores any notifications while the consumer is disconnected. But Elvin only supports that the consumers reconnects to the same proxy from which they are disconnected. Hence, Elvin needs to support proxy discovery at different sites to locate a local proxy and a handover mechanism between local proxy and home proxy to resume receiving and sending notifications.

2.4.3 JEDI

JEDI (Java Event-based Distributed Infrastructure) [18] is an object-oriented infrastructure implemented in Java programming language that supports the development of event-based applications. It has been used in a workflow management system called OPSS (ORCHESTRA Process Support System) and PROSYT process support system.

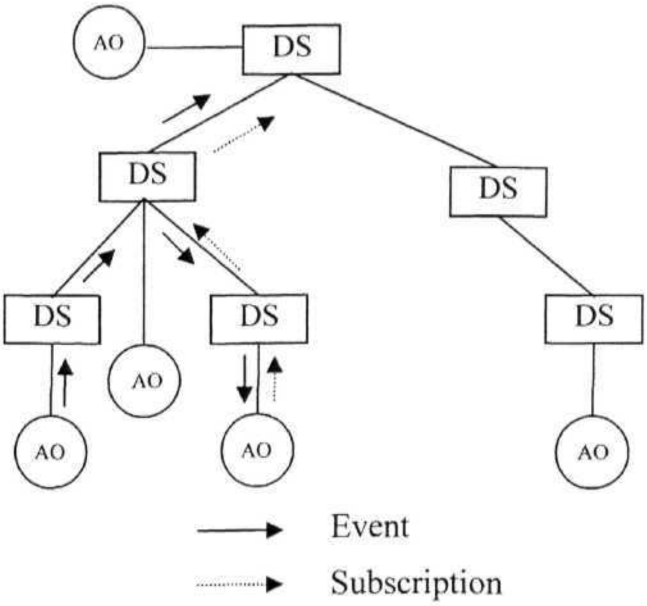


Figure 2.2: Hierarchical strategy of JEDI

Architecture

JEDI is a composition of an event dispatcher (ED) and several active objects (AOs). An active object (AO) is an autonomous entity (i.e., producer or consumer) performing an application-specific task. An event is generated by an AO and sent to the ED. The ED then notifies the event to those AOs that have declared their interest by invoking a subscription operation. The ED must have a global knowledge of all the event notifications that are generated and all the subscriptions that are declared. A centralized implementation of ED is not suitable for a distributed system. Because a centralized implementation of ED causes bottleneck and also becomes vulnerable to system failure. As a solution to this problem, an ED is split into Dispatch Servers (DSs). JEDI has a hierarchical architecture with DSs as tree nodes. Each AO is connected to a DS and AOs are not necessary to located at the leaf nodes as shown in Figure 2.2.

Delivery of Notifications

JEDI uses subscription forwarding routing strategy. The subscriptions are propagated only upward in the DS tree as shown in Figure 2.2. Thus, when a subscription is generated by an AO and sent to a DS, only the ancestors of that DS will eventually receive it. When a

DS receives a notification from one of the AOs that are connected to it, it propagates the notification to the following:

- its parent,
- a subset of its descendents and
- the AOs that are directly connected to it if these AOs register a subscription that matches the received notification.

A DS on receiving a notification from another DS, forward the notification to the parent DS only. Thus the root dispatcher acts as an attractor causing flow of all the notifications to the root. To relief this situation, JEDI is implementing advertisement forwarding strategy as described in Section 2.2.4. JEDI follows tuple-based description of subscriptions. A subscription/notification in JEDI is represented by a tuple with name and related attributes. For example,

print (MyDocument, OurLaserPrinter) where *print* is the event name and MyDocument and OurLaserPrinter are the event parameters. Subscription *print* *(^, -*)* would match all notifications whose name starts with print and that have two parameters.

Ordering of Events

JEDI guarantees only a particular form of partial ordering among events, i.e., *causal ordering*. Events *E* and *El* are delivered according to a *casual ordering* policy which is described in [38], i.e., if £1 causes £2, then any AO registered to receive both *E* and *El* must receive *E* after *El* and not vice versa. The required causality is the relationship among events generated by the same AO. Thus, JEDI ensures that the events published by a producer are delivered to all the interested consumers in the order they have been published.

Mobility

JEDI offers two operations to handle mobility of active objects AOs: *moveOut* and *moveln* operations. By invoking the *moveOut* operation an AO is able to temporarily disconnect from its DS. Through the *moveln* operation, the AO can either reconnect to the DS it was initially connected to or it can connect to new DS. While the AO is disconnected the DS stores the events which the AO subscribed. The new DS engages a direct communication with the old DS in order to obtain information about all subscriptions issued by the AO and to receive all events that are waiting for delivery. Moreover the new DS sends subscriptions of the AO to its parent DS to update routing tables. Hence AO receives the new events through a new path.

2.4.4 Hermes

Hermes [61, 62, 63] is an event-based middleware that provides a platform to build large-scale distributed publish/subscribe system which is implemented by Computer Laboratory, University of Cambridge. It uses hybrid subscription style (Section 2.2.3) to bridge the semantic gap between events and programming language types.

Architecture

In Hermes, event servers are interconnected with each other with peer-to-peer topology and use message passing to communicate with their neighbors. Hermes uses *rendezvous* nodes in the network, which are special servers that are known to both publishers and consumers. For each event type, a *rendezvous* node exists in the network. To find a particular *rendezvous* node, a hash value of the event type name is calculated, and the result is the node ID of the *rendezvous* node. Advertisements and subscriptions are then routed to the *rendezvous* node. Thus, unlike Elvin and SIENA, the knowledge on all subscriptions/ advertisements is not required. Based on application requirements in case of Hermes, a logical network can be defined on application servers such that application messages can be routed over this network. *Rendezvous* nodes are replicated to prevent from single point of failure. If an event client (producer/consumer) does not receive an acknowledgement from a *rendezvous* node after sending a subscription or advertisement, it will try contacting another replica.

Delivery of Notifications

Hermes applies *covering* method among subscriptions similar to SIENA [13]. An event server only passed on an advertisement/subscription if an equivalent or more general advertisement/subscription has not already been propagated. Hermes uses type- and attribute-based routing strategy described in Section 2.2.4. Due to introduction of *rendezvous* node, it does not require global broadcast like other event notification service systems such as SIENA, Elvin, and JEDI. Thus, it is more scalable than those systems. But Hermes needs type messages beside advertisement, subscription, and notification messages. Type messages are published by producers to set up *rendezvous* nodes for event types (i.e. subjects). A type message contains the definition of an event which is stored at a *rendezvous* node. Event notifications published by producers can be type-checked against this definition.

Each event type is managed by a *rendezvous* node. Event types are organized into *event type hierarchies* similar to class hierarchies in an object-oriented language. Thus, when a new event type is added to the system, a parent event type should be specified. A *rendezvous* node sends every subscription to the *rendezvous* nodes of all its descendent types.

2.4.5 REBECA

REBEC A (REBECA Event-based Electronic Commerce Architecture) content-Based Publish/Subscribe Middleware [27, 53, 87J is developed by Databases and Distributed Systems Group, Darmstadt University of Technology (TUD), Germany.

Architecture

Communication topology of the REBECA is a graph which is acyclic peer-to-peer. REBECA distinguishes two types of servers via border brokers and inner brokers. Each broker maintains a routing table which includes subscription/link pairs to use content-based routing strategy. Border brokers are the boundary of the REBECA system and maintain connections to the clients. Hence they forward the subscriptions and notifications to other inner/border brokers or deliver notifications to related consumers by checking the routing table. As inner brokers do not maintain any connections to clients, they are connected to other inner/border brokers and forward subscriptions and notifications to other inner/border brokers.

Delivery of Notifications

REBECA uses *covering* test for subscriptions like SIENA [13] and Hermes [61]. In [52], REBEC A introduced the idea of subscription *merging* method to reduce the size of routing tables and communication traffic. When a border broker receives a new subscription, it first executes covering test. If the covering test cannot find subscriptions which cover the new subscription, merging strategy is used to create a new subscription that covers of existing ones. Only the resulting merged subscription is forwarded to neighbor brokers. Every incoming notification is tested against the routing table's entries to determine the set of links with matching subscription. Then the notification is forwarded to the respective next broker along these links.

Mobility

REBECA supports the mobility based on two assumptions. First, brokers are able to install and maintain a buffer for all notifications that pass through them for a certain period of time. Second, the underlying routing infrastructure uses advertisement forwarding strategy (in Section 2.2.4) to efficiently reroute notifications to a moving consumer [87].

After a consumer has detected the change of location it reissues subscriptions automatically to new broker. Neither consumer nor new broker need to have any knowledge about the old location. When a new broker propagates subscriptions of the consumer for establishing new paths of notifications, the old and new paths from a producer to a consumer meet at one joint broker. The junction broker is aware of this by its routing table, its list of received advertisements, and comparing it to the subscription received. Then the junction broker sends a fetch request along the old path to previous broker of the consumer. The old broker replays all buffered notifications of the consumer to reach the new broker through the junction broker. In a mobile environment, redirecting buffered notifications from old location to new location takes time and this could be a drawback which is later overcome by following pre-subscription approach [15]. In pre-subscription approach, a broker which could be a probable destination of a host sets up virtual hosts (of the host) and subscribes so that notification flow for the host at probable destinations take place.

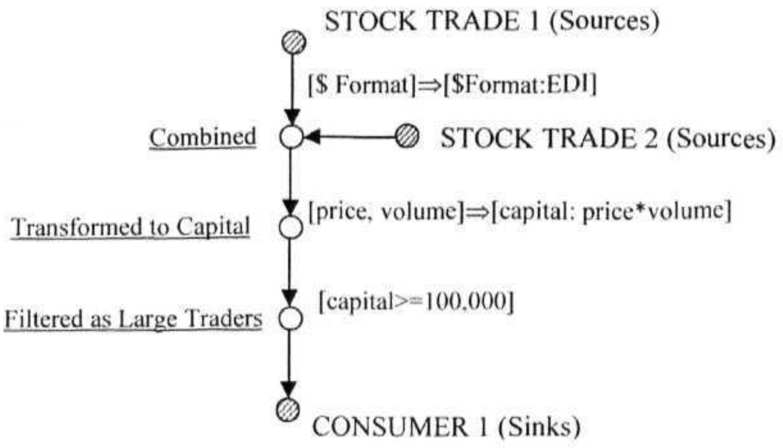


Figure 2.3: Sample information flow graph of Gryphon

2.4.6 Other Related Work

Gryphon [76J, research project of IBM, is a distributed content-based message brokering system. It maps subscription database to a network. In Gryphon, the flows of streams of events are described via an information flow graph. An *information flow graph* is a directed acyclic graph constituting an abstraction of the infomiation flow in the system. The information flow graph specifies the selective delivery of events, the transformation of events, and the generation of new events as a function of *state* computed from event histories. Event transformation means converting events by projecting and applying functions to data in events. In the example of Figure 2.3, two stock trades are combined, transformed, filtered and delivered to a consumer. The two sources produce events of type [price, volume], which are combined into a single stream. Then a new stream of events of type [capital] is computed (transformed). Next, events with capital less than $100,000 are **filtered** out to deliver to related consumers.

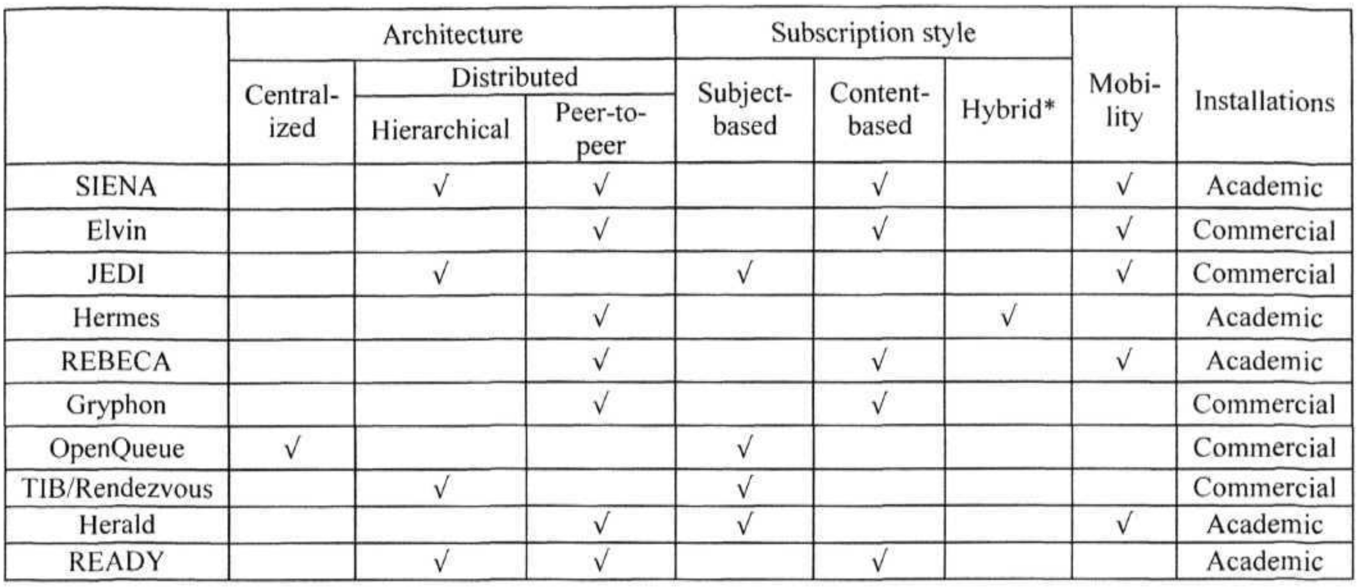
Gryphon uses a record-based event model and subscription-based routing strategy. It propagates every subscription everywhere in the network. Each node in the graph is called an *information space*. Each information space has a schema called its type, which defines its contents. The type system consists of atomic types (numbers, strings), tuples, bags, lists, and unions (tagged variants).

OpenQueue [56] is an open source protocol for publish/subscribe message queuing. OpenQueue is designed to provide real-time, reliable, and transactional message queuing services based on event notification service model and is used for notification and MIME (Multipurpose Internet Mail Extension) message delivery. OpenQueue is based on centralized client/server architecture and it uses subject-based subscription style. Any type of MIME-based data (HTML, XML, GIF, etc.) can be sent by publishers. When a consumer reconnects to a server after disconnection, the server sends all messages queued for that client while it was off-line. If message delivery fails for any reason, it is "rolled back", and the server will retry the delivery until it succeeds. This ensures that each message is guaranteed to deliver in the publishing order. Since OpenQueue is text-oriented, it is relatively easy to write interfaces to it in different languages. Currently there are OpenQueue tools written in Java, Perl, and Tcl.

TIB/Rendezvous [80] is an established messaging middleware with worldwide installations. In past years, it has been used to integrate financial and banking applications to share data across LANs and WANs. It uses subject-based subscription style. TIBCO's service uses a decentralized peer-to-peer architecture. Producers store messages until each consumer has acknowledged receipt. Events are composed of a set of typed data fields consisting of strings of tuples: fieldname, datatype, length, value. Programming interfaces support the languages such as Java, C, C++, ActiveX and Perl.

Herald [8] is a scalable global event notification service system that is being designed and built at Microsoft Research. Herald describes that there will not be a single organization that owns the entire event notification service infrastructure. Hence the Herald service is being implemented as federation of servers which are interconnected with peer-to-peer topology. A *rendezvous point* is a Herald abstraction to which producers send event notifications and consumers declare their interests. Since a rendezvous point is created on a particular subject (type) Herald uses subject-based subscription style for event delivery.

Herald provides for scalability and fault tolerance by replicating rendezvous points. For scalability, when a rendezvous point starts causing too much traffic at a particular server, some or all work of that rendezvous point is moved to another server. By replicating rendezvous points at two or more servers, Herald provides a degree of fault-



	Architecture			Subscription style			Mobi-lity	Installations
	Central-ized	Distributed		Subject-based	Content-based	Hybrid*		
		Hierarchical	Peer-to-peer					
SIENA		√	√		√		√	Academic
Elvin			√		√		√	Commercial
JEDI		√		√			√	Commercial
Hermes			√			√		Academic
REBECA			√		√		√	Academic
Gryphon			√		√			Commercial
OpenQueue	√			√				Commercial
TIB/Rendezvous		√		√				Commercial
Herald			√	√			√	Academic
READY		√	√		√			Academic

*It is a combination of subject- and content-based subscription style. *Rendezvous* nodes depend on subjects.

Table 2.1: Classification of event notification service systems

tolerance that continues communication when one of those servers becomes unavailable. Herald also supports for disconnection by queuing events for disconnected clients until they reconnect. It also allows a consumer to request for keeping a history of published events while the consumer is disconnected. The consumer can send a request by indicating how much history it wants to keep and servers are free to either accept or reject the request. Herald does not directly support an event filtering service and it supports a query language such as SQL as a separate service for data retrieval.

READY event notification service system [31] is developed at AT&T Lab. Ready is implemented using one or more event domains and each event domain may contain multiple servers. Boundary routers link two or more event domains and each domain may use several routers. Boundary routers can be linked together in either a hierarchical or peer-to-peer topology depending on the characteristics of applications.

Clients interact with READY system using sessions. Admin sessions are used for creating/destroying producer sessions, consumer sessions, session groups, and adding/removing session to/from session groups [31]. Producer sessions are used to publish events to the service and consumer sessions are used for registering/removing subscriptions. A consumer session can be disconnected from the service. Hence READY can support disconnected consumers by retaining notifications until the consumer session is reconnected at which point notifications are delivered. READY only considered the ordering of events received within some time window and these events are ordered according to ordering constraints.

2.5 Some Research Results

The issues of event notification service system are described in Section 2.3. Among them, some research results of event filtering (matching), event ordering and security issues are described in this section.

2.5.1 Event Filtering

Event notification Service System performance is primarily dependent on efficiency of event matching or filtering algorithm. An event matching algorithm should be scalable so that the increasing number of subscriptions do not cause considerable degrade in system performance. Table 2.2 describes the filtering (matching) algorithms related with event notification service systems.

Formally, we represent events and subscriptions as

$$\text{Event } e \equiv \{(A_i, V_i)\}_{i=1}^{n}$$

is a collection of pairs of attribute A, and its value V_i.

$$\text{Subscription } S \equiv \{P_i\}_{i=1}^{m}$$

is a set of predicates where a predicate P, is defined as a triplet

$$P_i \equiv (<A_i>, <operator>, <V_i>).$$

The operator is usually a relational operator (<, >, =, !=, < etc.). Some other operators like '*contains*' and '*a kind of* are also have been used in [68] for specifying predicates. In [81] substring matching algorithm is used for matching of string attributes. Event matching problem can be stated as a problem of finding *S* such that predicates of *S* are satisfied by an event (using event attributes and associated values). If all the predicates of a subscription are satisfied by an event the subscription is chosen for notification.

The performance of event matching algorithm in content-based subscription system depends on how fast the predicates are identified and evaluated. The reported algorithms are divided into three categories:

(1) Predicate indexing based algorithms

(2) Tree based algorithms and

(3) XML based algorithms.

2.5.1.1 Predicate Indexing Based Algorithms

Predicate indexing based algorithms work in two phases. In the first phase for a given event all the predicates due to all consumers are fetched. These predicates are evaluated and subscriptions that match with the event are determined in the second phase of the algorithm.

Predicate indexing algorithms use a set of one-dimensional index structures to index the predicates of the subscriptions. Predicate indexing algorithms are different from each other by whether or not all the predicates in the subscriptions are placed in the index structures. For example in the counting algorithm described in [59], a one-dimensional index structure is defined on all the predicates available in subscriptions. The Hanson *et al.* algorithm [32] is an example where not all the predicates in the subscriptions are placed in the index structures.

The counting algorithm

In Counting Algorithm described in [59], a one-dimensional index structure is defined on all the predicates available in subscriptions. Predicates are classified into several families such that all the predicates belong to a predicate family which has the same attribute and operator. For example, a set of subscription *S* consists of two subscriptions S_1 = [(price = 20), (quantity < 10)], S_2 = [(price_< 12) J then there are three predicate families: (price =), (price ≤), and (quantity <). Different kinds of data structures are used to arrange predicates making suitable for searching predicates. For example, hashing provides fast access to equality predicates, but it is not usable for non-equality predicates. B+-trees or IBS-trees (Interval Binary Search trees) are more suitable for non-equality predicates. IBS-tree is an extended binary search tree for indexing both interval and point data. Detail of IBS-tree is described in [32].

When an event arrives, the first phase of algorithm computes the list of satisfied predicates. In the second phase, for each satisfied predicate, the set of subscriptions containing that predicate is fetched. For each one of these subscriptions, its number of satisfied predicates is incremented by one. If all predicates associated to a subscription are satisfied by an event then corresponding consumer is notified.

The Hanson et al. algorithm

The *Hanson et. al.* algorithm described in [32] is an improvement of counting algorithm that incurs the overhead of considering all subscriptions. The counting algorithm evaluates all the predicates of subscriptions to get the list of satisfied predicates. But, in [32] the algorithm uses the *most selective predicates.* The *most selective predicate* of a subscription is the predicate which is used for the highest number of times in subscriptions. The algorithm chooses the most selective predicates for each subscription and places it in the corresponding one-dimensional index which is the IBS-tree. IBS-tree allows efficient searching to determine which interval and equality predicates match a value. The improvement over the counting algorithm is that the Hanson *et al.* algorithm considers only those subscriptions whose most selective predicate is satisfied.

When an event arrives, the algorithm uses the set of one-dimensional indexes to find all the subscriptions whose most selective predicates are satisfied. In the second phase for each of these subscriptions, the remaining predicates are checked to find out matching subscriptions. If the number of satisfied predicates associated to a subscription is equal to the number predicates of event, the event matches with this subscription.

The cluster propagation algorithm 1

The cluster propagation algorithm in [59] improves the Hanson *et al.* algorithm. The goal of this algorithm is to limit the number of subscriptions that have to be evaluated. In a pre-processing step the subscriptions arc grouped into clusters in the following way: For each cluster C,

(1) There exists the most selective predicate *p* of every subscription occurring in C.

(2) All subscriptions in C have the same number of predicates (i.e., size of C).

Hence, each cluster groups subscriptions that have the same most selective predicate/? and the same number of predicates. The subscriptions in the cluster list associated to the most selective predicate *p* need to be checked if and only if *p* is satisfied. Within each cluster, the remaining predicates for each subscription are stored in decreasing order of selectivity (i.e., from the second most selective to the least selective predicate).

When an event arrives, the algorithm uses predicate matching to compute a set of predicates satisfied by the event. Then the algorithm evaluates a set of clusters whose most selective predicates are satisfied with an event by the predicate matching algorithm. The algorithm uses an association table between the satisfied predicates and clusters. In the second phase, only subscriptions contained in selected clusters have to be checked. For each cluster, a cluster *propagation strategy* is used to evaluate the rest of the predicates for the subscriptions contained in the cluster. In cluster propagation strategy, the second most selective predicates are evaluated. The subscriptions which are not satisfied this step are eliminated. The subscriptions satisfied are considered for the next step where the third most selective predicates are evaluated, and so on. A subscription which succeeds all steps is a matching subscription and the event is notified to the consumer.

The cluster propagation algorithm 2

The filtering algorithm proposed in [23] groups subscriptions into clusters and is similar to the cluster propagation algorithm 1. But it does not restrict a cluster's access predicate (i.e., the most selective predicate) to be composed of a single predicate. This algorithm groups together subscriptions in terms of their number of predicates and a common conjunction of equality predicates as an access predicate. It denotes the set of equality predicates of a subscription *S* by *P(S)* and the *schema A(S)* represents the set of all attributes occurring in the equality predicates *of S.* For example, for the subscription "*S* = (movie, titanic, =), (price, \$10, ≤), (price, \$5, ≥)" *P{S)* = (movie, titanic. =) and A(*S*) = movie. The algorithm uses *multi-attribute hashing structure* to find out the relevant clusters when an event occurs. Each hashing structure is intended to check predicates having a schema (i.e., an event is checked in a hash table whose schema is included in the schema of the event). Subscriptions and event patterns may change over time and hence the algorithm uses *dynamic clustering* that incrementally adapts clustering to changes in subscriptions and event patterns. The detailed description of hashing configuration and dynamic clustering are described in [23].

2.5.1.2 Tree Based Algorithms

In tree-based algorithms subscription predicates are arranged in a tree in such a way that nodes in a tree path from root node to a leaf node represents all the predicates associated to a subscription to which the leaf node points. On arrival of an event, the matching of the event starts from the root node and passes through all intermediate nodes to reach a leaf node where a reference to a matching subscription is stored.

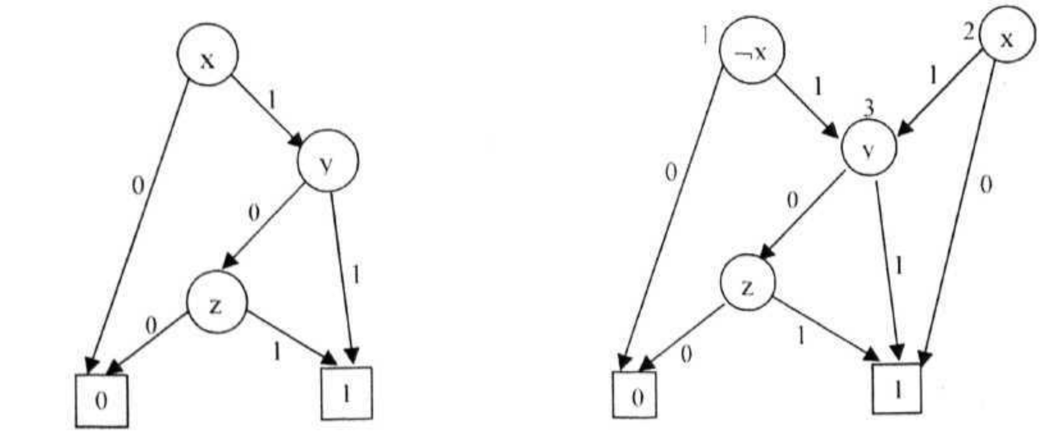
Gryphon content-based subscription system

The matching algorithm used in Gryphon content-based subscription system is proposed in [1]. In the pre-processing phase, the algorithm creates a matching tree for all the subscriptions. In the matching tree, each non-leaf node is a test on some of the attributes, and the edges are results of such test. Each lower level of the tree is a refinement of the tests performed at higher levels, and the leaf nodes of the tree contain subscriptions. The matching can also have special "do not care edges" called *-edges that do not care about the result of the test. When an event occurs, the algorithm finds the matched subscriptions by traversing the tree starting from the root. At each node, the algorithm performs the test prescribed by the node and follows all edges that represent the result of the test, and *-edge if it is present. If an event of matching process reaches to some leaf nodes, subscriptions at these leaf nodes match the event. This algorithm is used in Gryphon content-based message brokering system [76].

Algorithm based on binary decision diagrams

The event matching algorithm that uses *binary decision diagram*, proposed in [9] is suitable for large scale event notification service systems. Binary decision diagram (BDD) is a compact data structure for representing Boolean functions. Suppose A is the set oi propositional variables and “π” a linear order on A. An ordered binary decision diagram (OBDD) over A is an acyclic graph (V, E) whose non-terminal vertices (nodes) are labeled by variables from A. The edges and terminal nodes are labeled by 0, 1. Each non-terminal node v has out-degree 2 as it represents a boolean function. The terminal nodes represent the constant functions given by their labels. For example, the BDD in Figure 2.4(a) represents the Boolean function x and (y or z). The variable ordering is x π *y* π Z. The algorithm uses shared BDDs that represent shared subfunctions of subscriptions. Figure 2.4(b) represents a shared BDD. The two root nodes 1 and 2 represent the function x and (y or z) and ¬x and (y or z) respectively. Node 3 represents the function (y or z).

In this scheme BDD represents subscriptions where each internal node of a BDD represents a predicate of a subscription *S.* It is expensive to represent a subscription in a BDD. In order to lessen storage requirement, BDD of a predicate may be shared among more than one subscription (i.e., shared BDD).



(a) A BDD for function x AND (y OR z)

(b) A shared BDD

Figure 2.4: Binary Decision Diagrams

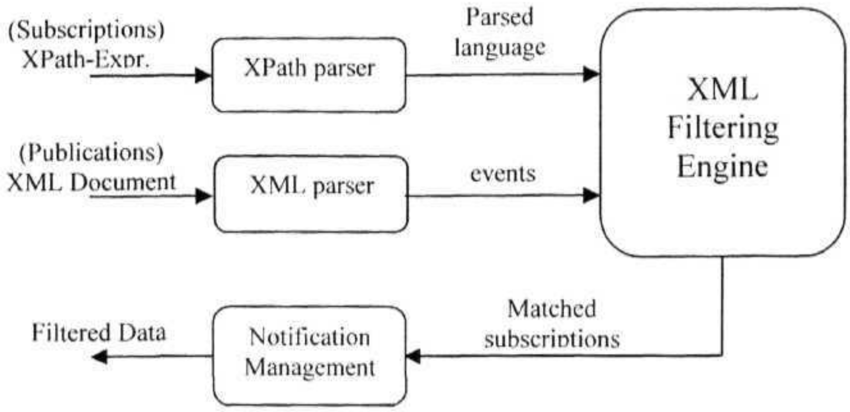


Figure 2.5: Architecture of a XML filtering system

2.5.1.3 XML Based Algorithms

Interoperability between different devices and infrastructures is achieved by using common data presentation format and standards like XML [24]. XML (eXtensible Markup Language) can be used on any platforms because it is text-based, non-binary format and use syntax to organize data. XML allows organizing information of a document into hierarchical structure which has a root element that includes sub-elements. The elements can be nested to any depth. Figure 2.5 describes the basic architecture of XML filtering systems.

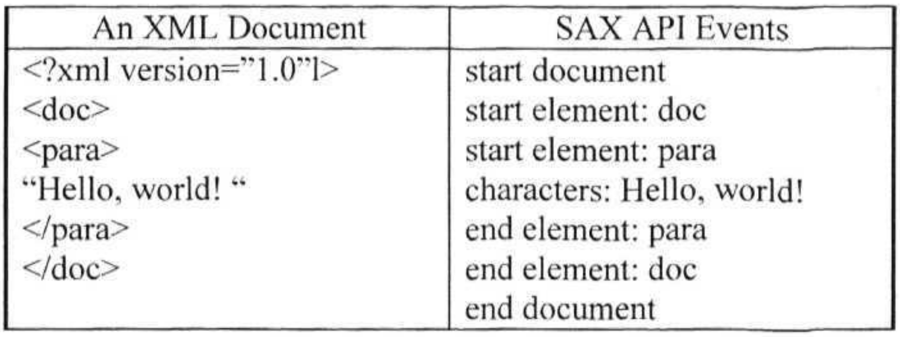
XPath is a language for expressing user subscriptions to filter the XML messages they are interested [83]. XPath also allows to use a wildcard operator "*" which matches any element names. An alternative of using XPath is XML query language *XQuery*. For example,

S1 = //real-estate/*/bedrooms[number = 2]

In the above XPath expression, the consumer is interested in real-estate objects of unspecified kind which has two bed rooms.

XPath parser converts subscriptions (XPath-expression) into a format that can be efficiently stored and evaluated by the filtering engine. New subscriptions can be added to a filtering engine only when the engine is not actively engaged in processing a document.

XML parser identifies events to be executed in the filtering engine on processing XML document.



An XML Document	SAX API Events
<?xml version=”1.0”l>	start document
<doc>	start element: doc
<para>	start element: para
“Hello, world! “	characters: Hello, world!
</para>	end element: para
</doc>	end element: doc
	end document

Figure 2.6: SAX API example

XFilter

XFilter [3] is a document filtering system that provides highly efficient matching of XML documents to large numbers of subscriptions. In XFilter, subscriptions are represented as XPath queries and events are represented as XML documents. XFilter converts each XPath query into a *Finite State Machine* (FSM) which reacts to XML parsing events. In the XFilter, a subscription is considered to match a document when the final state of its FSM is reached.

XML parser used in XFilter is based on the SAX event-based interface, which is a standard interface for event-based XML parsing [69]. Figure 2.6 shows an example of how a SAX event-based interface converts the structure of an XML document into sequence of events to use these events in subscription matching process. The events are parsing events such as the start and end tag of an element.

WebFilter

WebFilter [24] is proposed to use at large-scale and high-throughput XML processing for selective information dissemination on the Internet and in mobile environments. Similar to XFilter [3], WebFilter considers publications (events) which are XML documents and subscriptions which are XPath expression. Subscriptions are maintained in a database using suffix array which allows a binary search. In XML parsing phase, WebFilter processes XML-documents to recognize events that consists of attribute-value pairs. The algorithm uses *dynamic clustering* strategy which is similar to cluster propagation algorithm2 described in Section 2.5.1.1 for dynamic changes of subscriptions and event patterns.

2.5.1.4 Other Types

Some filtering algorithms [68, 81] use neither indexing scheme nor tree structure.

Websphere

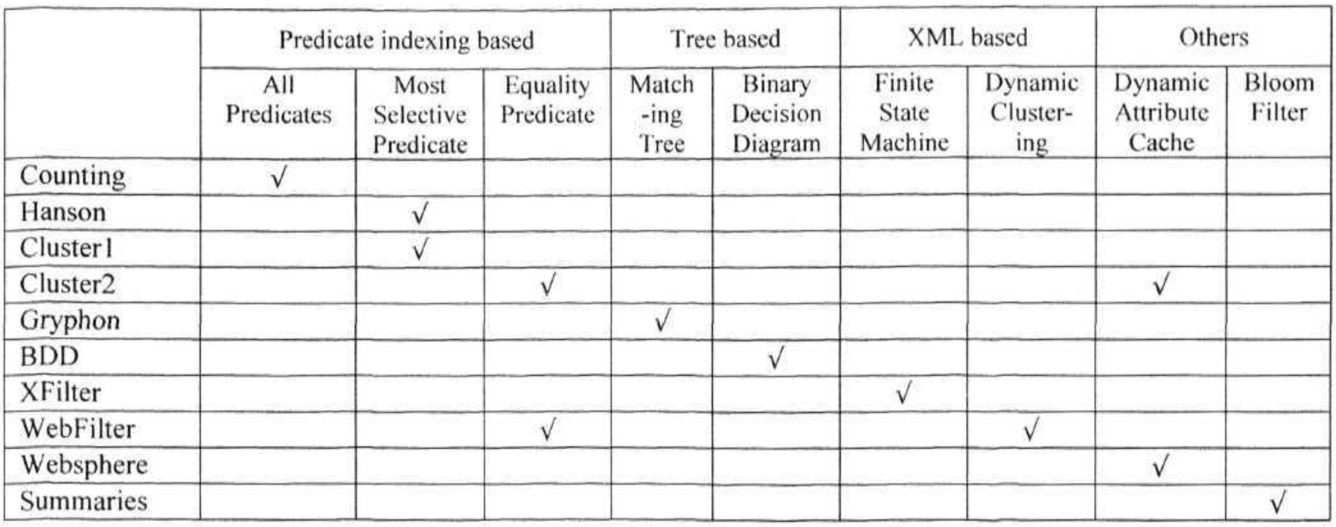
Websphere [68], research project of IBM, introduces the novel concept of *symmetric* subscription system where all consumers and producers can specify their choices. A producer can specify the consumers for whom its events could be useful and at the same time a consumer specifies choices of events it is looking for. The choices are specified by conjunction of predicates where each is triplet (A,, operator, V_i) as descried in event filtering of Section 2.5.1.

The key component of WME (Websphere Matching Environment) is its *symmetric* matchmaking engine (MME). Producers use the WME advertising interface to advertise products and services to the MME. Consumers use the WME query interface to subscribe to the MME. Because of the symmetric nature of the MME, both advertisement and query consist of three arguments. These are

(1) Matchmaking union name: It is a string representing the type of the product or service advertised/queried. The name must be a valid name as defined in the system's data dictionary.

(2) Properties: A property is a list of attribute-value pairs describing product or service advertised/queried. If the value of an attribute needs to be computed dynamically (i.e., dynamic attribute) during the matching process, the value place holder contains a rule (program) that is used to compute the attribute's value.

(3) Rules: Rules are set of programs written in script language to compute attribute values that are not available or change dynamically.



	Predicate indexing based			Tree based		XML based		Others	
	All Predicates	Most Selective Predicate	Equality Predicate	Match -ing Tree	Binary Decision Diagram	Finite State Machine	Dynamic Cluster-ing	Dynamic Attribute Cache	Bloom Filter
Counting	√								
Hanson		√							
Cluster1		√							
Cluster2			√					√	
Gryphon				√					
BDD					√				
XFilter						√			
WebFilter			√				√		
Websphere								√	
Summaries									√

Table 2.2: Classification of filtering algorithms

WME provides a mechanism called *dynamic attribute cache* that stores the values that are computed using rules in order to reduce search time. The details on dynamic cache management can be found in [68].

Subscription summaries algorithm

For scalable event notification service system, the subscription information stored for event matching needs to be as compact as possible. The goal of subscription summaries algorithm [81] is to save network bandwidth and reduce the processing time at the server by making subscription information compact. The algorithm is based on *Bloom filters.* A *Bloom filter* is a method which a vector V containing *m* bits (initially all set to zero), is used for representing information in a set A = $\{a_1, a_2. \ldots . a_n\}$ by hashing each value into V. For each element a, e A, the bit positions $h_1(a_i), h_2(a_i). \ldots . h_k(a_i)$ in V are set to 1. A particular bit might be set to 1 multiple times. Given a query for element b, the same k hash functions are applied on b and the bits of V in positions of $h_1(b), h_2(b), \ldots, h_k(b)$ are checked. If at least one of these bits is 0. then b docs not belong to A. Otherwise, it is possible that b is in the set A although there is a certain probability that may be wrong. The parameter k and *m* need to be chosen such that the probability of a false positive is acceptable.

In the pre-processing four data structures are used to hold the key information about the subscriptions received by a specific server. These data structures are

(1) Subscription Attribute Summary (SAS): SAS stores all the attribute names of subscriptions. The algorithm checks if each attributes of the event matches the attribute names of SAS when an event arrives. If SAS consists of one attribute name of the event, the algorithm continues to check Attribute Association List.

(2) Attribute Association List (AAL): AAL stores information to link between the names attributes (attribute ID) and subscriptions. The algorithm uses AAL to check the other attributes of subscriptions which have a matched attribute name from SAS. Only if all attribute names of a subscription are included in the event, this subscription may be possible to match the event. Then the algorithm checks if values of the associated attributes are matched by using AACS and SACS.

(3) Arithmetic Attribute Constraint Summary (AACS): AACS stores the range (i.e., min and max values) or value of each arithmetic attribute of a subscription.

(4) String Attribute Constraint Summary (SACS): SACS stores the values of subscription's string attributes. SACS structure consists of three bit vectors as Bloom filters because string attributes may include substring (*), prefix (>*), and suffix (*<) operators.

If there is a match for all associated attributes of subscriptions, the algorithm notifies the event to the consumers.

2.5.2 Event Ordering

A distributed event notification service system is an asynchronous system consisting of several subsystems viz. producer clients, consumer clients and servers without common memory and global clock. The subsystems communicate each other by sending messages with unpredictable transmission delays causing unordered propagation of messages. Ordering of events is also required to avoid the loss or duplication of events besides forming the correct sequence of events in an event notification service system.

In a large-scale network, an event notification service system has unpredictable bounds and large variations on transmission delay. Two events can be delivered to a recipient without passing through the same serialization point. This means that the order in which events are generated is not the same order in which the recipient receives them. In order to maintain consistency in event notification service system, an event ordering process should be defined. For example, two events A and B that sent a request at time t_1 and t_2 respectively, being $t_1 < t_2$. Suppose d_1 and d_2 are the time needed by the requests to reach a server. Due to the variable latency of the network, it may cause that $d_1 > (t_2 - t_1 + d_2)$. In this case, the server observes two events are not in order. The order of event delivery can be guaranteed only if events are tagged with a timestamp (i.e., a global clock among all event servers is assumed) and the communication network provides a guaranteed fixed latency time. In practice, global clock is achieved by synchronizing local clocks of servers to a given time and such synchronization is not always possible because of drifting in local clock times.

SIENA [13, 14] describes it uses global timestamp and JEDI [18] guarantees causal ordering of events published from same source. But no algorithm is described how ordering of events is maintained. [42] proposed an algorithm to maintain causal ordering of events among event servers. It also provides to prevent loss or duplication of events during handoff of mobile users. We describe the existing ordering technologies which use logical clock (i.e., using sequence number) and global clock in the following section.

2.5.2.1 Lamport's Clock

In 1978, Leslie Lamport proposed the concept of logical clocks to define the order between events in distributed systems [38]. Lamport assumes that sending or receiving a message is an event in a process. Lamport defines the *happen-before* relation (causality relation) "→" over events such that

(1) If A and B are events in the same process and A occurred before B, then $A \rightarrow B$.

(2) If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.

(3) If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

Two distance events A and B are said to be *concurrent* if $\neg(A \rightarrow B)$ and $\neg(B \rightarrow A)$. Lamport's clocks capture the order between causally related events but they do not detect

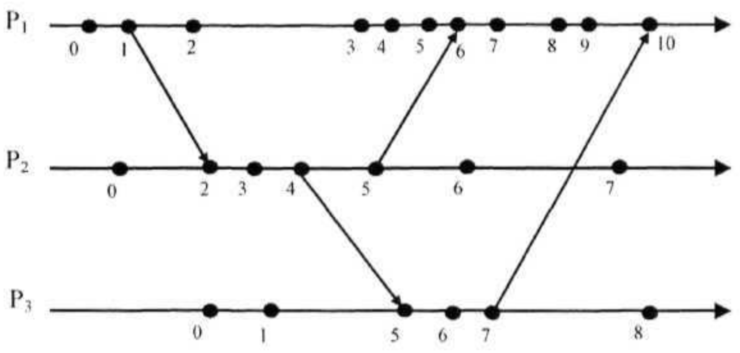


Figure 2.7: Example of Lamport's clock

concurrency between events and it is not possible to decide if the associated events are causally related as shown in Figure 2.7.

Each process P_i keeps its own logical clock C_i, to be a function which assign a number $C_i(A)$ to any event *A* in that process. If event *A* happened before event *B* then the clock value associated with *A* is less than the clock value associated with *B.* But the converse is not always true because any two concurrent events must occur at the same time. For example,

(1) If $A \rightarrow B$, then $C(A) < C(B)$

(2) For any two events *A* and *B* in a process P_1, if *A* occurs before *B* then $C_1(A) < C_1(B)$

(3) If *A* is the sending of a message from P_1 and *B* is the receipt of that message at P_2, then $C_1(A) < C_2(B)$

2.5.2.2 Vector Clock

In 1988, F. Mattern proposed the use of *vector clocks* [48] for finding causality among events. The algorithm uses a *vector clock* V_i of size n where n is the total number of processes. A message has to carry a vector time-stamp which is the value of the sending process's vector clock at the time the message is sent (sending event). Each process P_i has a simple local clock implemented by a counter. which is incremented by 1 each time an event happens. For each event, a process P, ticks by incrementing its own component of the local clock Vj. Process P_i updates V_i according to the following rules:

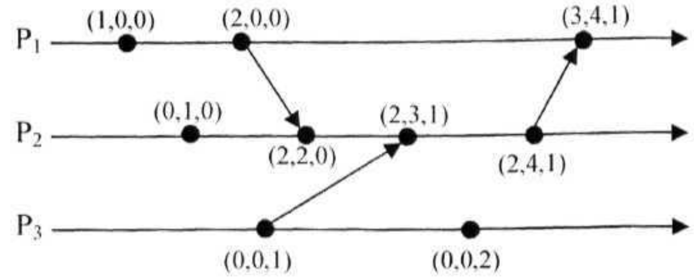


Figure 2.8: Example of vector clock

(1) When an event generated: V_i [i] = V_i [i] + 1

(2) When a message with timestamp W is received from any process:

(a) V_i [k] = max (V_i [k], W[k]) $1 \leq k \leq n$

where n = total number of processes and

W[k] = the received vector clock from another process.

(b) V_i [ij = V_i [i] + 1

In vector clocks, if n becomes large, there are growing storage costs. The example *oi* using vector clock among three processes is shown in Figure 2.8.

2.5.2.3 Matrix Clock

The first algorithm using *matrix clock* was proposed in [67]. This algorithm appends n^2 integers to every message where n is total number of processes. Vector clocks tell what P_j knows about P_k, but they do not reflect what P_i knows about what P_j knows about P_k. Matrix clocks extend vector clocks to capture "what A knows about what B knows about C". Each process P_i maintains a matrix clock M_i (n, n). M_i[j, k] represents that Pi knows about the number of messages P_j has sent to P_k. For example when process P| sends an event *E* to process P_2, P_1 appends its current matrix clock M_1 to *E*. Then P_1 increments its matrix clock M_1 [1, 2] by 1. On receiving the event *E* P_2 checks whether all the events sent to P_2, which are causally dependent on *E* have been delivered. Then P_2 updates its matrix clock M_2 to the pair-wise maximum of M_1 and M_2:

$$M_2\ [i, j] = \max\ (M_2\ [i, j], M_1\ [i,j]),\ \text{for all } i, j.$$

2.5.2.4 Global Clock

Every physical clock has *clock skew* which is the difference between the readings of two clocks and *clock drift* which is the difference between a clock and a nominal perfect reference clock. Hence a *global clock* is not possible without synchronizing local clocks of components. Many clock synchronization protocols have been described in the literature. In 1994, [82] proposed a global time approximation to define fundamental limits of time and order in distributed systems. It assumes that the maximum time difference between any two physical clocks at the same instant of lime is bounded by δ. The *granularity condition* states that the granularity of the global time-base *g* should not be smaller than 8, *g* > 8, ensuring that global clocks do not overlap. A global and total order of events can be determined if event timestamps are two or more clock ticks apart, known as *2g-precedence*.

NTP (Network Time Protocol) and DTSS (Digital Time Synchronization) provide accuracies typically within a millisecond on LANs and a few tens of milliseconds on WANs relative to UTC (Coordinated Universal time) via a GPS (Global Positioning System) receiver [21]. UTC is based on the Earth's rotation about its axis. In 1999, a mechanism based on NTP synchronized local clocks with global reference time is presented in [40] for compositions of events in large-scale system. NTP is an Internet standard protocol that enables local clocks of event servers to maintain global reference time by synchronization of GPS time servers. The algorithm uses accuracy intervals with reliable error bounds for time-stamping of events reflecting the inherent inaccuracy in time measurements. It uses a *window mechanism* to deal with varying transmission delays when composing events from different event sources. It also argues that this approach fits well into mobile environments, provided that the mobile devices are equipped with GPS receivers.

2.5.3 Security

In [50], it introduced a method to specify access control policies which define access rules for large-scale event notification service systems. It also describes threats related to lack of access control such as malicious producers/consumers. If there is no access control mechanism, all consumers can subscribe to all events. In [50], it uses access control filters (i.e., subscriptions or advertisement filters) for building groups of notifications and subscriptions to which the policy rules grant access rights. For a producer, it sends an event together with its credentials to its event server and the event server checks whether publication complies with the policy rules. Similarly a consumer sends a subscription with credentials to its event server. Access rights are based on upper bound or lower bound subscriptions filters. If a consumer is allowed to subscribe with policy rules which have upper bound filter U_c, the consumer can subscribe subscriptions *S* covered by U_c.

For mutual trust between producers and consumers, [28] proposed groups of trust to model and implement security constraints for both the application and system level. A group consists of producers and consumers who are interested in that group. The proposed approach delimits groups on application level to control the visibility of notifications outside of application components and orthogonal to their subscriptions. The original routing table is divided into two kinds of tables. One routing table is to connect subset of event servers for each group. Another is group routing table which allows communication among groups. Producers and consumers are allowed to advertise/subscribe by checking attribute certificate which is a credential with digitally signed identity and a set of attributes.

2.6 Summary

Event notification service (ENS) system is not only emerging as a key technology for message dissemination but also for development of distributed applications. The architecture and the features of an event notification service system strongly depend on its application domain. In this chapter, we begin overview and basic terminologies of event notification service system. Then routing strategies and subscription styles of event notification service systems are introduced. On realizing the uses of ENS systems and the need for better performance, the research issues are identified for further investigation. Other than describing the implementation of event notification service systems, comparisons of these systems are drawn in Table 2.1. The techniques and research issues of event notification service system introduced in this chapter will be used as guideline for designing and developing algorithms in following chapters.

In present days at the advent of new communication technologies, fast information has become feasible and the society has become information savvy. In that context in global scale, there is a requirement of huge infrastructure that not only can collect all possible information from any corner of globe but also can pump information out to consumers at just-in-time for their uses. In addition, a study on information representation has become important so that people from different cultures and background can avail seamless service of event notification service systems across the globe.

Chapter 3

The Architectures of Event Notification Service System in Mobile Computing Environment

This chapter identifies requirements for event notification systems and concentrates on the aspect of how to conceptually model an event notification service system for mobile computing environment. Designs of event notification service system are different on ***wireless network with base stations*** *and* ***ad-hoc network*** *(without base station) as the characteristics of these environments are different. A conceptual architecture of wireless event notification service system is presented on the* ***wireless network with base stations*** *which is widely used in practice (e.g., GSM cellular network).*

3.1 Introduction

In this chapter, we present an overview of an architecture of event notification service (ENS) system for mobile computing environment. In the process, we discuss what functionalities event notification service system should have and how these can be realized by different architectural components of event notification service system. The architecture **has a** significant impact on functionality and scalability of a system. For example, with a centralized event notification service, it is relatively easy to implement complex filtering of notifications, but it is evidently difficult to obtain a scalable service. The main problems of the hierarchical topology are the overloading of higher-level servers. A failure in one server disconnects **all** the subnets reachable from its parent server and all the client subnets from each other.

The routing of notifications is controlled by subscriptions/unsubscriptions and the routing algorithms must take care of disseminating notifications too. Different algorithms adopt different strategies and require more or less complex data structures and computations on event servers. Event notification service system has been recognized as a suitable application catering services to mobile users [18, 49]. But, movement of clients causes issues such as losses and duplications of events and re-routing of events to new locations of clients. We realize the importance of architecture and denote this chapter for its study.

In the following, the related work is described in Section 3.2. The requirements for event notification systems are identified in Section 3.3. Section 3.4 presents possible designs of event notification service systems for different mobile/wireless networks and a basic conceptual architecture for mobile network is presented in Section 3.5. The services for some of these requirements are designed in the next chapters by extending this basic architecture of event notification service system. Finally, we give a summary of the chapter.

3.2 Related Work

Several event notification service systems for the mobility of clients have been proposed in SIENA [10, 11], JEDI [18], Elvin [79] and REBECA [27, 87] as described in Chapter 2. Generally, in these ENS systems, during the time of disconnection notified events are stored at the event server at which the user registered last. On reconnection, the stored events are forwarded to the user's new location. Mobile Push [64] also proposes a similar approach for content dissemination service supporting mobile users. Huang and Garcia-Molina [34, 35] emphasized the problem of supporting mobility in event notification service systems by describing issues that make an event notification service system adaptable to dynamic changes in mobile and ad-hoc networks. Cugola [19] gave preliminary analysis of some issues in designing event notification service by identifying the requirements of underlying wireless communication infrastructure.

A technique for achieving timeliness and reliability for real-time event-based communication in ad-hoc wireless networks has been proposed by Hughes and Cahill [36]. Their conceptual model is the first to directly address the issue of achieving timeliness and reliability in dynamic networks.

3.3 Requirements of Event Notification Service Systems

This section describes the requirements of event notification service systems and it consists of two parts: Typical requirements for every event notification service and additional requirements which depend on different application domains and different environments.

3.3.1 Typical Requirements

Typical requirements of an event notification service system for any applications and environments are as follows:

(1) Delivery of an event to related consumers only: Notifications should be delivered to consumers that match one of their subscriptions.

(2) Delivery of an event exactly once to a consumer: Each notification should be delivered once to a consumer to avoid duplication of notifications.

(3) Delivery of events without losses: All notifications matching one subscription of a consumer should be delivered to the consumer. A consumer should receive his/her interested event if the event was published.

(4) Delivery of events in the same order: Notifications should be delivered in the same order with respect to their publications. Out-of-order event streams can be a problem if the order of events is significant, for example to establish a trend in the movements of a stock's price [34]. If event A was generated before event B, A must be delivered (to all consumers) before B. Some ENS systems may place strict ordering only for events published/generated by a producer/source. But some can tolerate disorder of events generated by different sources.

(5) Fault-tolerant mechanism: Isolated network or component failures (e.g., failures of event servers and link between them) should not affect the entire system. It is necessary to minimize the impact of faults occurring in an event notification service system.

(6) Scalability: A scalable distributed system is one that can easily cope with the addition of users and sites, and whose growth involves minimal expense,

performance degradation, and administrative complexity [51]. Scalability is a crucial requirement for Internet-scale distributed applications.

3.3.2 Additional Requirements

Different event notification service systems may have the following requirements depending on application domains and environments.

(1) Timely delivery of events: Many applications such as stock trading system, auction system, reservation system, sports reporting and traffic condition monitoring systems need to manage deadline bound data items. A user may be willing to receive sports and news information with delay by a few minutes. But it may not tolerate delays in quotes of stock prices.

(2) Support for mobility and disconnection of clients: The ENS system must have mobility support for applications of mobile users in both fixed and wireless networks.

(3) Expressiveness (specifying events and subscriptions): Some applications such as stock trade and auction system must provide powerful data model to specify events and subscriptions. For applications using location-dependent information, the service needs to specify location-related subscriptions and events.

(4) Security requirements: The levels of security requirement for different applications differ to each other. For example, security requirements of e-business and e-commerce applications are higher than weather/traffic report.

3.4 Mobility in Event Notification Service System

Mobility poses new challenges that have not previously been addressed. Client's movement introduces some problems. For example, clients may loose some events while they are disconnected. Mobility entails the study of systems in which components change location, in a voluntary or involuntary manner, and move across a space that may be defined to be either physical or logical.

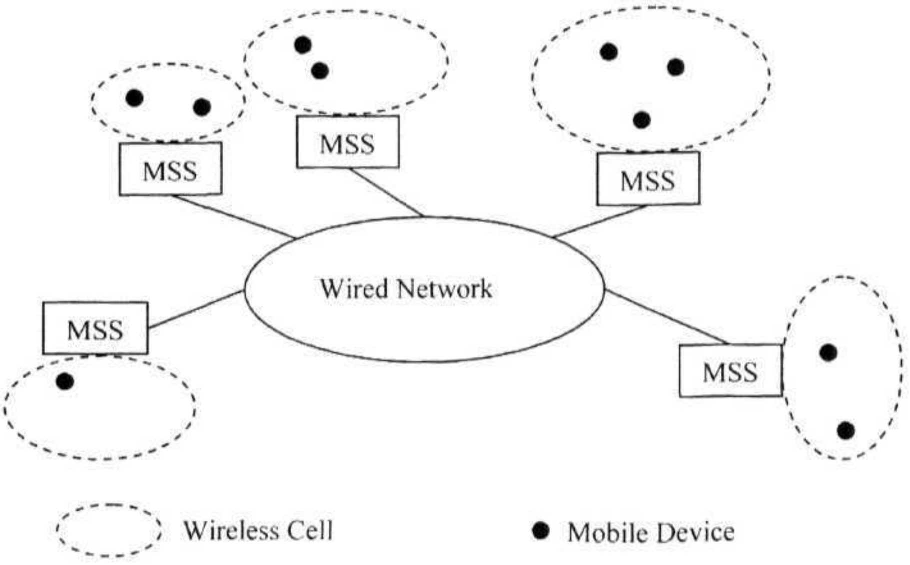


Figure 3.1: A mobile computing system

Host/physical mobility is similar to what in the area of mobile computing is called terminal mobility or roaming. It allows a terminal to be identified by a unique terminal identifier independent of its point of attachment to the network. Hosts move while the virtual names remain fixed. For example, mobile phones are moving while the phone numbers remain the same; a web server is accessible through the same domain name. This kind of mobility is managed at the network layer and, therefore the movement of the host is completely transparent at the application layer. This chapter focuses primarily on host mobility that mobile users move with their mobile devices.

Code/logical mobility involves the code and data movement among hosts. It can redefine bindings between the software components and the network hosts when they move from a node to another in network (for example, mobile agent). The ability to relocate code is a powerful concept that is a wide-spread interest.

3.4.1 Mobile Computing Environment

The design of distributed algorithms and protocols has traditionally been based on an underlying network architecture consisting of static hosts, i.e., the location of a host within the network does not change [5]. In traditional distributed computing environment, different forms of mobility exist. RPC (Remote Producer Call) and RMI (Remote Method

Invocation) mechanisms arc forms of control mobility where a thread of control moves from one (static) machine to another.

Mobile devices consist of portable computing devices, such as notebook computers and handheld devices. A mobile device being capable of wireless networking allows its application components to interact with components hosted by other mobile devices through wireless communication while moving in a mobile computing environment. Because of fundamental limitations of power and form factor, handheld mobile devices have low computing power (CPU speed and memory), restricted power consumption, short battery life, small displays and different limited input devices (e.g., a phone keypad, voice input, etc.). Similarly, wireless data networks present a more constrained communication environment compared to wired networks. Because of fundamental limitations of power, available frequency spectrum, and mobility, wireless data networks tend to have low bandwidth, more latency and low connection stability.

Characteristics of mobile computing environment differ from distributed computing environment due to above various limitations of mobile devices and wireless data network. The mobile computing applications should have the capability to respond to the changes of its environment and resource requirements. Mobile computing environments can use either the infrastructure or the ad-hoc network model for wireless communication [16]. Access points may be connected to a fixed network, such as Intranet or Internet, and act as portals allowing the components under their control to connect to the fixed network. Mobile computing requires integration of handheld mobile devices with existing data network. A mobile computing system is described in Figure 3.1. A host that can move while retaining its network connections is a *mobile host* (MH). A mobile host (MH) can connect to the network from different locations at different times. The infrastructure machines that communicate directly with the mobile hosts are called *mobile support stations* (MSS). The geographical area within which a MSS support mobile hosts is called a *cell.* All MHs that have identified themselves with a particular MSS are considered to be *local* to the MSS. Each MSS maintains a list of identities of local MHs.

On receipt *of leave* ( ) from a local MH, it is deleted from the list. MSS adds the MH in the list of local MHs when the MH sends *register* (id_MH). When a MH connects to a new MSS, the *handofj* procedure is executed between the previous MSS and new MSS.

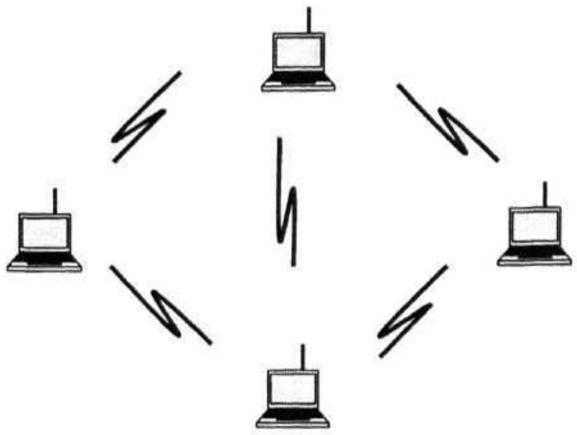


Figure 3.2: Wireless ad-hoc network

When a MH leaves a MSS, it will eventually show up in some MSS. The MH may not always be able to supply the identity of its previous MSS with the *reconnect* ( ) message; in that case, the new MSS may have to query each MSS to determine the previous location of the MH.

Ad-hoc networks are self-organizing wireless networks (Figure 3.2) composed of mobile nodes and requiring no fixed infrastructure [57]. It allows application components to communicate with each other without the aid of access points or a fixed network. Rather, a peer collection of stations within range of listening to each other may dynamically configure themselves into a temporary wireless network [75]. For example, mobile devices can communicate to one another when one comes into range of another as it happens in Bluetooth technology. Ad-hoc networks are extremely useful in scenarios where a natural disaster has wiped out the infrastructure or where rapid deployment is required and an infrastructure is not possible (for example, in the battlefield). Ad-hoc networks can also play a role in civilian forums such as electronic classroom, convention centers, and construction sites [57].

3.4.2 Event Notification Service Systems and Mobile Computing Environment

A good event notification service system has to deal gracefully with both the producer's and consumer's offline [34]. After a user is disconnected or out of reach, event notification

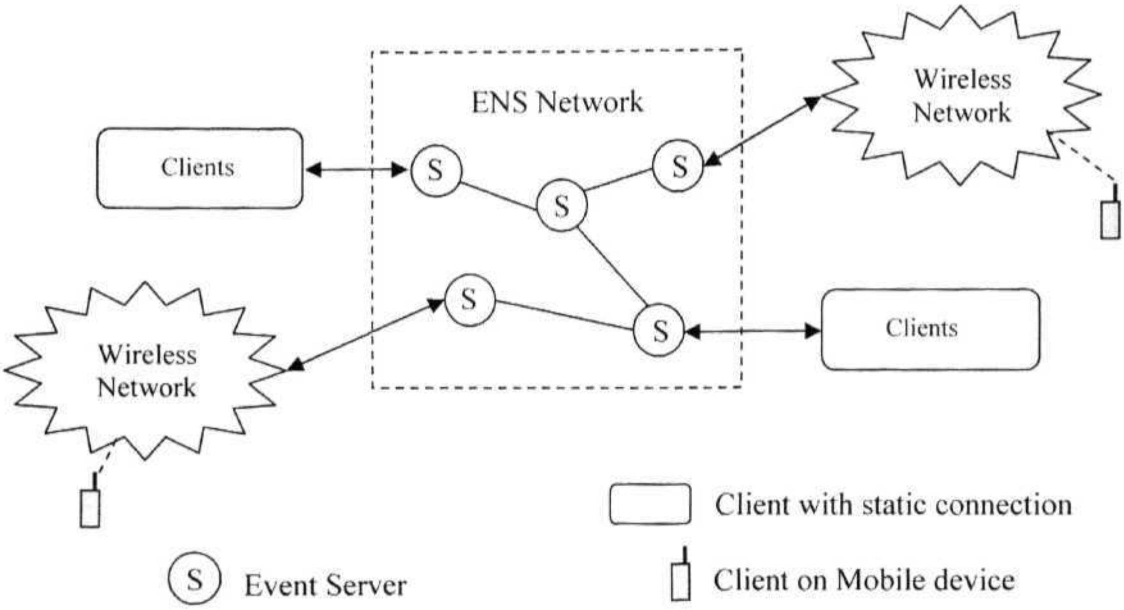


Figure 3.3: Mobile clients on existing ENS system

service system needs to queue the user's events so that they can be delivered later when the user comes back online. In an event notification service system for mobile computing environment, producers and consumers can reside on mobile devices. Event servers may reside on fixed host as they require a fair amount of computing resources. For example, filtering of events may need to check the events against a large amount of subscriptions stored in routing table. But in event notification service for ad-hoc network, event servers must be installed on mobile devices. Hence applications which require fast and large computing power cannot be used in ad-hoc network. The architectures of event notification service systems for mobile environment can be designed as follows.

(1) Wireless network with base stations (e.g., cellular network): When a mobile consumer moves to a new location served by a new event server, subscriptions of the mobile consumer need to be re-subscribed for new paths of notifications. In this case, a consumer can carry its subscriptions and re-subscribes them to new event server. It causes an advantage that the consumer can receive new events even if the old event server is temporarily down. But it needs storage overhead of subscriptions at mobile users and usage of precious bandwidth required for re-subscribing these subscriptions. Hence the event server of previous location

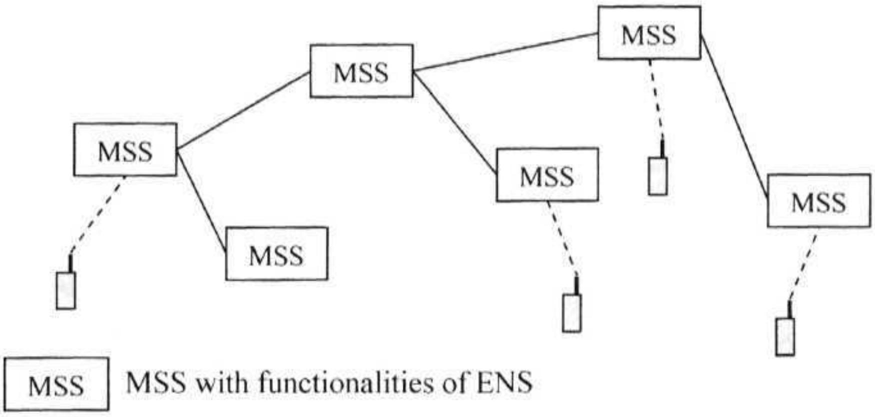


Figure 3.4: Mobile clients on ENS system for wireless network

transfers subscriptions of mobile consumer to the event server of new location and the new event server re-subscribe these subscriptions on behalf of mobile consumer. The event notification service systems for ***wireless network with base station*** can be implemented as follows.

(a) Extending on an existing ENS system as described in Figure 3.3. This architecture is also used in [10]. As base stations of wireless network do not have functionalities of ENS, some base stations are necessary to connect to some event servers of existing ENS system. Hence, event servers need to be extended their functions for mobile users. These extended functionalities consist of storing subscriptions and notifications of mobile users and transferring these subscriptions and notifications to new event servers of mobile users.

(b) Adapting functionality of ENS system to the mobile environment (in Figure 3.4). This architecture can operate itself without the support of fixed ENS system as mobile support stations (MSS) have functionalities of event servers. In many mobile applications such as information on traffic jams, traffic maps and free parking spaces, consumers who are interested events are mobile users. If producers publish these events to wireless network, subscriptions of mobile users are not necessary to forward to event servers of fixed network of ENS system. Hence, this architecture of ENS system should be designed for applications which provide services to mobile users. The details of components and their functions are described in Section 3.5.

(2) **Ad-hoc network** (e.g., Bluetooth): The time-varying capacity of wireless links, limited resources and node mobility make maintaining accurate routing information very difficult in ad-hoc wireless networks [36]. Routing protocols traditionally rely on flooding of queries to discover a destination and they are based on a source initiated query/reply process. In event notification service, the routing of an event is only based *on* subscriptions and the event is delivered to all consumers who issued matched subscriptions. In [19], the designs of event notification service systems for ad-hoc network are described as follows.

(a) Event servers are installed on a set of MHs. Consumers must store and carry their subscriptions and periodically refresh their subscriptions to connected event servers. If the event server is new, the consumers re-subscribe automatically their subscriptions. Otherwise, the old event server could go out of reach and a new event server takes over without knowing about the consumer's subscriptions. It needs to enable dynamic reconfiguration of the system whenever an event server is not anymore reachable through wireless communication.

(b) Each MH acts as a producer/consumer and the service does not rely on event server topology. Event notifications may be filtered at both the producer and the consumer side. All hosts need to cooperate to deliver events from producers to interested consumers. Hence alternative dissemination techniques of events are required in case of lack of event servers.

3.5 Architecture of Wireless Event Notification Service System

This section focuses on event notification service system in ***wireless network with base station*** described in Section 3.4.2. Figure 3.5 shows a basic conceptual architecture of an event server of event notification service system for mobile computing environment. Each event sever consists of several components that are linked together to provide high performance event notification service. In the following sections, the details of these components are described.

Event servers are interconnected with each other to establish distributed event notification service system as shown in Figure 3.4. For simplicity, it is assumed that every MSS is installed with functionalities of event servers although in practice, only some MSSs can be used as event servers. It follows *subscription forwarding* routing strategy as described in Chapter 2.

3.5.1 Subscription, Notification and Routing Table

Each event server maintains a routing table (RT) for propagation of subscriptions and notifications. Routing table (RT) is a collection of tuples (<subscription>, <consumers>, <incoming servers>, <outgoing servers>}; i.e. RT := { < S, C_ID, IS, OS> }

In RT, the consumers associated to a subscription S mean the local consumers who issue subscription S (i.e., RT.S.C_ID).Incoming servers of subscription S (i.e., *RT.S.JS)* are the event servers from which subscription S is sent, and outgoing servers of subscription S (i.e., RT.S.OS) are the event servers to which S is forwarded to.

Registration of subscriptions by each consumer at every event server leads to explosion of information as well as traffic. This problem is managed by using subscriptions *covering method* as described in Section 2.4.1. In covering method, when an event server receives a new subscription S_{new}, it checks whether S_{new} is covered by previous forwarded subscriptions S_{old} (i.e., $S_{\text{new}} \subseteq S_{\text{old}}$) from its routing tale. The detail of covering method can be found in [13, 14].

As and when a mobile user ceases interest to an event, it can unsubscribe to stop receiving the event. For unsubscribing, the relevant subscription is not only removed from the routing table of local event server, but also removed from the outgoing servers. For example on receiving unsubscription S_{u} from a mobile consumer, an event server deletes S_{u} and the mobile consumer's subscriptions covered by S_{u} from routing table. Then S_{u} is forwarded to some neighboring event severs (i.e., RT.S_{u}.OS) for unsubscribing.

If a notification N (an event) is received from a producer, an event server executes *filtering* (matching) process. In filtering process, the event server matches (filters) N with

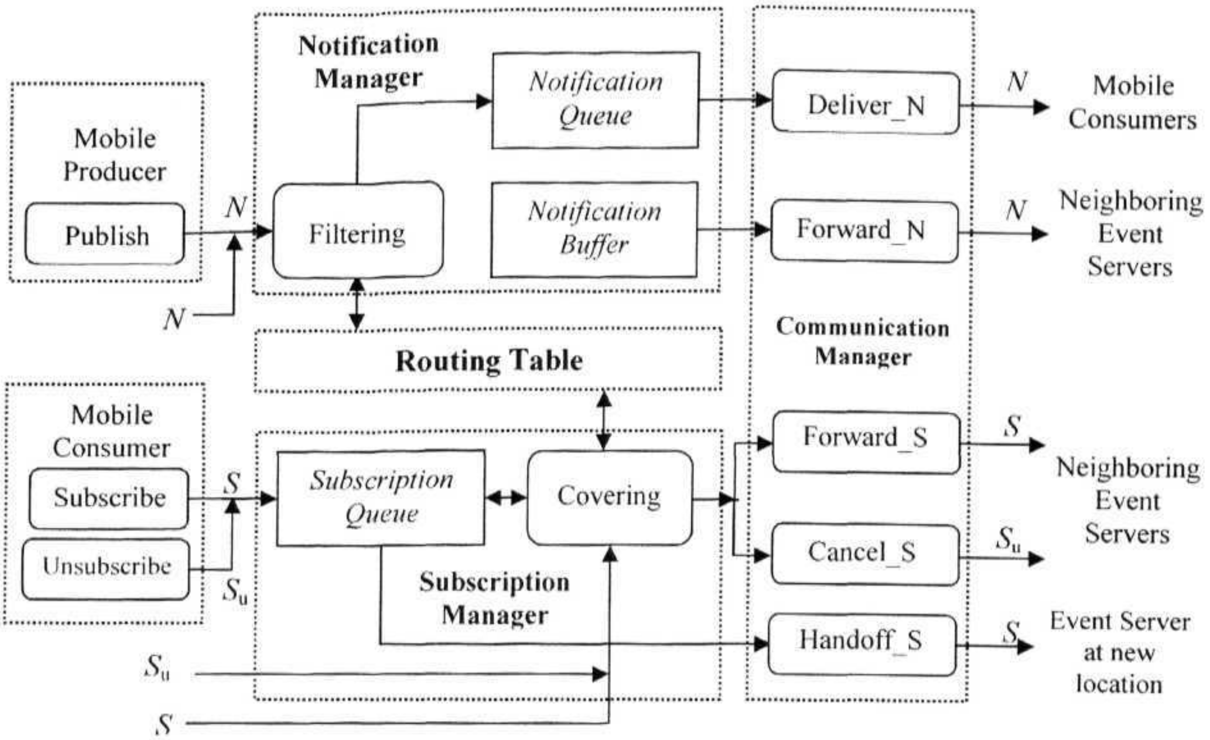


Figure 3.5: The architecture of ENS at event server

all subscriptions stored in RT. A notification *N* matches a subscription *S* if *S* covers *N* ($N \subseteq S$). If RT has a subscription *S* matched with *N* (i.e. $\exists$ *S* in RT | $N \subseteq S$), the results of filtering process are:

(1) a set of local consumers *(LC)* residing in the area covered by local event server (i.e., $LC \in$ RT.*S.C_ID*)and

(2) a set of neighboring event servers (R_{ES}) to which *N* needs to be forwarded (i.e., $R_{ES} \in$ RT.*S.IS*).

Then, the event server delivers *N* to *LC* and forwards *N* to R_{ES}.

3.5.2 Subscription Manager

Subscription manager consists of two parts: *Subscription Queue* of mobile consumers and Covering process. In first case, when subscription manager receives a subscription *S* from a local mobile consumer, it stores *S* in *Subscription Queue* with identity of MH (i.e., <id_MH> <*S*>) and then *S* is sent to Covering process. In Covering, *S* is stored in routing table (RT) and covering method is executed to forward *S* to neighboring event servers. The detail of covering method is described in Section 3.5.1. During handoff of the mobile consumer, its subscriptions stored in *Subscription Queue* are transferred to its new location. After handoff process, MH and its subscriptions are deleted from the list of subscription queue. In case of receiving a subscription *S* from neighboring event servers, subscriptions received from neighboring event servers are not required for handoff. Hence, it is not necessary to store *S* in subscription queue and the covering process is executed directly. In both cases, subscription manager sends the subscription *S* to communication manager. The functionality of communication manager is described in Section 3.5.4.

Subscription manager can receive an unsubscription S_u from a local mobile consumer or a neighboring event server. On receiving unsubscription S_u from a mobile consumer, S_u and the mobile consumer's subscriptions covered by S_u are deleted from *Subscription Queue* and is sent to covering process. Covering process also deletes S_u and the mobile consumer's subscriptions covered by S_u from routing table. Then it sends S_u to communication manager to forward it to neighboring event severs. In case of receiving S_u from neighboring event servers, if S_u is not related with *Subscription Queue*, then it is deleted from routing table.

3.5.3 Notification Manager

The notification manager can receive events/notifications from producers and neighboring event servers. On receiving a notification *N*, notification manager executes Filtering process to match *N* with all subscriptions in routing table (RT). The detail of filtering is described in Section 3.5.1. Filtering process stores matched notifications that are to be forwarded to neighboring event servers, in *Notification Buffer. Notification Buffer* stores these notifications temporarily until it receives acknowledgements of receipt of notices from neighbors. Notification manager uses *Notification Queue* to store matched notifications of local mobile consumers with the identities of the mobile consumers (i.e., <id_MH> <*N*>*)*. Notification manager sends matched notifications to communication manager to deliver/forward them to interested consumers/event servers. If a mobile consumer is disconnected or unreachable, *Notification Queue* stores notifications for a certain period to deliver to the consumer later. When the mobile consumer reconnects to the current event server, notifications of the mobile consumer are delivered. If a mobile consumer connects to a new event server, the related notifications are transferred to the new event server. Notifications which are stored for a mobile consumer MH_s in *Notification Queue* are deleted in the following three cases:

(1) The notifications have been delivered to the MH_s. (i.e., communication manager have executed "Deliver_N" process.)

(2) The notifications have been transferred to a new event server of MH during handoff.

(3) The deadline (expiry time) of the notifications elapsed. Even if storage of notifications at an event server is not a concern, the sheer amount of time and precious bandwidth required to transmit all of the queued notifications to the consumer when it reconnects might be unreasonable. Moreover, many time-sensitive events may become useless when the consumer reconnects to the service (For example, events of whether/traffic information and stock price).

3.5.4 Communication Manager

Communication manager receives subscriptions/unsubscriptions from subscription manager and notifications from notification manager. Communication manager executes "Forward_N" to forward a notification *N* to neighboring event servers. It executes "Deliver_N" to deliver a notification *N* to a local mobile consumer. On receiving subscription *S* "Forward_S" is executed to forward *S* to neighboring event servers. If communication manager receives unsubscription S_u, it executes "Cancel_S" to inform neighboring event servers to cancel subscription *S* covered by S_u (i.e., to unsubscribe *S).*

3.6 Summary

This chapter concentrates on the aspect of how to conceptually model an event notification service system for mobile computing environment. First the typical requirements in designing of event notification service systems arc identified. Additional requirements which are dependent on different application domains and environments are also described. Then the mobile computing environment and its limitation, and possible designs of event notification service system arc presented. Designs of event notification service system are different on wireless network (with base station) and ad-hoc network (without base station) as the characteristics of these environments are different. After that, a conceptual architecture of wireless event notification service system is presented and it focuses on the *wireless network with base stations.* As wireless network with base stations is widely used in practice (e.g., GSM cellular network), it is selected to design event notification service on it. This basic architecture will be used in the following chapters to study the research problems dealt in this thesis.

Chapter 4

Causal Ordering in Event Notification Service Systems for Mobile Users[2]

A distributed event notification service system is an asynchronous system consisting of several subsystems, namely, producer clients, consumer clients and servers without common memory and global clock. The subsystems communicate each other by sending messages with unpredictable transmission delays causing unordered propagation of messages. Moreover, an event notification service system in a mobile environment causes some issues such as losses and duplications of events during handoff between event servers. In this chapter, we present a causal ordering algorithm of event notification service systems for mobile environment.

4.1 Introduction

In this chapter, a causal ordering algorithm for event notification service systems [42] is described. And the algorithm can be used for ordering of messages (subscriptions and notifications) so that mobile users can receive them in the order they are generated. A distributed event notification service system is an asynchronous system consisting of several subsystems, namely, producer clients, consumer clients and servers without common memory and global clock. The subsystems communicate each other by sending messages with unpredictable transmission delays causing unordered propagation of messages. In event notification service systems, it is not desirable for a consumer to receive notifications out of order. For example, the event notification service system should deliver the values of stocks related to a company in correct sequences to an interested consumer. Moreover, an event notification service system in a mobile environment causes some issues such as losses and duplications of events during handoff

* This work is presented and published in *Proceedings of IEEE International Conference on Information Technology, Coding, and Computing (ITCC 04)*, Las Vegas, NV, DSA, pages 735-740, April 2004 [42].

between event servers. Several implementations of event notification service systems [11, 18, 79] provide mobility support for delivering of events. However, none of these implementations consider the ordering of the events. Any of the existing causal ordering algorithms can not be used in event notification service systems due to decoupling characteristics of event notification service as mentioned in Chapter 1. In Section 4.2, after surveying the related works we discuss the reason why causal ordering problem is to be studied for event notification service systems for mobile users.

This chapter is organized as follows: First, related works of causal ordering algorithms for mobile computing environments are described in Section 4.2. After that, the system model, definitions and notations are presented in Section 4.3. A causal ordering algorithm that consists of static module and handoff module is presented in Section 4.4. Correctness proof and analysis of the algorithm is discussed in Section 4.5 and 4.6 respectively. Section 4.7 evaluates the performance of causal ordering algorithm.

4.2 Related Work

Alagar and Venkatesan (AV) [2] proposed three causal ordering algorithms for mobile environment. These are based on the algorithm (RST) proposed in [67]. In RST algorithm, a Process Pj maintains an *n* x *n* integer matrix *SENT* to count the number of messages sent fry every process to all other processes, and an integer vector of size *n* to count the number of messages received by P_i from every other process. Each process appends *SENT* matrix to every message it sends. Each destination process uses the *SENT* matrix received with a message *m* to determine if *m* is deliverable. The complexity of the first algorithm AV-1 [2] is a function of n_h, the number of mobile hosts; hence it is not scaleable. The other two algorithms AV-2 and AV-3 overcome this problem by maintaining causal ordering among mobile support stations (MSSs) to reduce the message overhead but they have unnecessary delay.

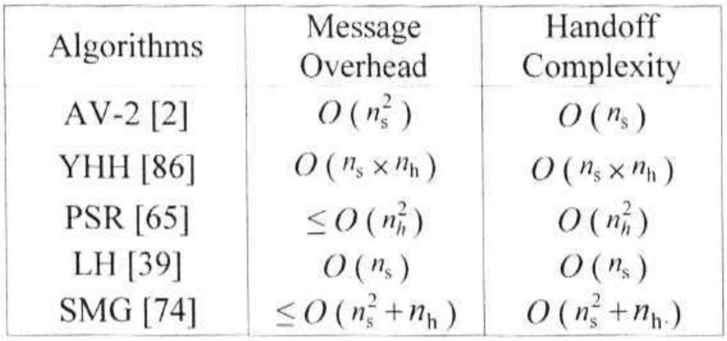
Yen, Huang and Hwang (YHH) [86] have proposed another protocol also based on RST [67]. The scheme proposed by Prakash, Raynal, and Singhal (PSR) [65] is suitable for dynamically changing multicast communication groups. Unlike algorithms in [2, 86], here a message *m* carries direct dependency information to efficiently enforce causal ordering. As it depends on n_h so it is unsuited for dynamic mobile environment.

Li and Huang (LH) [39] presented a scalable causal multicast algorithm for mobile environment whose message overhead is smaller than that of other algorithms. MSS maintains a vector of size n_s, the number of MSSs, to record the number of messages it has received from other MSSs. It also maintains two vectors of length n_s for each of its local mobile hosts to store the sequence number of messages sent by and received from MSSs.

Skawratananond, Mittal and Garg (SMG) [74] proposed an algorithm based on the algorithms of AV-2 [2] and YHH [86] to reduce the unnecessary inhibition (delay). In this algorithm, each MSS maintains an n_s x n_s matrix *SENT* for every local mobile host. An MSS also maintains two vectors of size n_s each to store the last number of messages sent by and received from it. Each MSS sends a message *m* with the matrix *SENT* and its local knowledge of the locations of all the mobile hosts *UPCELL* that have changed their cells since it last communicated with the other MSSs.

The comparisons of the results of the existing causal ordering algorithms for mobile computing environment are shown in Table 4.1. In the above causal ordering algorithms, the mobile support station (MSS) that sends messages knows the destinations of messages. Hence a causal ordering algorithm is only executed at the destination MSS. The algorithms reported in [2, 86, 39, 65, 74] intends to order the events occurring at all the servers and hosts. But in case of event notification service systems it is enough to consider ordering of events transmitted by an event server to its neighbors only lying on a path between a producer and a consumer. In this chapter, we provide a causal ordering algorithm for the ordering problem applicable to ENS system only. In distributed event notification service systems, as the routing of messages depends on their contents, each event server is only aware of its neighboring event servers to which the events need to be forwarded. An event notification service system has two kinds of clients (producer and consumer) and their functionalities are different. The producers publish events and need not receive any messages. But the consumers send subscriptions to the service and receive notifications which match their subscriptions. Moreover, when a consumer moves to the area under a new event server, the new event server needs to establish a route of the events to the consumer.

The ordering of events is more important for event notification service system in a mobile computing environment as frequent disconnections and changing of event servers of clients. We propose a causal ordering algorithm [42] in event notification service systems for mobile users to receive events with correct sequence and to avoid loss and duplication of events. The proposed algorithm consists of two modules: the *static* module for normal operation and *handoff* module for mobile hosts which migrate to new locations. An event server appends a corresponding sequence number to every message (subscription/notification) which it forwards to neighboring event servers. On receiving notifications or subscriptions, an event server executes causal ordering algorithm before executing other processes.



Algorithms	Message Overhead	Handoff Complexity
AV-2 [2]	$O(n_s^2)$	$O(n_s)$
YHH [86]	$O(n_s \times n_h)$	$O(n_s \times n_h)$
PSR [65]	$\leq O(n_h^2)$	$O(n_h^2)$
LH [39]	$O(n_s)$	$O(n_s)$
SMG [74]	$\leq O(n_s^2 + n_h)$	$O(n_s^2 + n_h)$

Table 4.1: A Comparison of causal ordering algorithms for mobile computing system

4.3 System Model and Definitions

4.3.1 System Model

A mobile computing system consists of a set of mobile hosts (MHs) and fixed mobile support stations (MSSs). An MSS communicates with the MHs within its cell via a wireless medium. The MSSs are connected each other with wired channel as a static network. All MHs that arc located within a cell of an MSS are considered to be local to that MSS. We can assume that wireless channel between an MSS and each of its MHs is FIFO as described in [2, 39, 74]. But a logical channel between every pair of MSSs need not be a FIFO channel. Both wired and wireless channels are reliable and take an arbitrary but finite amount of time to deliver events.

The proposed algorithm belongs to the architecture of event notification service system for wireless network with base stations which is described in Section 3.4.2. Usually event servers are implemented on separate servers. However, MSSs with enough computing resources engaged in wireless communication can also be used as event servers. For simplicity, we assume that every MSS in the system is used as an event server although in practice, only some MSSs can be used as event servers. We also assume that all connections are bi-directional. Therefore, the abstract architectural model is an undirected graph. In this algorithm, each event server maintains causal ordering of events and forwards/delivers them to neighboring event servers/local consumers.

4.3.2 Causal Ordering

The causal ordering is defined by Lamport's *happened-before* relationship [38] indicated by → symbol.

For any two events E_1 and E_2, $E_1 \rightarrow E_2$ is true if

(1) E_1 and E_2 occur at the same host, and E_1 occurs before E_2, or

(2) E_1 is a sending of a message, and E_2 is receiving of that message, or

(3) There exists an event E_3 such that $E_1 \rightarrow E_3$ and E3 → E_2.

Subscription Ordering

If a consumer issues two subscriptions *S* and S_2, then *S* and S_2 must be delivered to event servers in the same order they were issued.

As wireless channel is FIFO, it ensures that *Subscribe (S\)* → *Subscribe* (S_2) at MH then *Receive* (S_1) → *Receive* (S_2) at local event server. Hence the causal ordering among event servers must satisfy that

if ***Forward {S\)*** → ***Forward*** (S_2) **then** ***Recv*** *(S\)* → *Recv* (S_2)

Forward *(S\)* means that an event server forwards subscription *S* to neighboring event servers and ***Recv*** *(S\)* means that subscription *S* is received and stored in routing tables of each event server of neighboring event servers.

Notification Ordering

Two notifications *N* and N_2 published by the same producer must be delivered to consumers whose subscriptions are matched with both N_1 and AS, in the same order they were published.

We assumed wireless channel is FIFO and so it ensures that if ***Publish (N\)*** → ***Publish*** (N_2) at producer then *Receive (N\)* → *Receive* (N_2) at local event server. Among the event servers in wired network, the causal ordering must satisfy that

if *Notify* (N_1) → *Notify* (N_2) **then** ***Deliver*** (N_1) → ***Deliver*** (N_2)

Notify (N_1) means that an event server sends notification *N* to relative event servers and ***Deliver*** (N_1) means that an event server delivers notification N_1 to consumers whose are interested in N_1.

4.4 Algorithms for Causal Ordering

We propose a set of algorithms for subscriptions and notifications for mobile users. A mobile user can function as producer as well as consumer. A subscription is initiated by a mobile consumer and the subscription is enlisted in the network of event servers. Similarly a mobile producer publishes an event and it spreads through the network of event servers for matching with subscriptions. Moreover, both subscription and notification algorithms are required to be adaptive to disconnections of producers and consumers. The terms of notifications and messages are used interchangeably in the following sections. For the above functionalities, we have proposed the following algorithms:

Algorithms are partitioned into two categories viz. static module and handoff module. Algorithms of static module perform core activities like subscription forwarding and notification delivery. These algorithms are run at event servers connected over static network. Whereas, algorithms of handoff module manage disconnections of mobile users. In subsequent two subsections we present these algorithms. Before formal presentations of algorithms, the important data structures used in these algorithms are introduced.

Each event server ES_i maintains four vectors each of size n_{ns}, the total number of neighboring event servers as follows:

(1) MSS_SSEQ_i: a vector of sequence number of subscriptions sent by ESj. The j^{th} entry of MSS_SSEQ_i, $MSS\_SSEQ_i[j]$ denotes the number of subscriptions sent by ES_i to ESj.

(2) MSS_NSEQ_i: a vector of sequence number of notifications sent by ESj. The j^{th} entry of MSS_NSEQ_i, $MSS\_NSEQ_i[j]$ denotes the number of notifications sent by ES_i to ESj.

(3) MSS_SRCV_i: a vector of sequence number of subscriptions received by ESj. The j^{th} entry of MSS_SRCV_i, $MSS\_SRCV_i[j]$ denotes the number of subscriptions sent by ESj that have received at ESj.

(4) MSS_NRCV_i: a vector of sequence number of notifications received by ESj. The j^{th} entry of MSS_NRCV_i, $MSS\_NRCV_i[j]$ denotes the number of notifications sent by ESj that have received at ESj. Initially, all entries of each vector are set to zero.

Each event server ESj also maintains two sets of sequence numbers MH_SENT_k and MH_DELIV_k for its local producers and consumers respectively.

(1) MH_SENT_k denotes the last sequence number of notifications published by a producer MH_k that have received at ESj.

(2) MH_DELIV_k denotes the last sequence number of notifications delivered to a consumer MH_k by the ES;.

An event server ESj uses the two queues to store temporarily subscriptions and notifications received from neighboring event servers if they arc not deliverable.

(1) $S\_PENDING_j$: a queue to store subscriptions received from ES_j.

(2) $N\_PENDING_j$: a queue to store notifications received from ES_j.

Each event server ESj uses two set of FIFO queues for storages of subscriptions issued by mobile consumers and notifications for disconnected mobile consumers.

(1) $SSUBQ\_MH_k$ is used to store subscriptions issued by local consumer MH_k.

(2) $DELIVQ\_MH_k$ is used to store notifications which will be delivered to local consumer MH_k.

Each mobile consumer MH_k stores a sequence numbers MH_NRCV_k which is the last sequence number of the latest notification received from its current event server. Each mobile producer MH_k stores a sequence number MH_NSEND_k which is the last number of notifications sent to its current event server.

4.4.1 Static Module

Subscription Forwarding

When a consumer MH_s wants to get information it first sends ***Subscribe*** *(S, MH_s)* to its local ESj. On receiving ***Subscribe*** (S, MH_s), ESj stores *S* in the routing table (RT) and sends an acknowledgment to MH_s (Step Al. (2)). All subscriptions issued by MH_s are also stored in $SSUBQ\_MH_s$ with subscribing order. Then ESj makes *covering test* (Step A3) and the result is *F,* a set of neighboring event servers, to forward *S.* ESj sends ***Forward*** (ES_i, *S,* $MSS\_SSEQ_i[p]$) to p where p **e** *F* with its own identity "ES_i"and then increases $MSS\_SSEQ_i[p]$ by one (Step A3).

On receiving ***Forward*** (ES_i, S, MSS_SSEQi[j]) from ESj, ESj checks whether *S* is deliverable ($MSS\_SSEQ_i[j] \leq MSS\_SRCV_j[i]$) (Step A2). If *S* is not currently deliverable, ESj stores *S* temporarily in $S\_PENDING_i$ until it becomes deliverable. If *S* is deliverable, ESj stores *S* in the routing table (RT) and increases MSS_SRCVj[iJ by one. Then ESj executes the *covering test* (Step A3). Subscription_ forwarding algorithm is described in Figure 4.1.

Notification Delivery

A producer MH_p sends ***Publish*** (N, MH_p, MH_NSEND_p) to its local event server ESj to publish a notification (an event) *N.* On receiving ***Publish*** (N, MH_p, MH_NSEND_p), ESj sends an acknowledgment to MH_p (Step A4. (1)) and executes *event matching* process with subscriptions stored in RT. The result is (1) a set of local consumers *LC* to deliver N (Step A6. (1)) and (2) a set of neighboring event severs *R* to forward TV (Step A6. (2)). ESj delivers *N* to *LC* and sends ***Notify*** (ES_i, N, MSS_NSEQi[q]) to q where *qeR.* Then ESj increases $MSS\_NSEQ_i[q]$ by one.

On receiving a ***Notify*** (ES_i, N, MSS_NSEQ_i), ESj determines that *N* is deliverable ($MSS\_NSEQ_i[j] \leq MSS\_NRCV_j[i]$) (Step A5). If N is not cun-ently deliverable, it is kept in N_PENDINGj. If TV is deliverable, ES_j execute event matching (Step A5. (2)) and then delivers/forwards *N* to concerned consumers/event servers. Notification_delivery algorithm is described in Figure 4.2.

```
A1.   At server ES$_i$ ; MH$_s$ (consumer) residing at ES$_i$.
      On receiving Subscribe ($S$, $MH_s$) do
      begin
         (1)   Store subscription $S$ in RT;   /* Routing Table (RT)*/
         (2)   Send an acknowledgment to MH$_s$;
         (3)   Store $S$ in SSUBQ_MH$_s$;      /* store with subscribing order*/
         (4)   Call Cover ($ES_i$);   /* Step A3. Covering method*/
      end

A2.   At server ES$_j$;
      On receiving a Forwd ($ES_i$, $S$, MSS_SSEQ$_i$[j]) from ES$_i$ do
      begin
         /* $S$ is deliverable if the sequence number (MSS_SSEQ$_i$[j]) sent from ES$_i$ is less than
         or equal to receiving sequence number (MSS_SRCV$_j$[i]) of ES$_j$.*/
         if (MSS_SSEQ$_i$[j] $\leq$ MSS_SRCV$_j$[i])      then
            (1)   $S$ is stored in RT;
            (2)   MSS_SRCV$_j$[i] ++;
            (3)   Call Cover ($ES_i$);   /* Step A3. Covering method and $S$ is sent from $ES_i$ */
            (4)   For all $S \in$ S_PENDING$_i$ do  /* to check subscriptions in S_PENDING$_i$*/
                     begin
                        if (MSS_SSEQ$_i$[j] $\leq$ MSS_SRCV$_j$[i])
                           (a)   $S$ is stored in RT;
                           (b)   MSS_SRCV$_j$[i] ++;
                           (c)   Call Cover ($ES_i$);   /* Step A3 and $S$ is sent from $ES_i$ */
                        endif
                     end
         else
            Append message ($ES_i$, $S$, MSS_SSEQ$_i$[j]) to S_PENDING$_i$; /* store $S$ from ES$_i$ */
         endif
      end

A3.   At each server;
      On calling Cover ($ES_M$) do
      begin
         if ($\exists$ $S_1$  in RT | $S \subseteq S_1$ )  then
            $F = n_{ns} - \mathrm{RT}.\ S_1\ .OS - ES_M$;  /* $n_{ns}$ means all neighboring event servers of ES$_M$*/
         else
            $F = n_{ns} - ES_M$;
         endif
         For all p $\in$ $F$ do
            begin
               Send Forward ($ES_M$, $S$, MSS_SSEQ$_M$[p]) to p;
               MSS_SSEQ$_M$[p]++;
            end
      end
```

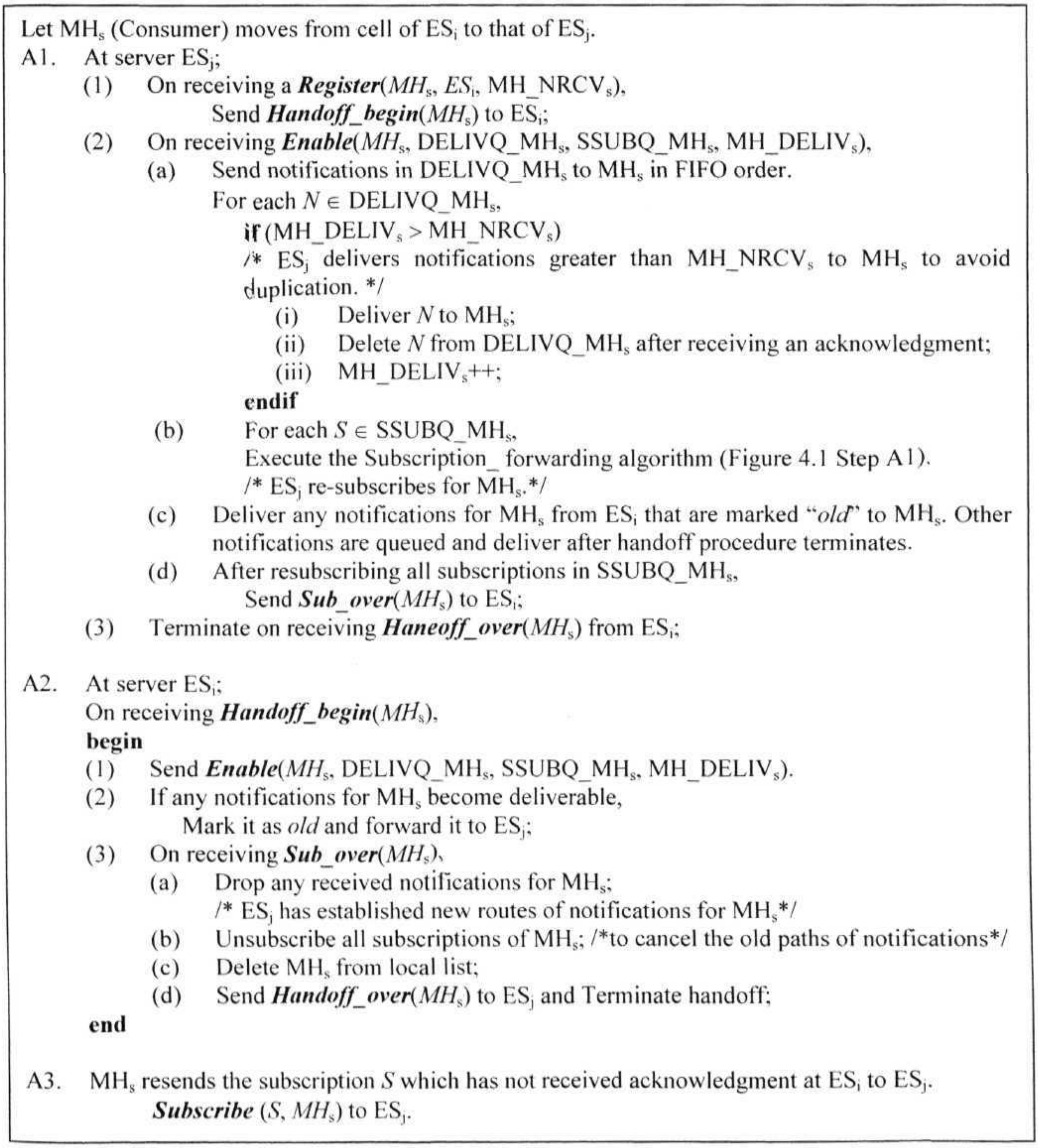
Figure 4.1: Subscription_forwarding algorithm

```
A4.  At server $ES_i$ ; $MH_p$ (producer) residing at $ES_i$
     On receiving a Publish ($N$, $MH_p$, $MH\_NSEND_p$) do
     begin
        if ($MH\_NSEND_p > MH\_SENT_p$ )    then
           begin
              (1)    Send an acknowledgment to $MH_p$;
              (2)    $MH\_SENT_p$ ++;
              (3)    Call Matching ($ES_i$);    /*Step A6. Event Matching*/
           end
        else    Drop $N$;   /* $ES_i$ has already received $N$ */
        endif
     end
A5.  At server $ES_j$;
     On receiving a Notify ($ES_i$, $N$, $MSS\_NSEQ_i[j]$) do
     begin
        if ($MSS\_NSEQ_i[j] \leq MSS\_NRCV_j[i]$)    then
           begin
              (1)    $MSS\_NRCV_j[i]$ ++;
              (2)    Call Matching ($ES_j$);    /*Step A6. Event Matching*/
              (3)    For all $N \in N\_PENDING_i$ do
                     begin
                        if ($MSS\_NSEQ_i[j] \leq MSS\_NRCV_j[i]$)
                           (a)    $MSS\_NRCV_j[i]$ ++;
                           (b)    Call Matching ($ES_j$);    /*Step A6. Event Matching*/
                        endif
                     end
           end
        else
           Append message ($ES_i$, $N$, $MSS\_NSEQ_i[j]$) to  $N\_PENDING_i$; /*store $N$  in Queue. */
        endif
     end
A6.  At each server;
     On calling Matching ($ES_N$) do
     begin
        if ($\exists S_1$   in RT | $N \subseteq S_1$   ) then
           (1)    $LC$ = RT. $S_1$.$C\_ID$;  /* $LC$ is a set of consumers residing at $ES_N$ */
                  For all y $\in LC$ do
                  begin
                     (a)    Deliver $N$ to y and store $N$ in $DELIVQ\_MH_s$;
                     (b)    Delete $N$ from $DELIVQ\_MH_s$ after receiving an acknowledgment;
                     (c)    $MH\_DELIV_s$++;
                  end
           (2)    $R$ = RT. $S_1$ .$IS$;  /*$R$ is a set of servers to which $N$ needs to be forwarded by $ES_N$*/
                  For all q $\in R$ do
                  begin
                     (a)    Send Notify ($ES_N$, $N$, $MSS\_NSEQ_N[q]$) to q;
                     (b)    $MSS\_NSEQ_N[q]$ ++;
                  end
        else      Terminate forwarding of $N$;
        endif
     end
```

Figure 4.2: Notification_delivery algorithm

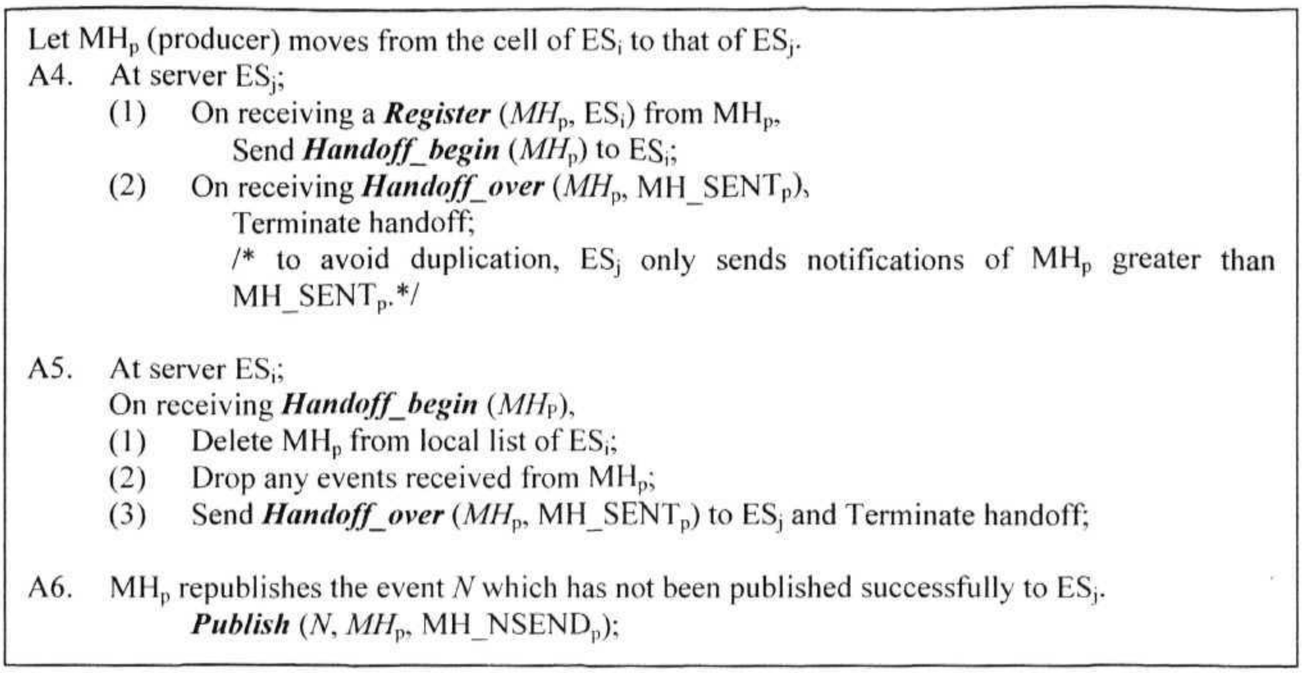
4.4.2 Handoff Module

When a mobile host (MH) moves to a new location, the algorithm ensures that subscriptions and notifications are neither lost nor replicated besides maintaining the correct sequence of subscriptions and notifications.



```
Let $MH_s$ (Consumer) moves from cell of $ES_i$ to that of $ES_j$.
A1.  At server $ES_j$;
     (1)  On receiving a Register($MH_s$, $ES_i$, $MH\_NRCV_s$),
               Send Handoff_begin($MH_s$) to $ES_i$;
     (2)  On receiving Enable($MH_s$, $DELIVQ\_MH_s$, $SSUBQ\_MH_s$, $MH\_DELIV_s$),
          (a)  Send notifications in $DELIVQ\_MH_s$ to $MH_s$ in FIFO order.
               For each $N \in DELIVQ\_MH_s$,
                  if ($MH\_DELIV_s > MH\_NRCV_s$)
                  /* $ES_j$ delivers notifications greater than $MH\_NRCV_s$ to $MH_s$ to avoid
                  duplication. */
                     (i)    Deliver $N$ to $MH_s$;
                     (ii)   Delete $N$ from $DELIVQ\_MH_s$ after receiving an acknowledgment;
                     (iii)  $MH\_DELIV_s$++;
                  endif
          (b)     For each $S \in SSUBQ\_MH_s$,
                  Execute the Subscription_ forwarding algorithm (Figure 4.1 Step A1).
                  /* $ES_j$ re-subscribes for $MH_s$.*/
          (c)  Deliver any notifications for $MH_s$ from $ES_i$ that are marked "old" to $MH_s$. Other
               notifications are queued and deliver after handoff procedure terminates.
          (d)  After resubscribing all subscriptions in $SSUBQ\_MH_s$,
                  Send Sub_over($MH_s$) to $ES_i$;
     (3)  Terminate on receiving Haneoff_over($MH_s$) from $ES_i$;

A2.  At server $ES_i$;
     On receiving Handoff_begin($MH_s$),
     begin
     (1)  Send Enable($MH_s$, $DELIVQ\_MH_s$, $SSUBQ\_MH_s$, $MH\_DELIV_s$).
     (2)  If any notifications for $MH_s$ become deliverable,
             Mark it as old and forward it to $ES_j$;
     (3)  On receiving Sub_over($MH_s$),
          (a)  Drop any received notifications for $MH_s$;
               /* $ES_j$ has established new routes of notifications for $MH_s$*/
          (b)  Unsubscribe all subscriptions of $MH_s$; /*to cancel the old paths of notifications*/
          (c)  Delete $MH_s$ from local list;
          (d)  Send Handoff_over($MH_s$) to $ES_j$ and Terminate handoff;
     end

A3.  $MH_s$ resends the subscription $S$ which has not received acknowledgment at $ES_i$ to $ES_j$.
          Subscribe ($S$, $MH_s$) to $ES_j$.
```

Figure 4.3: Handoff_consumer algorithm



```
Let $MH_p$ (producer) moves from the cell of $ES_i$ to that of $ES_j$.
A4.  At server $ES_j$;
     (1)  On receiving a Register ($MH_p$, $ES_i$) from $MH_p$,
               Send Handoff_begin ($MH_p$) to $ES_i$;
     (2)  On receiving Handoff_over ($MH_p$, $MH\_SENT_p$),
               Terminate handoff;
               /* to avoid duplication, $ES_j$ only sends notifications of $MH_p$ greater than
               $MH\_SENT_p$.*/

A5.  At server $ES_i$;
     On receiving Handoff_begin ($MH_P$),
     (1)  Delete $MH_p$ from local list of $ES_i$;
     (2)  Drop any events received from $MH_p$;
     (3)  Send Handoff_over ($MH_p$, $MH\_SENT_p$) to $ES_j$ and Terminate handoff;

A6.  $MH_p$ republishes the event $N$ which has not been published successfully to $ES_j$.
          Publish ($N$, $MH_p$, $MH\_NSEND_p$);
```

Figure 4.4: Handoff_producer algorithm

Handoff of Consumer

Let a consumer mobile host MH_s moves from cell of ESj to cell of ESj. After entering the cell of ESj, MH_s sends the message ***Register*** (MH_s, ES_i, MH_NRCV_s) to ESj with its own identity *"MH";* and identity *"ES";* of previous event server. On receiving ***Register*** (MH_s, ES_i, MH_NRCV_s), ESj sends ***Handoff_begin*** (MH_s) to ESj (Step Al. (1)). On receiving ***Handoffbegin*** (MH_s), ESj transfers the relevant information of MH_s to ES_j via the message ***Enable*** (MH_s, $DELIVQ\_MH_s$, $SSUBQ\_MH_s$, MH_DELIV_s) (Step A2. (1)).

On receiving ***Enable*** (MH_s, $DELIVQ\_MH_s$, $SSUBQ\_MH_s$, MH_DELIV_s), ESj delivers notifications in $DELIVQ\_MH_s$ with sequence number greater than MH_NRCV_s to MH_s in the FIFO order (Step Al. (2a)). Then ESj re-subscribes all subscriptions in $SSUBQ\_MH_s$ in FIFO order (Step Al. (2b)). After resending all subscriptions, ESj sends ***Sub_over*** (MH_s) to ESj. On receiving *Subover* (MH_s), ESj drops received notifications for MH_s as ESj has established new routes of notifications for MH_s. Then the previous ESj unsubscribe all subscriptions of MH_s to cancel old routes of notifications for MH_s (Step A2. (3b)). ESj deletes MH_s from its local list and send ***Handoffover*** (MH_s) to ESj (Step A2. (3d)). A description of the handoff algorithm for consumers is shown in Figure 4.3.

Handoff of Producer

Let a producer mobile host MH$_p$ move from area of ES$_i$ to area of ESj. After entering the area covered by ESj, MH$_p$ sends the message ***Register*** (MH_p, ES_i) to ESj with its own identity "MH_p" and identity "ES_i" of previous event server ESj. On receiving ***Register*** (MH_p, ES_i), ESj sends ***Handoff_begin*** (MH_p) to ESi (Step A4. (1)). After receiving ***Handoff_begin*** (MH_p), ESj deletes MH$_p$ from its local list and sends message ***Handoffover*** (MH_p, MH_SENT$_p$) to ES, (Step A5. (3)). On receiving ***Handoff_over*** (MH_p, MH_SENT$_p$), ESj terminates handoff procedure and starts executing steps of static module. A description of the handoff algorithm for producers is shown in Figure 4.4.

4.5 Correctness Proof

For ensuring the correctness, we take up strictly on *safety* and *liveness* properties. *Safety* property means the *causal ordering* described in Section 4.3.2 is never violated. *Liveness* property means the algorithm never delays a message indefinitely.

4.5.1 Safety Property Proof

Lemma 4.1. Let *N* be a notification with sequence number MSS_NSEQ$_i$[j] = *k* sent by ES, to ES,.

If *N* has not been forwarded to ESj, then MSS_NRCV$_j$[i] < *k* at ESj.

Proof. Let ESj sends *N* to ESj with sequence number *k* where "MSS_NSEQ$_i$[j] of N_1=k". Assume that *N* has not been delivered to ESj. By contradiction, suppose that there exists a notification N_2 with sequence number "MSS_NSEQ$_i$[j] of $N_2 \geq k$" that can be delivered to

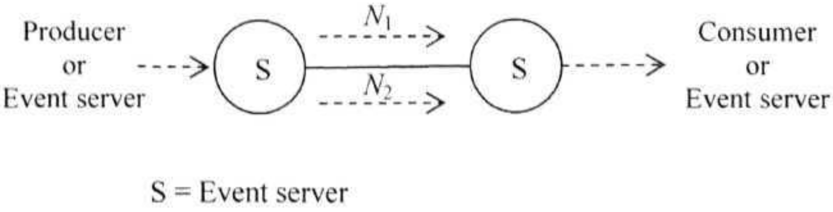


Figure 4.5: Notifications sent to/from same event server

ESj before N_1. It will cause that MSS_NRCV$_j$[i] $\geq$ *k* without delivering *N*\.

$N_1 \rightarrow N_2$ at ES$_i$, hence MSS_NSEQ$_i$[j] of N_2 > *k*. If N_1 has not been delivered to ESj, then MSS_NSEQ$_i$[j] of N_2 n MSS_NRCVj[i] at ESj. This condition is not satisfied the step A5 of Notification_delivery algotithm and, thus N_2 is not deliverable.

Lemma 4.2. Let *S*\ be a subscription with sequence number MSS_SSEQ$_i$[j] = *k* sent by ES$_i$ to ESj.

If S_1 has not been forwarded to ESj, then MSS_SRCV$_j$[i] < *k* at ES$_j$.

(The proof is similar to Lemma 4.1.)

4.5.1.1 Ordering of Notifications

Suppose a notification N_2 is published after *N*\ by a producer and subscriptions *8'* and *S"* of a consumer match with *N*\ and N_2 (i.e. *N*\ $\subset$ *S'* and $N_2 \subset$ *S")* respectively. *Safety* property must satisfy that N_2 can not be delivered if N_1 has not been delivered to the consumer.

Case 1. An event server sends *N*\ and N_2 to the same event server as shown in Figure 4.5.

Suppose ES$_i$ send *N*\ and N_2 to ES$_j$. Hence, at ES$_i$:

MSS_NSEQ$_i$[j] of N_1 < MSS_NSEQ$_i$[j] of tf$_2$ (1)

By Lemma 4.1, if *N*\ has not been delivered to ESj, at ES$_j$:

MSS_NRCV$_j$[i] < MSS_NSEQi[j] of N_1 (2)

From (1) and (2), MSS_NSEQ$_i$[j] *of* N_2 □ MSS_NRCV$_j$[i]. It is not satisfied the step A5 of Notification_delivery algorithm and N_2 is appended to queue N_PENDlNGj at ESj until it is deliverable. Hence, it ensures that N_2 cannot be delivered or forwarded before *N*\.

Case 2. N_1 and N_2 are sent from different event servers to consumers or event servers as shown in Figure 4.6 (i.e., it occurs when a producer moves and connects to a new event server).

Suppose a producer MH$_p$ moves from ESj to ESj.

ES$_i$ increments MH_SENT$_p$ by one after receiving N_1 (Step A4.(2) of static module). MH_SENT$_p$ is transferred to ESj during handoff (Step A5.(3) of handoff module).

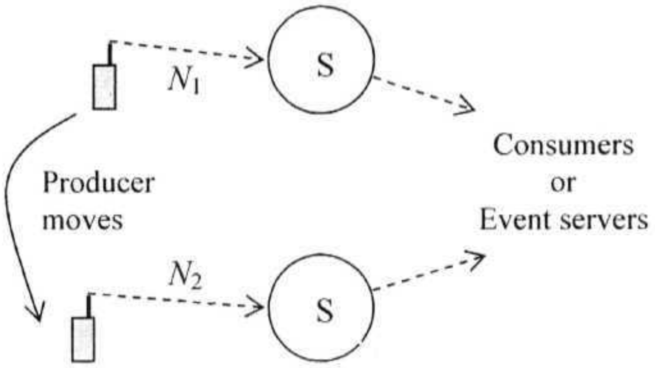


Figure 4.6: Notifications sent from different event servers

If the producer MH_p has not received acknowledgement of $N\backslash$ from ESj, it will republish $N\backslash$ to ESj. ES_j delivers/forwards new notifications (including N_1) published y MH_p after handoff procedure is completed if sequence number of N_1 is greater than MH_SENT_p. Hence it ensures that N_1 is sent before N_2.

Case 3. N_1 and N_2 are sent to different event servers as shown in Figure 4.7 (i.e., consumer who receives N_1 and N_2 moves and connects to a new event server).

Suppose a consumer MH_s moves from ES_i to ESj.

Although ESj receives N_2, it will not deliver N_2 to MH_s till handoff producer is completed. If ESj has not delivered $N\backslash$ to MH_s successfully, ESj forwards N_1 as "*old*" marked notification to ES_j (step A2. (2) of handoff module) or ESj would have transferred $N\backslash$ as contents of $DELIVQ\_MH_s$, FIFO queue (Step A2. (1) of handoff module) during handoff. Hence ESj will deliver N_1 before handoff procedure is terminated and thus N_2 is delivered to the consumer after N_1.

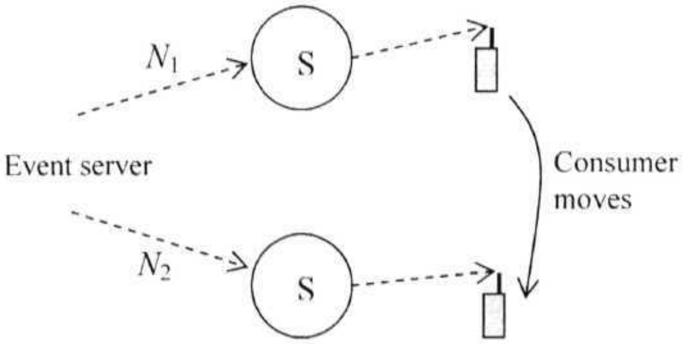


Figure 4.7: Notifications sent to different event servers

4.5.1.2 Ordering of Subscriptions

Suppose a subscription S_2 is subscribed after S\ by a consumer and *safety* property must satisfy that S_2 can not be forwarded if S_1 has not been forwarded to neighboring event servers.

Case 1. An event server forwards subscriptions S\ and S_2 to the same event server.

Suppose ESj forwards S_1 and S_2 to ESj. Hence, at ES_i:

MSS_SSEQi[j] of S_1 < $MSS\_SSEQ_i[j]$ of S_2 (1)

By Lemma 4.2, if S\ has not been delivered to ESj, at ES_j:

$MSS\_SRCV_j[i]$ < $MSS\_SSEQ_i[j]$ of S_1 (2)

From (1) and (2), $MSS\_SSEQ_i[j]$ of S_2 □ $MSS\_SRCV_j[i]$. It is not satisfied the step A2 of Subscription_forwarding algorithm and S_2 is appended to queue S_PENDINGj until it is deliverable. Hence, it ensures that S_2 cannot be forwarded before *S).*

Case 2. S_1 and S_2 are forwarded by different event servers (it occurs when a consumer moves and connects to a new event server).

Suppose a consumer MH_s has sent S_1 to ESj. Then MH_s moves and connects to ESj.

ES_i transfers S_1 stored in $SSUBQ\_MH_s$ to ESj (Step A2. (1) of handoff module) during handoff. If MH_s subscribes S_2 to new event server ESj, ESj will execute subscription S_2 after handoff procedure. Hence S_2 is forwarded after S\ to neighboring event servers.

4.5.2 Liveness Property Proof

Lemma 4.3. Every handoff procedure for a mobile user terminates.

Proof. Suppose a mobile consumer MH_s (or) mobile producer MH_p is connected to ES_i. Assume it first moves and connects to a new event server ESj. Then it connects to ES_k. Since ESj does not process the ***Handoff-begin***message (Al. (1) and A4. (1) of handoff module for MH_s and MH_p respectively) from ES_k until it receives the *Handoff-over* message (A2. 3(d) and A5. (3) of handoff module) from ESj, therefore we have

Handoff-over of ESj → *Handoff-begin* of ES_k

Suppose ES_i forwards notifications *N*\ before N_2 to ESj and ESj have subscriptions which **match** *N*\ and N_2. *Liveness* property must satisfy that N_2 is eventually delivered to consumers or event servers.

Case 1. An event server ESj sends *N*\ and N_2 to an event server ESj.

Assume that the notification N_2 sent by ESj that cannot be delivered at ESj indefinitely. It implies that MSS_NSEQ$_i$[j] of N_2 □ MSS_NRCV$_j$[i] (step A5 of Notification_delivery algorithm). This condition means that N_2 is not the next notification from ESj to be executed at ESj. Hence, there must be some notification *N*\ that ESj has not received before N_2. Since the channels among event servers are reliable, therefore ESj receives N_1 eventually. Hence the notification N_2 is eventually delivered to consumers or event servers.

Case 2. *N*\ and N_2 are received by a consumer MH_s from different event servers (i.e., it occurs when a mobile consumer moves and connects to a new event server).

Suppose that the consumer MH_s has moved from ES_j to ES_k before receiving N_1. The event server ESj transfers *N*\ to ES_k during handoff procedure (Al. 2 (a) or Al. 2 (c)) and ES_k will deliver *N*\ to the consumer MH_s. By Lemma 4.3, since a handoff procedure eventually terminates, ES_k delivers next matched notification N_2 to MH_s.

Case 3. N_1 and N_2 are published by producer MH_p to different event severs (i.e., it occurs when a mobile producer moves and connects to a new event server).

Suppose that the producer MH_p has successfully published N_1 to ESj and then it moved from ESj to ES_k. MH_p publishes a notification N_2 to ES_k but ES_k does not process N_2 before terminating handoff procedure with ESj. By Lemma 4.3, since a handoff procedure eventually terminates, ES_k delivers/forwards N_2 to consumers/event servers.

4.6 Analysis of Causal Ordering Algorithm

4.6.1 Ordering Complexity

Message overhead. Causal ordering algorithm need to append extra information to each subscription/notification to maintain causal ordering. The information is essentially overhead that increases the transmission delays when it is passed over the network. The algorithms reported in [2, 39, 65, 74, 86] maintain causal ordering among MSSs. Thus the *message overhead* required to maintain causal ordering is proportional to the number of MSSs as shown in Table 4.1. In our algorithm for event notification service systems, it is enough to maintain causal ordering among neighboring event servers and the algorithm needs to append a sequence number on each forwarded subscription/notification (Step A3 and A6. (2)). Hence, the *message overhead* is (9(1) for both subscriptions and notifications in our causal ordering algorithm.

Handoff complexity. When a mobile host connects from a MSS to another, the handoff procedure is executed. Tn existing algorithms [2, 39, 65, 74, 86] for mobile computing systems, *handoff complexity* is proportional to the number of MSSs or mobile hosts to maintain causal ordering as shown in Table 4.1. For example, Algorithm AV-2 [2] uses $O(n_s)$ messages of size 0(1) and Algorithm LH (39) uses 0(1) messages of size $O(n_s)$ when a mobile host moves to new MSS. In handoff module of our causal ordering algorithm, the previous event server is not necessary to send any information or any sequence numbers to other event servers except the event server to which the mobile host moves (Step A2. (1) and A5. (3)). Hence, the *handoff complexity* is 0(1) for both producers and consumers.

Space overhead. In an event notification service system, the main data storage is routing table at each event server although causal ordering algorithm is not implemented. We investigate the *space overhead* required for causal ordering,

ESj maintains four vectors of sequence number tor receiving and sending subscriptions/notifications (MSS_SSEQ_i, MSS_NSEQ_i, MSS_SRCV_i and MSS_NRCV_i) as described in Section 4.4. Each vector has size of n_{ns}, the number of neighboring event server of ESj. ESj also maintains two set of sequence numbers MH_SENT_p and MH_DELIV_s for its producers and consumers respectively. These sets have size of n_c which is the number of clients (producers and consumers). Hence, the *space overhead* for sequence numbers is 0 $(4n_{ns} + n_c)$.

ES_i uses two set of queues ($S\_PENDING_k$ and $N\_PENDING_k$) for each neighboring event server ES_k to store temporarily subscriptions and notifications which received with out of order until these subscriptions and notifications are deliverable (Step A2 and A5 of static module). Suppose the maximum number of out-of-order subscriptions or notifications from each ES_k is n_m until they are deliverable. Hence, these two queues have size of $(n_{ns} \times 2n_m)$. In practice, ESj receives most of the subscriptions and notifications with order and the number of subscriptions and notifications stored in these queues is very small. Hence the *space overhead* to store for out of order subscriptions and notifications is $O(n_{ns} \times n_m)$.

4.6.2 Duplication and Loss of Events

The advantage of the general peer-to-peer architecture is that it requires less coordination and offers more flexibility in the configuration of connections among servers [13]. The drawback of having redundant connections is that special algorithms must be implemented to avoid cycles and to choose the best paths. Hence, none of event notification service systems in the literature use general peer-to-peer architecture to avoid multiple paths of a notification to consumers and our algorithm also intends to provide for acyclic peer-to-peer architecture.

Lemma 4.4. Let *N* be a notification published by a mobile producer MH_p to ESj. ESj receives *N* with sequence number $\text{MH\_NSEND}_p \simeq k$. Then MH_p migrates to the area covered by ESj.

If ESj has received N_1 successfully, then $\text{MH\_SENT}_p \geq k$ at ESj.

Proof. Let ESj receives *N* and updates its sequence number $\text{MH\_SENT}_p \simeq k$. ESj transfers the sequence number $\text{MH\_SENT}_p = k$ to ESj during handoff (Step A5. (3)).

Suppose that ESj receives N_2 with sequence number $\text{MH\_NSEND}_p \leq k$ from MH_p. It will cause that N_2 cannot be deliverable (Step A4 of static module).

$N_1 \rightarrow N_2$ at MH_p, hence MH_NSEND_p of N_2 > MH_NSEND_p of N_1. If N_1 has been delivered to ESj, then MH_NSEND_p of N_2 | ! k at MH_p. Hence, MH_SENT_p *(=k)* < MH_NSEND_p of N_2 at ESj and N_2 is deliverable.

4.6.2.1 Subscriptions

Loss: When a consumer MH_s moves to a new event server, it resends subscriptions which have not received acknowledgments to the new event server to avoid loss of its subscriptions (Step A6 of handoff module).

Duplication: The previous event server need not send any sequence numbers to avoid duplication of subscriptions because each event server executes *covering test* (Step A3 of static module) for each received subscription. Suppose a consumer MH_s has sent a subscription *S* to previous event server. The previous event server transfers subscriptions of MH_s to the current event server ($SSUBQ\_MH_s$ of Step A2. (1) of handoff module). The new event server re-subscribes those subscriptions on behalf of MH_s for establishing new routing paths of notifications to MH_s. Although a consumer MH_s resends *S* to new event server, the new event server docs not forward S_1 again as the result of covering test (i.e., *S* $\subseteq S_1$).

4.6.2.2 Notifications

Loss: **If** a producer MH_p moves to a new event server, the MH_p republishes the events which have not received acknowledgments to the new event server to avoid loss of notifications. (Step A6 of handoff module). The previous event server transfers notifications ($DELIVQ\_MH_s$ of Step A2 of handoff module) that have not been delivered to the consumer to the new event server during handoff. While executing handoff procedure, the previous event server may receive notifications for MH_s. To avoid loss of these notifications, the previous event server sends them to new event server by marking as "*old*" (Step Al. 2(c) of handoff module). The new event server delivers those "*old*" notifications to MH_s immediately.

Duplication: In case of changing a producer's location, the previous event server ESj sends MH_SENT_p (the last sequence number of notifications published by producer MH_p which it has received) to the new event server ESj (Step A5. (3) of handoff module). Suppose the previous event server has received a notification *N* published by MH_p with "$MH\_NSEND_p = k$". By Lemma 4.4, $MH\_SENT_p \geq k$ at ESj. Hence, if MH_p has not received acknowledgment from ESj, it resends *N* to ESj. The ESj forwards/delivers *N* only if N_1 is greater than k (Step A4 of handoff module). It ensures that the algorithm avoids duplication of notifications published by a producer.

In case of changing a consumer's location, the previous event server sends MH_DELIV_s (the last sequence number of notifications, which it has sent to consumer MH_s) to the new event server (Step A2. (1) of handoff module). The MH_s also registers to a new event server with the last sequence number MH_NRCV_s of received notifications (Step Al. (1) of handoff module). The new event server delivers notifications greater than MH_NRCV_s to MH_s (Step Al. 2(a) of handoff module). It ensures that an event server cannot deliver duplicated notifications to the consumers.

4.7 Evaluation

To test the performance of causal ordering algorithm, we conducted simulations with varying number of event servers and rate of publications, p *{notifications per minute).* We study the cost of proposed causal ordering algorithm in terms of *latency* for delivery of notifications. *Latency* of a notification means the time required from a producer which publishes the notification to an interested consumer. In simulation, we use many publication rates to study the effects of different loads on causal ordering algorithm. In event notification service systems, on receiving a notification, each event server executes causal ordering algorithm to check that it is deliverable. We change number of event servers in the network to observe the effects of different distances (i.e., hop counts) between producers and consumers.

4.7.1 Simulation Setup

Simulation experiments were performed on a normal personal computer (PC), 2.8GHz Pentium 4 HT, 512M RAM running Windows XP. The simulation was done in Java (jdk1.4.2) language. The network topology has a major impact on any experiments. Its main parameters are: the number of event servers of the network, the number of neighboring event servers of each event server, and the diameter of the network (which is the longest path connecting two arbitrary event servers). In simulation, event servers are connected in hierarchical and symmetrical, i.e., tree-like topology and the number of event servers are varied from 4 to 70 for various network sizes. Each event server except leave nodes has 3 neighboring event servers and the flows of messages among event servers are bidirectional. We use 512 bytes for the size of each message. The propagation delay of a message between two event servers is exponentially distributed random variable with a mean of 7 ms.

Simulation only uses one message type (notification) and it is assumed that every notification has interested consumers at every event server. Hence, each event server forwards every notification received to its neighboring event servers. A producer generates messages (notifications) to its local event server which is chosen at randomly. For incorrect sequence number of messages, messages are attached by randomly assigning sequence number within variation range of ±2 at some event servers, without loss of any sequence number. For example, after sequence number 2, the possible sequence numbers are 3, 4 and 5 although the correct sequence number is 3.

When simulation starts, event servers are connected each other and waiting for messages. When a producer generates a message to an event server, the event server forwards the message to neighboring event servers. All event servers which receive the message execute the causal ordering algorithm before forwarding it to neighing event servers. Consumers receive messages with correct sequence and without loss.

For each message, a producer has publication time and a consumer has receiving time. The latency of a message is the difference of these time values. The value of every point (latency in ms) in the graphs is an average of the results of 10 experiments performed.

4.7.2 Simulation Results

Figure 4.8 and 4.9 show the average maximum latency of messages in the network of 25 and 50 event servers respectively. The maximum latency of a message generally depends on the number of hops it is routed. We take the maximum delays required to deliver messages to any consumers in experiments and calculate the average of these delays to represent maximum latency in the graphs. An event server executes causal ordering algorithm (Step A5 of static module) and forwards every message received to its neighboring event servers (Step A6. (2) of static module). As expected, the latency values of large network (Figure 4.9) are larger than the values of small network (Figure 4.8). The higher publication rate of messages causes the larger load at event servers and increases the number of messages waiting for causal ordering algorithm (Step A5 of static module). The graphs show that the differences of latencies between the process with causal ordering algorithm and the process without causal ordering algorithm become large with the message publication rate.

Similarly, the graphs of Figure 4.10 and 4.11 show the average latencies for two different network sizes. The average latency means the average of all latencies required to deliver messages from a producer to all interested consumers located in different distances from the producer. In the graphs, the difference of average latencies between the processes with and without causal ordering algorithm is smaller than the difference of maximum latencies (Figure 4.8 and 4.9) by increasing publication rate.

The graphs described above show that the latency increases rapidly before point A for both causal ordering and without causal ordering. In case of low publication rates (i.e., before point A), after an event server forwards a notification, it has not received next notifications. It causes some delay of waiting for next notification to restart processes of the event server. In case of high publication rates (i.e., after point A), after an event server forwards a notification, the next notifications are waiting and ready to process. Hence after point A, an event server has no delay of waiting for notifications and the latency depends on increasing load of notifications (messages) only.

Figure 4.12 looks at the effect of varying network size for two different publication rates (i.e., low and high rates). Every event server which is located along the routing paths from a producer to consumers executes the causal ordering algorithm. If the number of event servers increases in the network, the routing path of a message between a producer and interested consumers (i.e., hop counts) become large. Hence, the graph shows that the average latencies consistently increase with the number of event servers for both publication rates. The graph also demonstrates that the high publication rate causes a reasonable cost (in terms of latency) by using causal ordering algorithm.

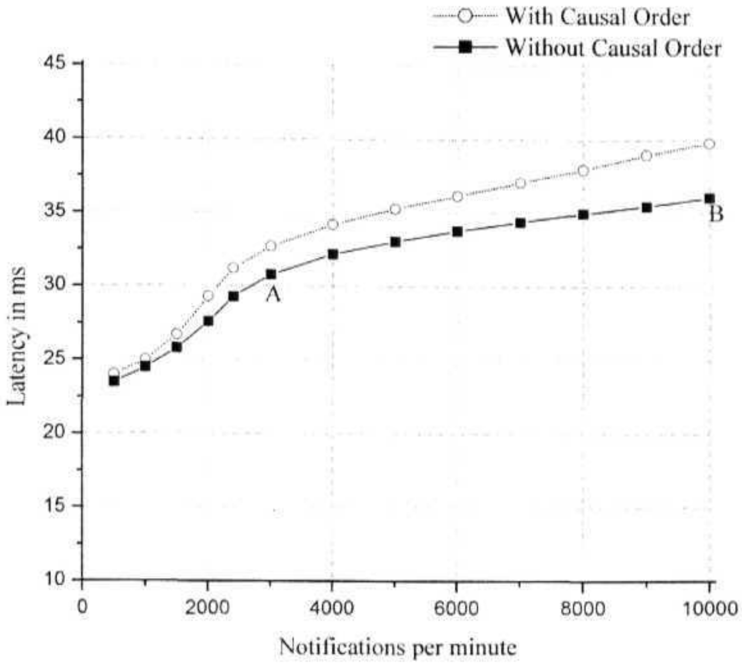


Figure 4.8: Maximum latency for 25 event servers

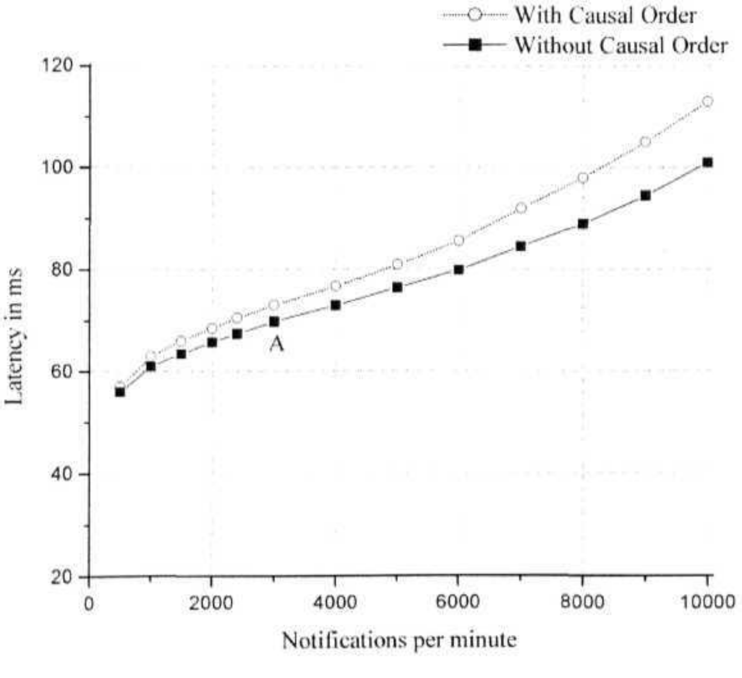


Figure 4.9: Maximum latency for 50 event servers

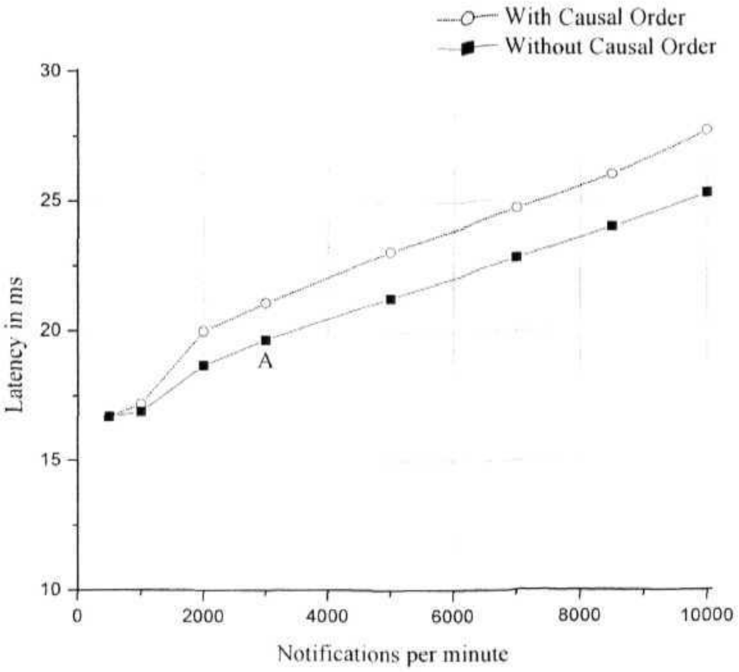


Figure 4.10: Average latency for 25 event servers

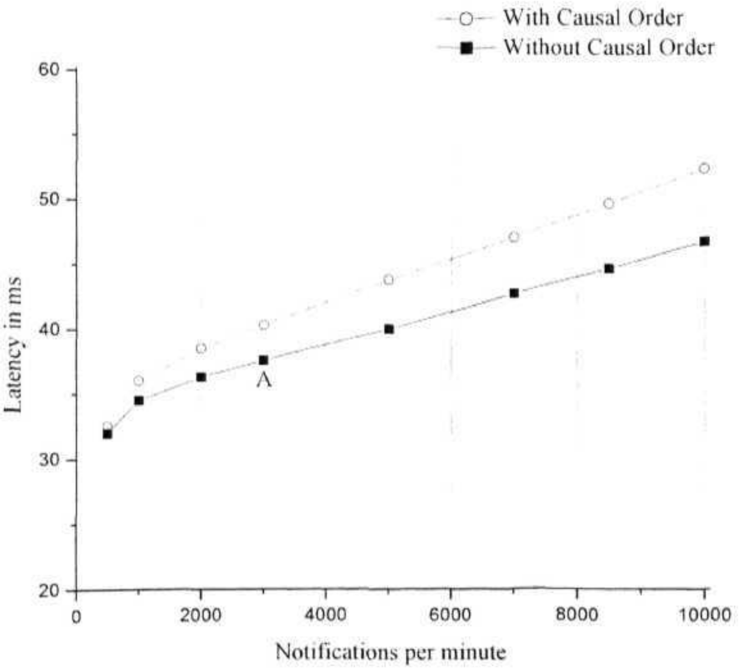


Figure 4.11: Average latency for 50 event servers

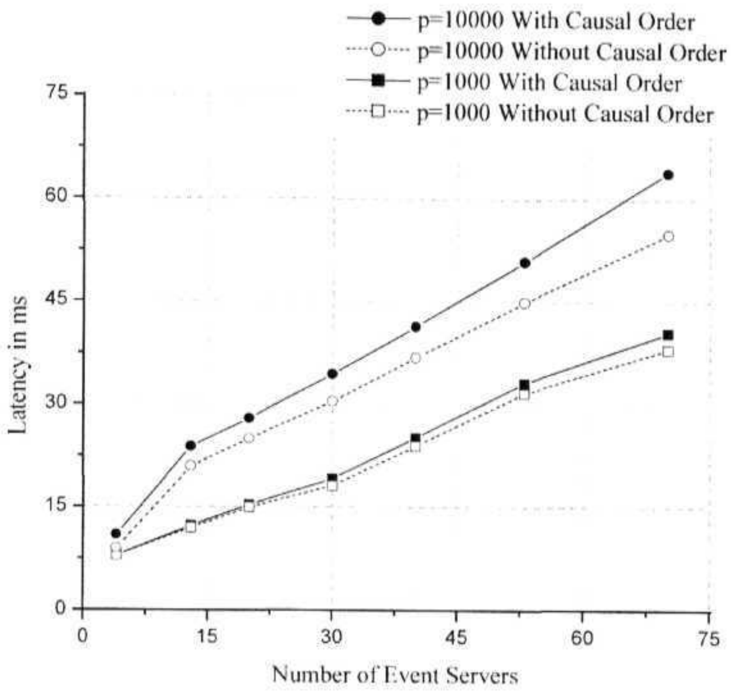


Figure 4.12: Average latency with various network sizes

4.8 Summary

We present a causal ordering algorithm for event notification service systems in mobile environment. An event notification service system in a mobile environment faces some problems due to incorrect sequences of messages, loss and duplication of messages due to movements of producers and consumers. The proposed causal ordering algorithm is designed to avoid these problems.

Our causal ordering algorithm is developed for event notification service systems which use *subscription forwarding* routing strategy (Section 2.2.4). We prove that our causal ordering algorithm satisfy the *safety* and *liveness* properties. Then we analyze the algorithm to investigate ordering complexity. The *message overhead* and the *handoff complexity* is *()){\)* in causal ordering algorithm for ENS system as described in Section 4.6.1. It is very low by comparing other causal ordering algorithms (described in Table

4.1) for distributed mobile system. Moreover, the message overhead does not vary with the number of mobile hosts. Hence, the proposed algorithm is scalable and suitable for mobile environment with constraints such as low power consumption, low computing power of mobile devices, and low bandwidth of wireless links. We prove that the causal ordering algorithm avoids the losses and duplication of subscriptions/notifications. In experiments of simulation, we demonstrate that the causal ordering algorithm causes relatively low cost in terms of latency for delivery of notifications for different network sizes and publication rates in event notification service system.

Chapter 5

Just-in-time Delivery in Event Notification Service Systems for Mobile Users[3]

In this chapter, we present strategies that guarantee just-in-time delivery of published events to subscribing mobile users. A mobile user may disconnect and reconnect to different event servers using a low bandwidth wireless interface. Mobility may cause delay in providing the service as new routes should be established for delivery of notifications. This delay is not tolerable for many deadline-bound applications, like stock trading, auctioning, monitoring of traffics, etc. We proposed proactive strategy called location-based pre-handoff approach that creates virtual consumers at a set of expected locations of a mobile consumer in order to set up flows of notifications before the mobile consumer reaches any of those locations.

5.1 Introduction

In this chapter we investigate on the issues involved in delivery of events to mobile users (consumers) as soon as they reach at new locations. We term this kind of event delivery as just-in-time delivery of events. Portable computing devices such as notebook computers, personal digital assistants (PDAs) and cellular phones along with wireless communication technology enable people to carry computational power with them as they keep changing their physical locations. Many of applications such as stock trading system, auction system, reservation system, sports reporting and traffic condition monitoring systems need to deal with data that is valid for limited time period. Mobile consumers while on move keep moving in and out of cells. These movements cause delay in delivery of events

[3] This work is published In *Proceedings of International Conference on Intelligence in Communication Systems (INTELLCOMM 2004)*, volume 3283 of LNCS, Springer, Bangkok, Thailand, pages 190-198, November 2004 [43].

because event notification service needs to define paths from event servers to mobile consumers' new locations. Because of this delay, time constrained data item may loose their validity causing concerns and losses to mobile applications. Seamless availability of events in spite of mobility is a possible solution. In other words, as a mobile consumer moves, events addressed to it must move along so that the mobile consumer on joining a location can immediately avail events of its choice. Such provision helps to deliver events before they loose their validity.

Several implementations of ENS systems (e.g., SIENA [11], JEDI [18], REBECA [27], and Elvin [79]) provide mobility support for delivering events to mobile users. Java Message Service (JMS) can store messages for disconnected consumers by allowing consumers to create durable subscriptions. In JMS, a consumer can register a durable subscription with a unique identity that is retained by JMS [77]. Obviously mobile users expect uninterrupted service. In order to achieve it, users* subscriptions are not only to be registered at their new locations, but new routes need to be established from event servers to their new locations. This causes latency in delivery of events at new locations. However, none of previously mentioned approaches has addressed the problem of providing just-in-time delivery of events, though this issue is important from user point of view. For the purpose we propose a pro-active concept called location-based pre-handoff approach [43] that disseminates events to all possible locations surrounding current location of a mobile consumer.

The rest of the chapter is organized as follows. Section 5.2 describes a model of mobile network for event notification service systems, location management scheme, and also presents definitions and notations used in the text. Section 5.3 describes the concept of location-based pre-handoff which creates virtual consumers at a set of expected locations of a mobile consumer and the corresponding algorithm that uses regional route map, i.e., a physical route map of a small neighborhood of the locations of mobile consumers. Analysis of the location-based pre-handoff algorithm is described in Section 5.4. Section 5.5 investigates the cost of message complexity for location-based pre-handoff with experiments in simulation.

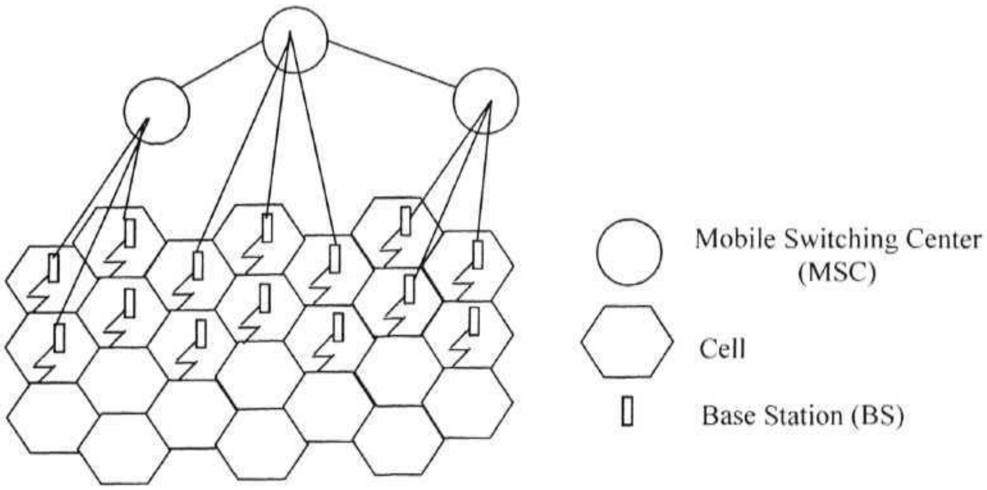


Figure 5.1: A cellular mobile architecture

5.2 System Model and Definition

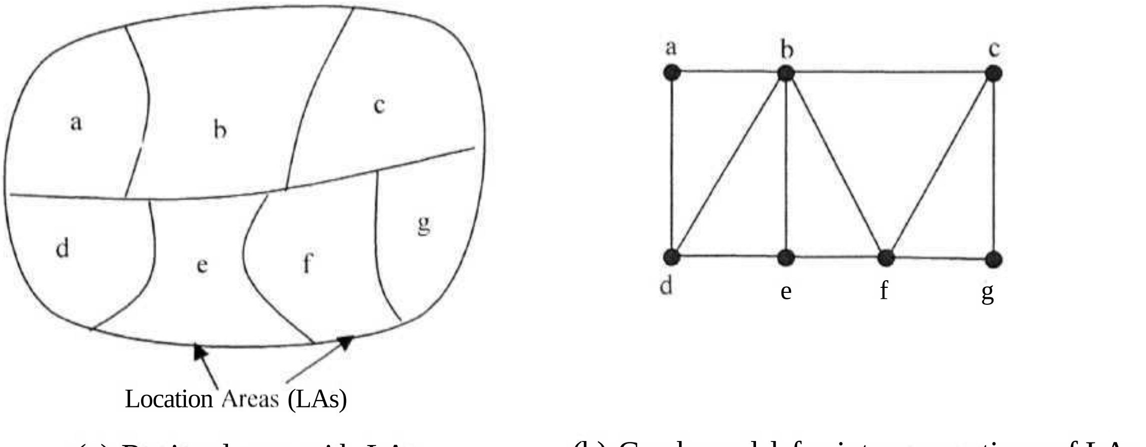
5.2.1 System Model

We choose a model used in mobile cellular environment which is a collection of geometric areas called cells (Figure 5.1). Each cell is serviced by a base station (BS) located at its center [20]. Mobile terminals are connected to the network via the base stations. Cells can have different sizes: *picocells* arc commonly used in indoor environments; *microcells* are used within cities; *macrocells* are used in rural areas and to cover highways [85].

A network is assumed to be divided into a few big location areas (LA) as described in [30]. The graph model is used for representing interconnections of LAs as shown in Figure 5.2. Each LA, consisting of many cells (about 50 to 100), is served by a mobile switching center (MSC). An MSC acts as a gateway of a mobile network to the existing wired network. The base stations of a LA are connected to a MSC which stores the location and service information for each registered local mobile user (consumer or producer). Tracking of the current location of a mobile user, involves two basic operations, *terminal paging* and *location update* that determine the mobile user's exact location within the cell granularity. Many work have been done on location management schemes |20] such as profile-based location update scheme, selective LA update scheme, movement-based update scheme, timer-based update scheme and distance-based update scheme.

In event notification service systems, the routing of an event is dependent on event content rather than particular destination address and each event server is only aware of its neighboring event servers through which events propagate using subscriptions stored in routing tables. When an event server receives an event it checks the event against all subscriptions of its routing table. Then it forwards the event to neighboring event servers or delivers the event to interested local consumers. Whenever a mobile consumer moves from the current cell into a new cell which is under another event server, the handoff process is executed between previous event server and new event server to update routing tables.

Usually two different event servers deal with communication and event routing separately. However, MSCs with enough computing resources can also be used as event servers. MSCs can be considered as MSSs of the basic architecture for wireless network as described in Section 3.5. Based on this architecture, the architecture of event notification service system for just-in-time delivery is presented in Figure 5.3. For location-based pre-handoff algorithm of just-in-time delivery service, the location manager maintains location tables described in Section 5.2.2. Whenever a mobile consumer changes its location (cell), the *location manager* checks the cell with the location tables and decides to execute the "Pre_Handoff" to send *pre-handoff* request message. An event server executes "Cancel Pre_handoff" to send *cancel-pre-handoff* message for canceling of pre-handoff processes of a mobile consumer as described details in Section 5.3.2. In the following sections, we



(a) Regional map with LAs (b) Graph model for interconnections of LAs

Figure 5.2: Regional map and graph model

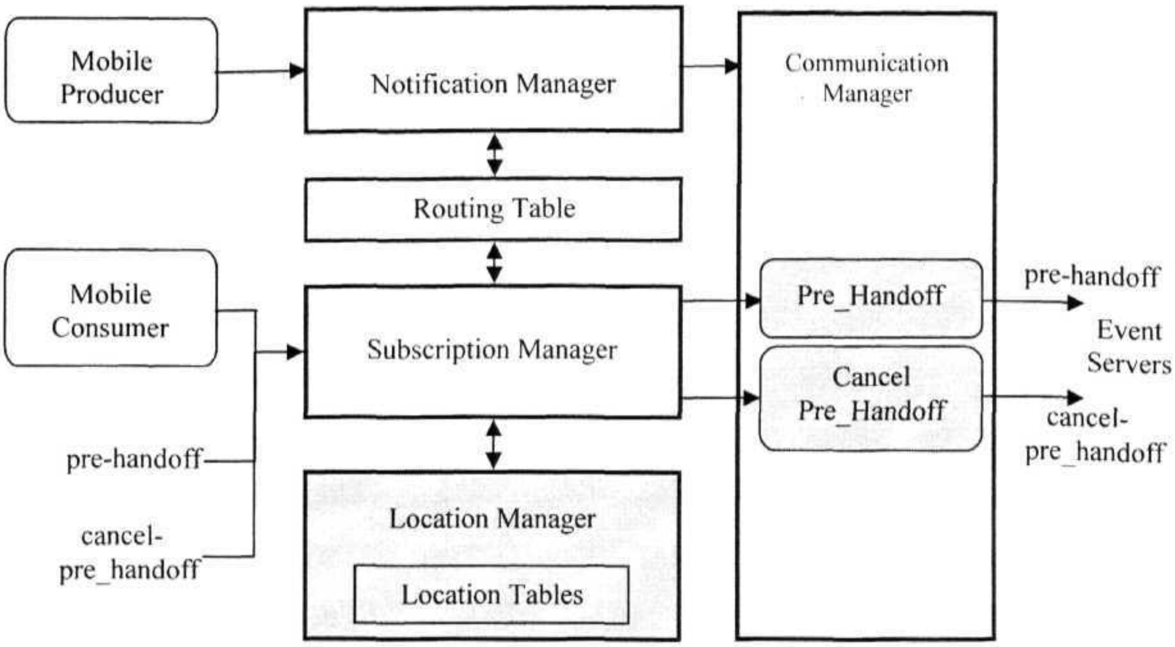


Figure 5.3: The architecture of ENS at event server for just-in-time delivery

use the term event servers (ESs) instead of MSCs to be in tune with discussion concerning event notification service systems. We assume that all connections are bi-directional. The following sections require realizing of subscription, notification and routing table (RT) described in Section 3.5.1.

5.2.2 Location Tables of Mobile Hosts

Each event server maintains mobile hosts location table (MET), and local border cell table (BCT) besides the routing table (RT). A MLT is a collection of tuples {<mobile_host_ID>, <cell_ID>}. In event notification service system, each event server decides to deliver/forward the event by checking against all subscriptions of RT. Routing paths of events depend on content of events. As the locations of mobile consumers keep changing, event routing paths may vary from time to time. The event routing other than contents also depends on consumer locations available at Mobile Host Location Table (MLT). Every time a mobile consumer crosses a boundary of a cell there is an update in MLT of an associated LA.

Each event server maintains a border cell table (BCT), a list of *harder event servers* adjacent to each border cell. Border cells mean the cells located at the periphery of a LA and adjacent to the cells of neighboring LAs. BCT is a collection of tuples {<border_cell>, <border_event_servers>}, i.e., BCT = {<CB>, <BES>}. By '·' extension convention, BCT·CB_k·BES, denotes the border event servers with respect to the border cell CB_k. The set of neighboring cells adjacent to a border cell can be found statically from the regional route map described in Figure 5.2. But it is important to minimize the size of the set of cells to where a mobile consumer may migrate from a border cell of a location area. Otherwise, the event notification service system can not scale up as number of mobile consumers increase. We can also use a pre-declared movement pattern of the mobile consumer as described in [30J to predict the possible cells where a mobile consumer may show up from a border cell of a location area.

5.2.3 Just-in-time Delivery

We investigate on the issues involved in delivery of events as soon as mobile consumers reach at new locations. We term this kind of event delivery as *just-in-time delivery* of events to deal with data that is valid for limited time period. For just-in-time delivery, it is necessary to minimize the *latency* of delivering events. The *latency* of an event is the time required until a consumer receives the event published by a producer through event dissemination tree.

Suppose T_{val} is the valid time period of an event after publishing. The latency of receiving an event for a mobile consumer at new location consists of the following delays.

$$\text{latency} = t_{pro} + t_d + t_{con} + t_{trans} + t_{sub}$$

where t_{pro} = delay from producer to its event server

t_d = propagation delay among event servers of fixed network

t_{con} = delay of an event server to its interested consumer

t_{trans} = delay of transferring subscriptions from previous location to new location

t_{sub} = delay of issuing subscriptions to dissemination tree (it depends on the diameter of network)

Hence, for just-in-time event delivery service, it requires to achieve

$$(i)\ t_{trans} = 0 \quad \text{and} \quad (ii)\ t_{sub} = 0$$

which may help to deliver an event at time $t \leq T_{val}$.

Our proposed location-based approach eliminates delays of t_{trans} and t_{sub} at new location of the mobile consumer as the processes which cause these delays are executed before the mobile consumer reaches its new location.

5.3 Pre-handoff Approach

The pre-handoff is a proactive strategy which initiates updates of location tables of LAs to which a mobile consumer may visit during its travel. The main idea behind pre-handoff is to create virtual consumers at a set of expected locations of a mobile consumer in order to set up flows of notifications before the mobile consumer reaches any of those event servers. Loosely speaking, a virtual consumer is created on all event servers to which the mobile consumer is expected to connect in the "near' future. The set of virtual consumers must change when the mobile consumer moves.

5.3.1 Simple Pre-handoff

For just-in-time delivery of events, an event server creates a virtual consumer (VC) of a moving mobile consumer and proactively forwards the subscriptions of the moving mobile consumer to the event servers to which the consumer may connect, We describe simple pre-handoff approach as the basic idea for location-based pre-handoff presented in the following section. In simple pre-handoff approach, an event server (1) sends a pre-handoff request and (2) pre-forwards (the term "pre-forward" is used instead of "forward" as the consumer has not moved to new location) subscriptions of a moving mobile consumer to all border event servers. On receiving pre-handoff request and pre-forwarded subscriptions of a particular mobile consumer, an event server (1) creates a virtual consumer and (2) subscribes the subscriptions to establish paths of notifications with respect to subscriptions of the virtual consumer. We call these paths related to virtual consumers as virtual paths. The notifications related to the mobile consumer's subscriptions follow these virtual paths to arrive at neighboring LAs at which the consumer may possibly register after sometime.

However, executing pre-handoff at every border event server causes substantially unnecessary overheads if the number of moving mobile consumers becomes large. In order to avoid this problem, only limited number of border event servers should be chosen for creation of virtual consumers.

5.3.2 Location-based Pre-handoff

If a mobile consumer keeps moving in the interior cells of a LA, then the event server for the LA will be able to directly deliver the related notifications to the consumer. As these consumers are very unlikely to move out to new LA, there is no need to create virtual consumers for such consumers. But if the consumer is moving around the border of a LA, it is likely that he/she may stray into a neighbouring LA. So we need to consider strategies to deliver the notifications related to mobile consumers moving around border cells of a LA should it ever stray into a cell under a neighbouring LA. It is done as follows.

Each event server executes pre-handoff for mobile consumers which are moving in its border cells. As described in Section 5.2.2, each event server maintains a local border cell table (BCT) that stores border cells and their adjacent LAs (i.e., border event servers). So, an event server uses BCT in order to determine the border event servers on which pre-handoff processes have to be executed on behalf of a moving consumer. The proposed mechanism creates virtual consumers (VC) and pre-subscribes at some border event servers of possible future locations instead of all border event servers. When a mobile consumer MH_s gets connected to a new event server on which a virtual consumer is already running, it finds that its subscriptions exist and routing paths are already established. This ensures that MH_s can receive matched notifications immediately after connecting to a new event server.

Each event server needs to cancel pre-handoff processes of a mobile consumer MH_s in the following cases:

(1) The MH_s is not in one of its border cells. It has to tackle two possible subcases:

(a) MH_s moves from a border cell to a non-border cell (interior cell) within current LA.

(b) MH_s moves to a border cell located within another LA.

(2) The MH_s stops moving (i.e., it does not come out of a border cell) although the mobile is in a border cell.

In case 1, if MH_s has moved from a border cell to an interior cell under the current LA there is no need to execute a pre-handoff. Also if MH_s already move to a border cell under a new LA, unless pre-handoff is cancelled, it may lead to ping-pong situation. In case 2, the algorithm also needs to cancel pre-handoff processes. We can define time bound T to determine whether the consumer is moving or not. If a mobile consumer stays in a border cell greater than time T, the algorithm assumes that the mobile consumer is not moving (i.e., flagmov $(MH_s) = 0$). In these cases, it is necessary to cancel virtual consumers running at predicted locations. Hence the cost of location-based pre-handoff approach is dependent on the number of mobile consumers which migrate to new locations as predicted by using BCT. We term this prediction as Correct Movement Prediction (CMP). For example, CMP = 0.5 means that 50% of moving mobile consumers which have virtual consumers in other location areas move to these location areas as predicted.

When a mobile consumer MH_s connects to a new event server, it must handle one of the following two cases.

(1) The new event server has a virtual consumer of MH_s.

(2) The new event server does not have a virtual consumer of MH_s.

In case 1, the new event server need not execute handoff process which includes transferring subscriptions and rc-subscribing and it can deliver matched notifications to MH_s immediately after connection. In case 2, the new event server did not create the virtual consumer of MH_s due to following reasons:

(1) The mobile consumer MH_s was disconnected for long time and it reconnects to another event server. Hence, the new ES does not have a VC for MH_s.

(2) The mobile consumer MH_s was in a border cell for long time (i.e., greater than 7) and so the pre-handoff process was cancelled.

A mobile consumer MH_s which is moving in a border cell has its virtual consumers at some border event servers. If MH_s sends a new subscription, the subscription needs to be forwarded to border event servers which have virtual consumers of MH_s. Hence, the values of Subscription Mobility Ratio (SMR) also affect on location-based pre-handoff algorithm. SMR is a measure of the number of subscriptions/unsubscriptions which a moving consumer subscribes/ unsubscribes from a cell while moving. For example, SMR = 1 means that a moving consumer in a border cell can send one subscription/unsubscription to its current event server before reaching to a new cell belonging to the same LA or another LA. SMR depends on the rate of subscriptions and the speed of moving mobile consumers. The algorithm for location-based pre-handoff approach is described in the following section.

5.3.3 Algorithm for Location-based Pre-handoff

Suppose a mobile consumer MH_s is moving in a border cell of event server ES,. Then location-based pre-handoff algorithm is executed at ESj and at some border event servers for just-in-time delivery of events. The algorithm for location-based pre-handoff approach is described in Figure 5.4. As discussed in Section 5.3.2, when a mobile consumer MH_s moves to a border cell, there are two possibilities:

(1) Ml-I_s moves to new LA under a different event server, or

(2) It either stops moving, or moves back into an interior cell belonging to the present LA (served by the current event server).

In case 2, not only pre-handoff is unwarranted but also if any pre-handoff was initiated for a mobile consumer, it should be cancelled. In case 1, a pre-handoff request is initiated. Also as MH_s enters into a new LA, it must register there. A mobile consumer on reaching a cell which belongs to a new LA registers at the server of LA. Registration is triggered by a border cell of the LA from where MH_s has migrated after a pre-handoff, or it may be initiated by some cell in a new LA where MH_s pops up independently. In first case a surrogate of MH_s is already up under the new event server. So MH_s just needs to replace its surrogate. In the second case, a ***normal handoff*** (Figure 5.4 A3) should be initiated by the cell where MH_s shows up.

Let us discuss how the pre-handoff and registration proceeds in case 1. Suppose a mobile consumer MH_s moves from a location area (LA) of ES, to a LA of ESj. Before MHs connects to ES,, it must pass through a border cell CB_k of ES,. When the border cell CBk sends ***Migration_alert*** message to its event server ES, (Figure 5.4 Al), ESj executes pre-handoff on behalf of MH_s. ES, transfers all subscriptions of MH_s using ***Preforward*** message to a set of border event servers, *e* e E, (i.e., E = BCT·CBk·BES) including ESj (Figure 5.4 Al (1)). Each of the event servers after receiving *Preforward* message creates a virtual consumer VC_MH$_s$ and forwards the subscriptions of MH$_s$ to related event servers (Figure 5.4 A4 and A5) to establish new routes of notifications for MH$_s$. After sending *Preforward* to *e*, ES$_i$ needs to send *Again_Preforward* to *e* whenever it receives a new subscription *S* from MH$_s$ (Figure 5.4 A2). On receiving ***Again_Preforward***,each event server updates its routing and forwards *S* to related border event servers to update routing paths of notifications (Figure 5.4 A6).

ES, executes pre-handoff on behalf of MH$_s$ when MH$_s$ comes in to a border cell. But MH$_s$ may not move after entering the border cell or it may move back to an interior cell. In these cases, ES, sends *CancelPreforward* to those event servers which have VC_MH$_s$ to cancel pre-handoff (as described in Figure 5.4, Al (2)),

When MH$_s$ connects to ES, which already has VC_MH$_s$, ES, changes VC_MH$_s$ as its local consumer and deliver notifications to MH$_s$ without the delay of transferring subscriptions from ES$_i$ and re-subscribing process. If ES, does not have VC_MH$_s$, ESj call ***Normal_handoff***process to execute normal handoff from previous ES,. Then normal handoff process is executed between ES$_i$ and ES$_j$. It is assumed that MH$_s$ moves from the LA of ES, to the LA of ES,. The data structures used in the algorithm are as follows:

- *cell_ MH$_s$* = current cell of mobile consumer MH$_s$.
- SSUBQ_MH$_s$ = queue to store subscriptions sent by local mobile consumer MH$_s$
- CB(ES$_i$) = {C$_1$, C$_2$, ..., C$_k$}= a set of border cells of ES$_i$.
- Cell(ES$_i$) = a set of cells belong to ES,.
- flag_mov(M//$_v$) = 1 if mobile consumer MH$_s$ is moving
- VC(ESjj) = ES$_i$'s awareness of virtual consumers created at ES$_j$ for possible movements of consumers from ESj to ESj.
- VC(ES$_i$) = all virtual consumers located at a server ESj.
- Local(ESj) = a set of local mobile consumers at ES,.

In the algorithm, we use the format {<***message*** *name*> <*parameters*> from <sender> to <receiver>} for sending messages between event servers.

Executed at event server ES_i where MH_s is residing.

```
A1.  On receiving Migration_alert ($cell\_MH_s$) from $MH_s$ to $ES_i$
     begin
       if ($MH_s \in Local(ES_i)$)
       (1)   if ($flag\_mov(MH_s) = 1 \wedge pre\_handoff(MH_s) = 0$)
                $E = BCT \cdot cell\_MH_s \cdot BES$
                  For all $ES_e \in E$ do
                     if ($MH_s \notin VC(ES_{ie})$)
                        (a)   Send Pre_handoffRequest ($MH_s$) from $ES_i$ to $ES_e$;
                        (b)   On receiving Enable_prehandoff ($MH_s$) from $ES_e$ to $ES_i$,
                                begin
                                  Send Preforward ($MH_s$, $SSUBQ\_MH_s$) from $ES_i$ to $ES_e$;
                                  $VC(ES_{ie}) = VC(ES_{ie}) \cup MH_s$;
                                  $pre\_handoff(MH_s) = 1$;
                                end
                     endif
                  endFor
             endif
       (2)   if ($stayperiod(MH_s) > T$)        // $MH_s$ stops moving and reside in a border cell.
                $flag\_mov(MH_s) = 0$;
             endif
       (3)   if ($pre\_handoff(MH_s) = 1$)    // $ES_i$ cancels pre-handoff in the following conditions.
                if ($flag\_mov(MH_s) = 0 \vee cell\_MH_s \notin Cell(ES_i)$)
                // if $MH_s$ stops moving in border cell (or) moves to a neighboring LA.
                  For all $ES_e \in BCT \cdot cell\_MH_s \cdot BES$
                     Send Cancel_Preforward ($MH_s$) from $ES_i$ to $ES_e$;
                  endFor
                endif
                if ($cell\_MH_s \notin CB(ES_i) \wedge cell\_MH_s \in Cell(ES_i)$)  // if $MH_s$ moves to inner cell.
                  For all $ES_e \in BCT \cdot prev\_cell\_MH_s \cdot BES$         // previous cell of $MH_s$
                     Send Cancel_Preforward ($MH_s$) from $ES_i$ to $ES_e$;
                  endFor
                endif
                if ($cell\_MH_s \in CB(ES_i)$)
                // if $MH_s$ moves to another border cell of $ES_i$
                  $ES_c \in BCT \cdot cell\_MH_s \cdot BES$                   // current cell of $MH_s$
                  $ES_p \in BCT \cdot prev\_cell\_MH_s \cdot BES$             // previous cell of $MH_s$
                  For all $ES_e \in (ES_p - (ES_p \wedge ES_c))$         // some BESs are same for 2 cells.
                     Send Cancel_Preforward ($MH_s$) from $ES_i$ to $ES_e$;
                  endFor
                endif
             $pre\_handoff(MH_s) = 0$;
             $VC(ES_{ie}) = VC(ES_{ie}) - MH_s$;
          endif
       endif
     end
```

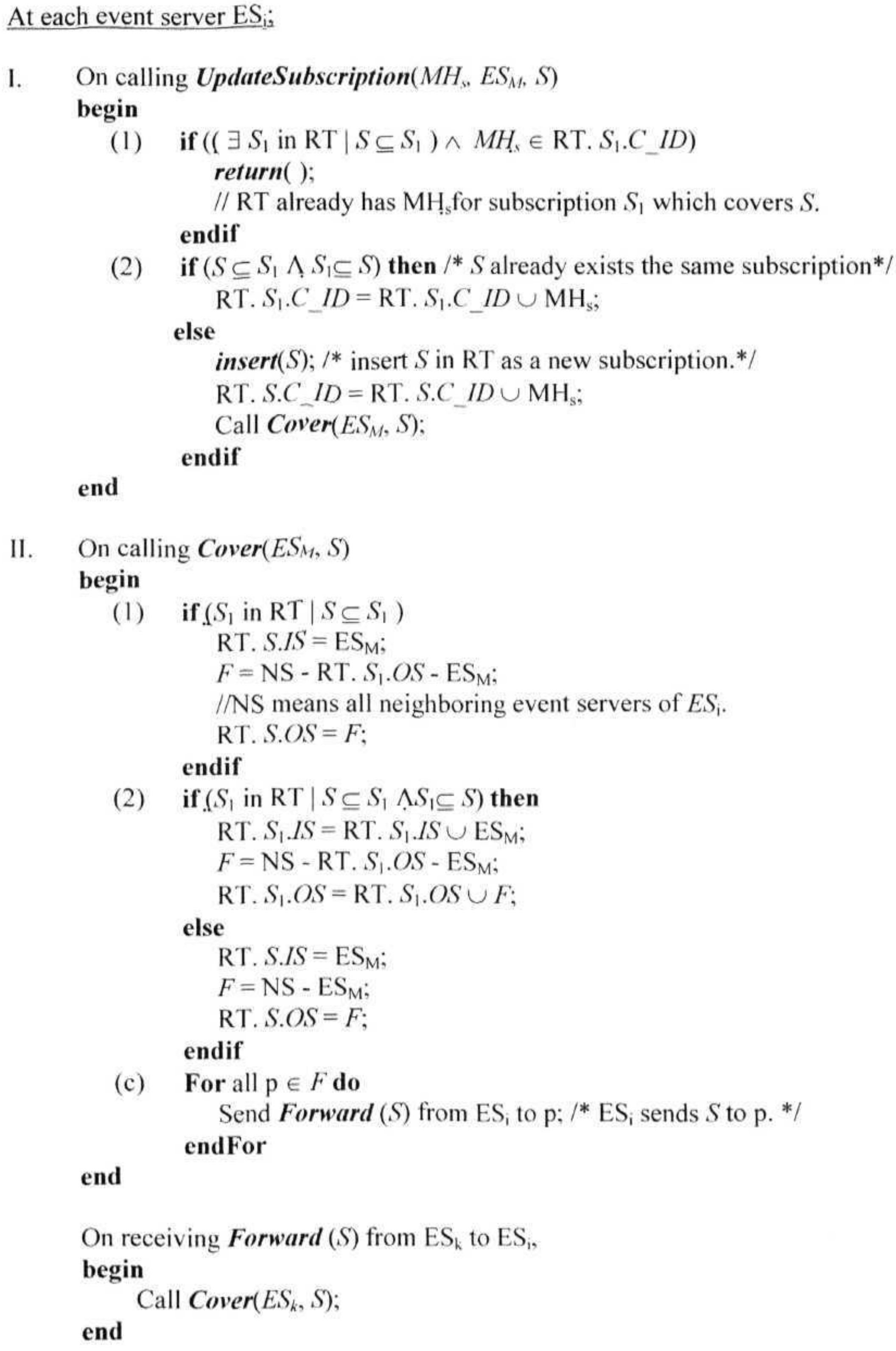
Figure 5.4: Algorithm of location-based pre-handoff

```
A2.   On receiving a subscription $S$ from $MH_s$ to $ES_i$,
      begin
         if (flag_mov($MH_s$) = 1 $\wedge$ pre_handoff ($MH_s$)= 1) then
            //forwarding new $S$ to $ES_e$.
               For all $ES_e \in$ BCT$\cdot$cell_$MH_s$$\cdot$BES
                  Send Again_Preforward ($MH_s$, $S$) from $ES_i$ to $ES_e$;
               endFor
         endif
      end
```

Executed at event server ES_j:

```
A3.   On receiving Registration ($ES_i$) from $MH_s$ to $ES_j$,
       // A host reaching a new LA sends a Registration message to event server of LA.
       // Previous and current event server are $ES_i$ and $ES_j$ respectively.
       begin
            if ($MH_s \in$VC($ES_j$))
               // VC is already up for $MH_s$ coming from another LA
                  (1)   Local($ES_j$) = Local($ES_j$) $\cup$ $MH_s$;
                  (2)   VC($ES_j$) = VC($ES_j$) – $MH_s$;
            else
                 Call Normal_handoff($MH_s$, $ES_i$);
                 /* Normal transfer of subscriptions and notifications for $MH_s$ (Figure 4.3 of
                   Chapter 4).*/
            endif
       end

A4.   On receiving Pre_handoffRequest ($MH_s$) from $ES_i$ to $ES_j$,
         (1)   VC($ES_j$) = VC($ES_j$) $\cup$ $MH_s$;
         (2)   Send Enable_prehandoff ($MH_s$) from $ES_j$ to $ES_i$;

A5.   On receiving Preforward ($MH_s$, SSUBQ_$MH_s$) from $ES_i$ to $ES_j$,
         For all subscriptions $S \in$ SSUBQ_$MH_s$ do
            Call UpdateSubscription($MH_s$, $ES_i$, $S$);
         endFor

A6.   On receiving Again_Preforward ($MH_s$, $S$) from $ES_i$ to $ES_j$,
         (1)   SSUBQ_$MH_s$ = SSUBQ_$MH_s$ $\cup$ $S$;
         (2)   Call UpdateSubscription($MH_s$, $ES_i$, $S$);

A7.   On receiving Cancel_Preforward ($MH_s$) from $ES_i$ to $ES_j$,
         (1)   For each $S \in$ SSUBQ_$MH_s$ do
                  Call Unsubscription($ES_j$, $S$, $MH_s$);
               endFor
         (2)   VC($ES_j$) = VC($ES_i$) – $MH_s$;
```

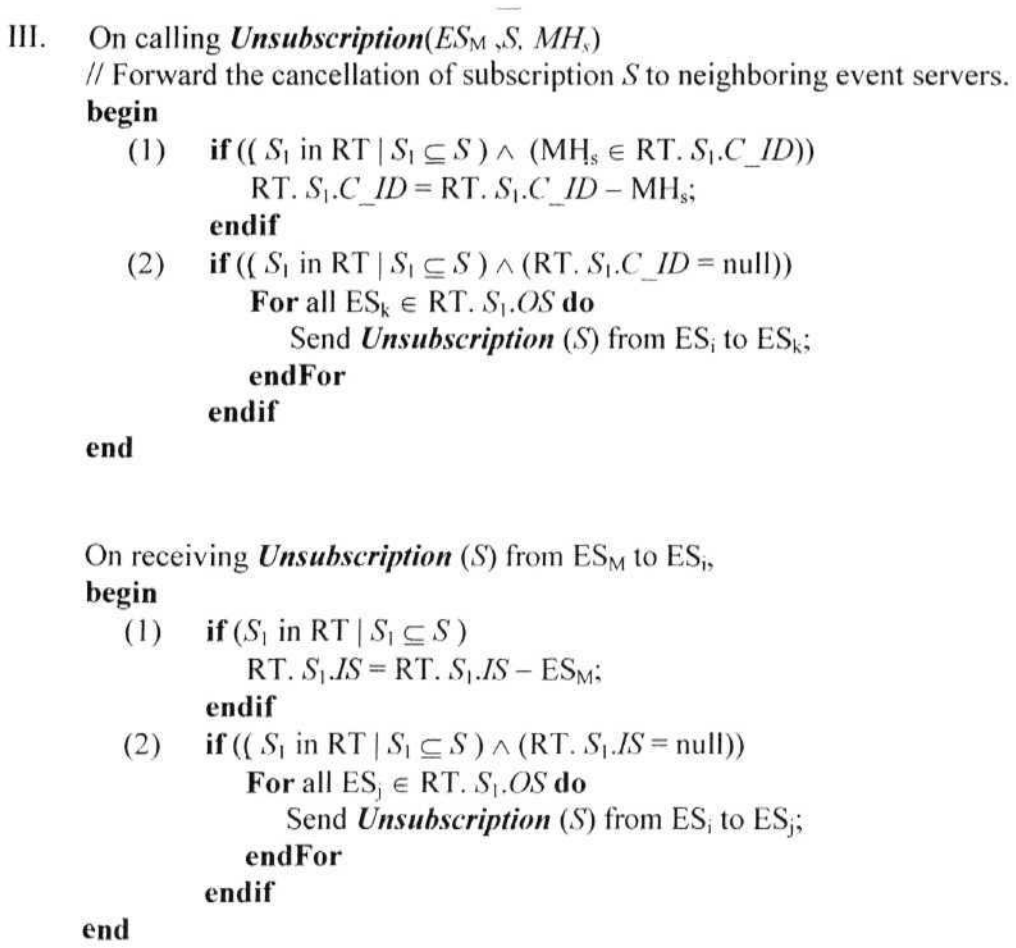
Figure 5.4: Algorithm of location-based pre-handoff (Contd.)



At each event server ES_i:

```
I.    On calling UpdateSubscription($MH_s$, $ES_M$, $S$)
      begin
        (1)   if (( $\exists S_1$ in RT | $S \subseteq S_1$ ) $\wedge$ $MH_s \in$ RT. $S_1.C\_ID$)
                  return( );
                  // RT already has $MH_s$ for subscription $S_1$ which covers $S$.
              endif
        (2)   if ($S \subseteq S_1 \wedge S_1 \subseteq S$) then /* $S$ already exists the same subscription*/
                  RT. $S_1.C\_ID$ = RT. $S_1.C\_ID \cup MH_s$;
              else
                  insert($S$); /* insert $S$ in RT as a new subscription.*/
                  RT. $S.C\_ID$ = RT. $S.C\_ID \cup MH_s$;
                  Call Cover($ES_M$, $S$);
              endif
      end

II.   On calling Cover($ES_M$, $S$)
      begin
        (1)   if ($S_1$ in RT | $S \subseteq S_1$ )
                  RT. $S.IS = ES_M$;
                  $F$ = NS - RT. $S_1.OS$ - $ES_M$;
                  //NS means all neighboring event servers of $ES_i$.
                  RT. $S.OS = F$;
              endif
        (2)   if ($S_1$ in RT | $S \subseteq S_1 \wedge S_1 \subseteq S$) then
                  RT. $S_1.IS$ = RT. $S_1.IS \cup ES_M$;
                  $F$ = NS - RT. $S_1.OS$ - $ES_M$;
                  RT. $S_1.OS$ = RT. $S_1.OS \cup F$;
              else
                  RT. $S.IS = ES_M$;
                  $F$ = NS - $ES_M$;
                  RT. $S.OS = F$;
              endif
        (c)   For all p $\in F$ do
                  Send Forward ($S$) from $ES_i$ to p; /* $ES_i$ sends $S$ to p. */
              endFor
      end

      On receiving Forward ($S$) from $ES_k$ to $ES_i$,
      begin
          Call Cover($ES_k$, $S$);
      end
```

Figure 5.5: Routing of subscription/unsubscription Messages



```
III.  On calling Unsubscription($ES_M$ ,$S$, $MH_s$)
      // Forward the cancellation of subscription $S$ to neighboring event servers.
      begin
        (1)   if (( $S_1$ in RT | $S_1 \subseteq S$ ) $\wedge$ ($MH_s \in$ RT. $S_1$.$C\_ID$))
                 RT. $S_1$.$C\_ID$ = RT. $S_1$.$C\_ID$ – $MH_s$;
              endif
        (2)   if (( $S_1$ in RT | $S_1 \subseteq S$ ) $\wedge$ (RT. $S_1$.$C\_ID$ = null))
                 For all $ES_k \in$ RT. $S_1$.$OS$ do
                    Send Unsubscription ($S$) from $ES_i$ to $ES_k$;
                 endFor
              endif
      end


      On receiving Unsubscription ($S$) from $ES_M$ to $ES_i$,
      begin
        (1)   if ($S_1$ in RT | $S_1 \subseteq S$ )
                 RT. $S_1$.$IS$ = RT. $S_1$.$IS$ – $ES_M$;
              endif
        (2)   if (( $S_1$ in RT | $S_1 \subseteq S$ ) $\wedge$ (RT. $S_1$.$IS$ = null))
                 For all $ES_j \in$ RT. $S_1$.$OS$ do
                    Send Unsubscription ($S$) from $ES_i$ to $ES_j$;
                 endFor
              endif
      end
```

Figure 5.5: Routing of subscriptions/unsubscriptions Messages (Contd.)

5.4 Analysis for Algorithm of Location-based Pre-handoff

5.4.1 Time Complexity

Time complexify is the maximum time to execute location-based pre-handoff algorithm for a mobile consumer located in a border cell. Suppose MH_s is moving in a border cell of ES,. Assume that processing time for each message is negligible. Let

N_0 = the number of border event servers adjacent to the current location of a consumer for a particular cell (i.e., card (BCT • $cell\_MH_s$ ·BES)) and

n_{sub} =the number of subscription issued by MH_s (i.e., SSUBQ_ MH_s).

Lemma 5.1. Let each message transaction between two event servers in location-based pre-handoff algorithm takes a time period t_{pre}. If the time required for pre-handoff messages is $Time_1$, then

$$Time_1 \leq 4t_{pre}$$

Proof. ES_i sends the following messages to border event servers (BCT · $cell\_MH_s$ ·BES) to which MH_s may migrate (Figure 5.4). For simplicity, we assume SMR = 0 (i.e., MH_s does not send a new subscription during executing pre-handoff) and hence ES_i need not send ***Again_Preforward*** (A2).

(1) ***Pre_handoffRequest*** (A1. (1a))

(2) ***Enable_prehandoff*** (A4 (2))

(3) ***Preforward*** (A1 (1b))

(4) ***Cancel_Preforward*** (A1 (3))

These messages are sent to N_θ event servers. Each type of messages is sent to all the border event servers N_θ in parallel during time t_{pre}. Hence the maximum time required for these messages is $4t_{pre}$.

Lemma 5.2. Let each subscribe/unsubscribe message take a time period t_{sub} to forward to the network. If the time required for subscribing of a virtual consumer is $Time_2$, then

$$Time_2 \leq 2\ n_{sub} \times t_{sub}$$

Proof. Each event server executes the following processes for establishing virtual paths of notifications and for removing virtual paths for the virtual consumer of MH_s (Figure 5.4).

(1) ***UpdateSubscription*** (A5) (i.e., to establish virtual paths) or

(2) ***Unsubscription*** (A7 (1)) (i.e., to remove virtual paths)

The processes are executed for n_{sub} subscriptions of $SSUBQ\_MH_s$. Hence the maximum time required for these two processes is $2\ n_{sub} \times t_{sub}$.

Theorem 5.1. The *time complexity* of location-based pre-handoff algorithm for a mobile consumer is $(4t_{pre} + 2n_{sub} \times t_{sub})$.

Proof. This follows due to Lemma 5.1 and 5.2.

5.4.2 Message Complexity

Message complexity is the maximum number of messages communicated between event servers in location-based pre-handoff algorithm for a mobile consumer in a border cell. Suppose MH_s is moving in a border cell of ES,.

As described in Lemma 5.1, ES, need to send four messages to event servers (BCT • *cell_ MH_s* ·BES) to which Ml Is may migrate for MH_s. Hence the total number of messages communicated between event servers is $4N_\theta$.

As described in Lemma 5.2, each event server executes two processes for establishing virtual paths and for removing virtual paths for the virtual consumer of MH_s. Hence the total number of messages sent by an event server is $2n_{sub}$. The total number of messages in the network is $2\times O(n_{sub})$ as it depends on the number of event servers to which the messages (subscriptions/unsubscriptions) forwarded in the network. Then the maximum number of messages in location-based pre-handoff is 4No + $2\times O(n_{sub})$. Hence,

Theorem 5.2. The *message complexity* of location-based pre-handoff algorithm for a mobile consumer is $4N_\theta + O(n_{sub})$.

5.4.3 Storage Complexity for Subscriptions

The main storage required in an event notification service system is its routing table (RT). When an event server receives a notification (or) subscription, it executes *covering test* (or) *matching* with subscriptions stored in RT. Suppose each event server ES_i has the following.

H_l = total number of local consumers in LA of each event server

H_m = total number of moving consumers in border cells of each event server

K = maximum number of subscriptions submitted by each mobile consumer MH_s

b_s = average size of each subscription (bits)

N_s = the number of event servers which are adjacent to location area of ESj

Lemma 5.3. Let ES, has the number of virtual consumers (VC (ES,)) for moving mobile consumers in border cells of its neighboring location areas. Then

$$VC\ (ESj) < H_m$$

Proof. Each event server has H_m moving mobile consumers in its border cells and N_s neighboring LAs. Assume that each event server sends same pre-handoff requests to every neighboring LA (i.e., moving consumers are equally located at each boundary of each LA). Hence each event server sends H_m/N_s pre-handoff requests to each neighboring LA. Since ESj receives these requests from N_s event server, the maximum total number of virtual consumers running at ESj is H_m (i.e., $(H_m/N_s) \times N_s$).

Lemma 5.4. The maximum number of subscriptions at an event server $\leq (H_l + H_m)$ x K.

Proof. If location-based pre-handoff approach is not used, ES_i only stores subscriptions of H_l local consumers. Hence the maximum number of subscriptions at ES, is (H_l x K). For location-based pre-handoff approach, ES, stores subscriptions of H_m virtual consumers (by Lemma 5.3) besides subscriptions of H_l local consumers in its routing table. Hence, the maximum number of subscriptions at an event server is $\{(H_l + H_m)$ x K$\}$.

Theorem 5.3. The *storage complexity* at an event server for subscriptions in location-based pre-handoff algorithm is $\{(H_l + H_m)$ x K x $b_s\}$ bits.

Proof. This follows due to Lemma 5.4.

5.5 Evaluation

This section first studies the cost in terms of *message complexity* of location-based pre-handoff algorithm in experiments of simulation. Location-based pre-handoff algorithm can reduce the latency of notifications received by mobile consumers at new locations as it eliminate the following delays occurred in normal handoff (i.e., without pre-handoff).

(1) The previous event server transfers subscriptions to a new event server.

(2) The new event server re-subscribe above subscriptions to establish new paths of notifications.

(3) The previous event server may transfer new notifications received for the mobile consumer before finishing handoff.

5.5.1 Simulation Setup

Experiments of this section were done on the same specifications of PC and in Java (jdk 1.4.2) as descried in Section 4.7.1. The number of event servers is 50 and the flows of messages among event servers are bidirectional. The propagation delay between two event servers is exponentially distributed random variable with a mean of 7ms. The value of every point in the graphs is an average of the results of 10 experiments performed. In experiments, we varied the number of mobile consumers and it is directly proportional to the number of virtual consumers running. In all experiments, low rates of publications are used since we want to observe the cost of pre-handoff without hitting any processing capacity constraints at event servers.

A mobile consumer in a border cell can have at most two adjacent cells in a hexagonal cell pattern and three adjacent cells in a square cell pattern. A corner border cell of a location area has two or three adjacent cells of other location areas. Hence we assume that the current event server ES, of a moving consumer must send pre-handoff requests to three event servers if the mobile consumer is in a corner border cell ES,. We also assume the number of moving mobile consumers in border cells is 50% of the total number of mobile consumers in each location area.

In experiments, the randomly generated possible locations of a mobile consumer in a location area are as follows:

(1) Border Cell (BC): Pre-handoff request is sent to one event server.

(2) Corner Border Cell (CBC): Pre-handoff request is sent to three event servers.

(3) Interior Cell (IC): It is not necessary to send pre-handoff request.

The next movement of the mobile consumer is also randomly generated. For example, a mobile consumer in a border cell (BC) can move to BC, CBC and IC. It may also stop moving in BC.

5.5.2 Simulation Results

Figure 5.6, 5.7 and 5.8 show the message complexity of location-based pre-handoff algorithm with the values of SMR = 0, 1 and 2 respectively. These experiments demonstrate that the values of CMP greatly affect the number of messages communicated in location-based pre-handoff algorithm. If the mobile consumers do not migrate to new locations predicated, location-based algorithm needs to cancel pre-handoff (i.e., to delete virtual consumers and removing virtual paths of notifications). Hence the message complexity of large CMP value is lower than those of small CMP values. In practice, the value of CMP is greater than 0.5 since the pre-handoff is only executed for mobile consumers which are moving in border cells and migrating to border cells from inner cells. Hence, most of the mobile users migrate to other location areas as predicted by using BCT (Border Cell Table). Location-based pre-handoff algorithm needs to forward more new subscriptions received from a mobile consumer MH_s to event servers which have virtual consumer of MH_s for larger SMR (Subscription Mobility Ratio) values. Hence, the graphs show that the values of (SMR) also affect on message complexity of location-based pre-handoff algorithm.

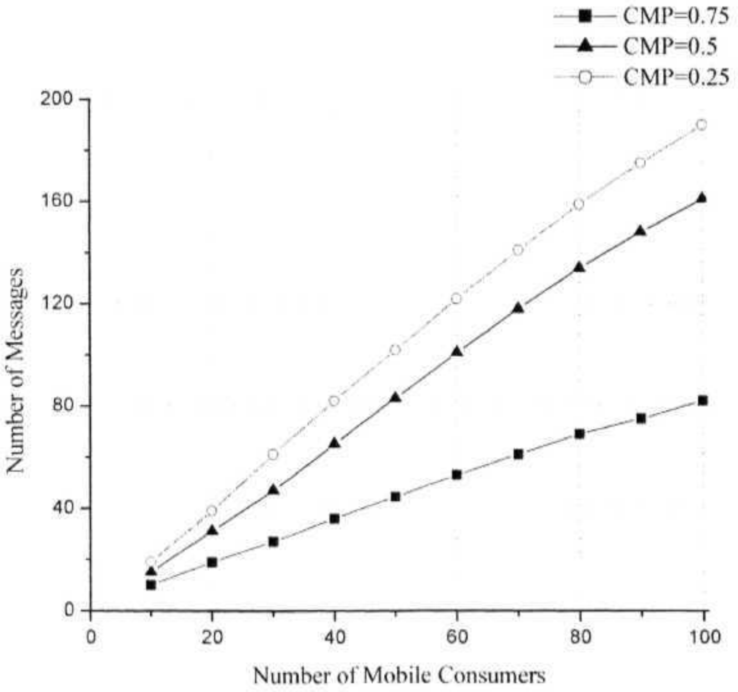


Figure 5.6: Message complexity of location-based pre-handoff with SMR=0

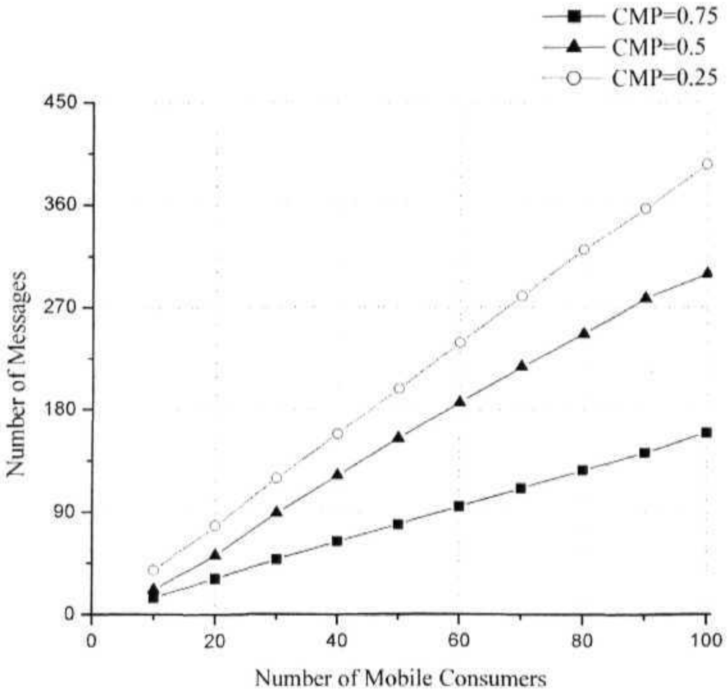


Figure 5.7: Message complexity of location-based pre-handoff with SMR=1

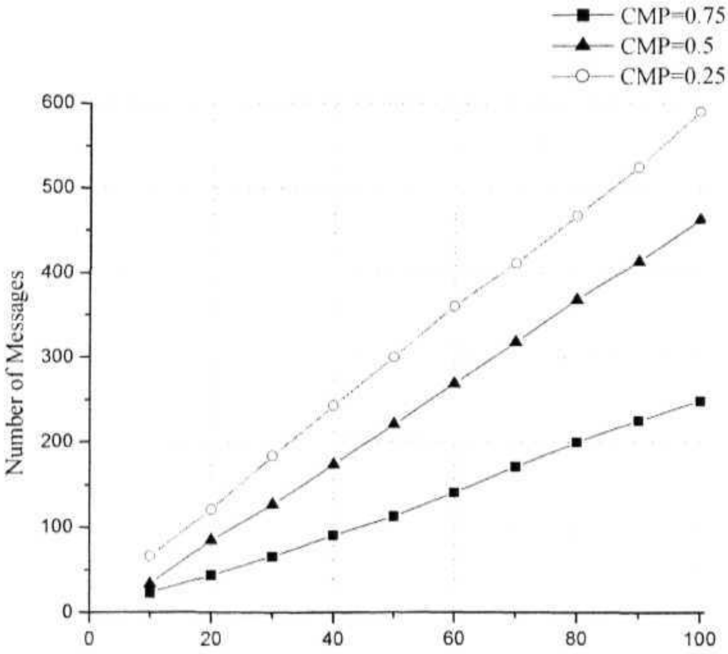


Figure 5.8: Message complexity of location-based pre-handoff with SMR=2

5.6 Summary

In this chapter we investigate on the issues involved in delivery of events as soon as mobile consumers reach at new locations. In wireless network event notification service system, mobile consumers while on move keep moving in and out of cells. These movements cause delay in delivery of events because event notification service needs to define paths from producers to mobile consumers' new location areas. Because of this delay, time constrained data items may loose their validity causing concerns and losses to mobile applications. To eliminate this delay, we propose a proactive concept called location-based pre-handoff approach that disseminates events just-in-time to all possible locations surrounding current location of a mobile consumer.

Firstly, we describe a cellular mobile architecture which is mostly used in wireless network with base stations. Then we present a regional map and location tables which are used in just-in-time delivery service. For just-in-time delivery of events to mobile users, a location-based pre-handoff algorithm is executed for mobile consumers which are moving in border cells. We present an analysis for the location-based pre-handoff algorithm to observe the time complexity, message complexity and storage complexity. Finally we demonstrate that location-based pre-handoff algorithm has a relatively low cost of message complexity with the experiments of simulation. Experiments also show that Correct Movement Prediction (CMR) greatly effect the number of messages communicated among event servers in location-based pre-handoff algorithm.

Chapter 6

Resilient Dissemination of Events in a Large-Scale Event Notification Service System[4]

In a wide-area network failures of links/nodes are not uncommon. Therefore, fault-tolerance is critical for smooth operation of a large-scale event notification service system. In this chapter, we first describe fault-tolerant region-based architecture of event notification service system for large scale network. The region-based architecture reduces the size of routing tables, thus the latency of notification delivery to consumers also decreases. We then present a fault-tolerance algorithm for region-based architecture so that event notification service system is resilient to failures of event servers and links between them, and ensures dissemination of events. We analyze the fault-tolerance algorithm to observe the time complexity, message complexity and storage complexity, and demonstrate the effectiveness of the proposed region-based architecture in experiments of simulation.

6.1 Introduction

This chapter first describes the fault-tolerant region-based architecture of event notification service (ENS) system for large scale network. Then it presents a fault-tolerance algorithm based on primary-backup replication so that event notification service system is resilient to failures of event servers and links between them, and ensures dissemination of events (notifications). In a wide-area network, failures of links/nodes are not uncommon. Event servers implement the entire functionality of an event notification service system and these are interconnected as a scalable distributed network. Therefore, fault-tolerance in event servers is critical for smooth operation of a large-scale event notification service system.

[4] This work is presented and published In *Proceedings of IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE 2005)*, Hong Kong, China, pages 502-507, March 2005 [44].

In a large-scale network, routing tables for subscriptions increase at each event server when the number of consumers becomes large. When an event is reported, each event server matches the event with all subscriptions of its routing table to forward the event to neighboring event servers. Hence, the event is matched with subscriptions at all event servers that are located along the path between the producer who publishes the event and an interested consumer. It increases matching time and causes some delay in delivering events to consumers in a large-scale network. In region-based architecture, the network of event notification service system is divided into a few big regions which have separate networks composed of event servers. Each region contains multiple event servers and has a *region leader* event server (RES) that is responsible for communicating other event servers located in different regions. Region-based architecture has benefits on latency of notification delivery as each RES forwards the notification to reach all regions without matching with subscriptions. The region-based architecture reduces the size of the routing tables, thus the latency of notification delivery to consumers also decreases.

For resilient dissemination of events, a fault-tolerance strategy is presented on the region-based architecture to minimize the impact of faults occurring in event notification service system. The fault-tolerance strategy is based on primary-backup replication models described in [55, 58, 71J. In event notification service systems, these existing replication techniques cannot be used directly as notification messages do not contain destination addresses and routes are based on contents of notifications. In our proposed fault-tolerance strategy [44], routing tables of region leaders are replicated at their backup event servers. Hence each backup event server maintains its routing table and a replicated routing table of its region leader. We term the replicated routing table as virtual routing table (VRT).

The rest of the chapter is structured as follows. The related work of fault-tolerance in event notification service systems is described in Section 6.2. The types of failures in event notification service system are identified in Section 6.3. Section 6.4 describes proposed region-based architecture which consists of three network layers, and its advantages. Section 6.5 presents the concept of fault-tolerance strategy implemented on the region-based architecture. The algorithm for fault-tolerance strategy is described in Section 6.6. Analysis of fault-tolerance algorithm is presented in Section 6.7 and Section 6.8 investigates the effects on routing table sizes of the region-based architecture in simulation.

6.2 Related Work

Herald event notification service system [8] uses many replicated servers in different locations to execute some or all work of a *rendezvous point* for scalability and fault-tolerance. It provides a degree of fault tolerance that allows clients (both publishers and subscribers) to interact with any of the replicas of a *rendezvous point* for any operations. Hermes [61] uses overlay routing network which is a logical application-level network built onto the physical network topology. Link and node failures are dealt with transparently by the overlay network. It also replicates *rendezvous* nodes so that a consumer is still able to receive events from a replica of rendezvous node when its original rendezvous node fails. In summary, both the systems [8, 61] provide fault tolerance of *rendezvous* nodes by making use of replicas. But, this strategy does not work in case of link failure and producers/consumers can only continue to publish/receive events in partitioned networks they are located.

References [17, 60] describe reconfigurations that involve the *removal* of a link and the *insertion* of a new one, thus keeping the dispatching tree connected. The approach is based on unsubscription (to remove a link) and subscription (to insert a new link), It reduces reconfiguration messages by enabling subscriptions and unsubscriptions of events only at endpoints of new link and removed link respectively. The subscriptions triggered by the appearance of a link are issued immediately, while the unsubscriptions due to a link failure/removal are issued only after a predefined *delay* to ensure insertion of the link completes before removal of the link. An approach that is resilient to join and leave of event servers is presented in [6]. Each event server maintains a list of neighbors for its neighbors. The list is updated whenever a new event server joins or leaves. After an event server leaves, new connections arc established and the state of the routing tables is kept consistent by removing the subscriptions that were hosted by the leaving event server and adding new links to maintain connectivity. Hence if an event server fails, each neighboring event server of it requests a new connection to an event server in the list. But the method follows exclusion of faulted event server and hence the event server facilities are not available to others.

The services of faulted event server can be made available by replicating its functionalities elsewhere as done in |55, 58]. But these replication techniques cannot be applied directly for fault-tolerance in ENS systems. Because here messages contain destination addresses and alternate routes are explored by routing servers. But in ENS systems, notification messages do not contain destination addresses and routes are found based on message contents.

6.3 Types of Failures

Fault-tolerance is an important feature in a large-scale event notification service system as link or node failures are expected to be more frequent in wide-area networks than in local-area networks. An event notification service system may experience the following types of failure:

(1) Link failure: This can happen by a failure at the IP routing level. An event server may also be disconnected from one or more of its neighbors due to failure of physical links among them.

(2) Event server failure: If an event server fails, none of its neighbors can contact it. One possible solution could be, event server that detects a failure of an event server establishes new paths without including the failed event server.

(3) Consumer/Producer failure: If a consumer/producer fails, it can no longer contact the local event server. After a consumer fails, the consumer's event server will unsubscribe the subscriptions of the consumer after a stipulated (expiry) time. A producer cannot publish notifications during its failure and may resume publishing notifications on its recovery.

Fault-tolerance mechanism is to be integrated to an event notification service system so that server/link failures do not affect the entire system [61]. In [72 j, it states that a *fault-tolerant system* should continue to function perhaps in a degraded form, when faced with failures. The degradation can be in performance, in functionality, or in both. Hence fault tolerance and scalability are closely related to each other.

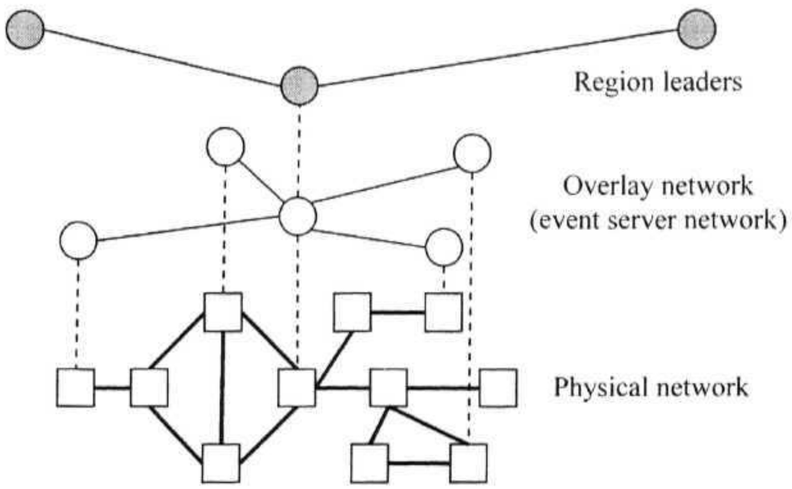


Figure 6.1: An overlay network of a region

6.4 System Model

We proposed a fault-tolerant region-based architecture for a large-scale network of event notification service system as our system model.

6.4.1 Region-based Architecture

As in most of the ENS systems [13, 17, 61, 87], we use overlay networks for interconnection of event servers. Hybrid architectures to use different architectures at different levels of network (for example, local area and wide area) for different levels of administration are introduced in [13] but it does not provide for fault-tolerance. Hermes [61] uses overlay routing network which is a logical application-level network built onto the physical network topology. Link and node failures are dealt with transparently by the overlay network. But this strategy can support fault-tolerance of link failure and producers/consumers can only continue to publish/receive events in partitioned networks they are located. We proposed a region-based architecture for our fault-tolerance strategy to minimize the impact of faults occurring in ENS systems. In region-based architecture, a large-scale network of ENS system is divided into a few regions having separate networks composed of event servers. Each region contains multiple event servers and has an event server that is responsible of interfacing to event servers located in other regions. We call such an event server as Region Leader event server (RES). In each region, a center event server (i.e., center node of a graph) is selected as RES to minimize the latency of forwarding and delivering notifications to event servers of other regions and local region respectively.

A three layered network architecture is illustrated in Figure 6.1. The bottom layer is the physical layer network with servers and links. The routing of this level only requires full unicast connectivity between nodes in the network, such as provided by IP communication on the Internet. The middle layer constitutes the overlay network and event servers cooperate with each other by forming an overlay routing network. The relation of neighboring event servers forms the overlay network of the ENS. A single hop in the overlay network may result in multiple hops in the underlying physical network topology. The overlay network has redundant paths between event servers so that a different path can be chosen when a given physical link is down. The top layer is RES network and it is built on top of an overlay network. As described above, each region contains one RES. RESs can be linked together forming any interconnection topology. Other event servers of a region can be interconnected in any topology within a region depending on the characteristics of the applications and a choice of interconnection topology in one region could be independent of that in other regions. We call the above three layered network architecture as region-based architecture and advantages of region-based architecture are described in the following Section 6.4.2. In RES network of top layer, each RES forwards notifications published from its region to reach all RESs without matching with subscriptions. We can differentiate the process of dissemination of notifications into 2-tiers: inter-region for top layer and intra-region for middle layer as indicated below.

Inter-regional Communication

Notification forwarding: When an event server receives a notification from its local producer, it sends the notification to its RES. On receiving a new notification, each RES forwards the same to its neighboring RESs, if the notification has not been forwarded earlier. A RES then matches the notification against subscriptions of its routing table to forward the same to neighboring event servers. In our scheme, notifications generated from a region are forwarded to all other RESs. This eliminates the need for forwarding subscription to other regions.

Intra-regional Communication

Subscription forwarding: When an event server receives a subscription from a consumer or its neighboring event servers, it forwards the subscription to neighboring event servers. Subscriptions of consumers of a region are not forwarded to other regions as region leader of this region receive all notifications published from other regions.

Notification forwarding: On receiving a notification from a neighboring RES or a neighboring event server in its region, each RES matches the notification with subscriptions of its routing table and forwards them to its neighboring event servers only if at least an interested consumer exists in its region.

6.4.2 Advantages

The proposed region-based architecture has the following advantages.

(1) Improves reliability. If a link or an event server fails in a region, other regions are not affected. Notifications can reach to all regions if RESs do not fail and overlay network remains minimally connected. Additional event servers can be added in order to improve performance or reliability of the system in a region without affecting performance and functionality of other regions.

(2) Reduces the size of routing tables. In large-scale network, subscriptions of a routing table at each event server increases when the number of consumers becomes large. As an event server of a region maintains subscriptions of regional consumers, the size of its routing table is greatly reduced.

(3) Reduces matching overhead. In existing event notification service systems [13, 18, 79, 87], when an event is reported, each event server matches the same with all subscriptions stored in its own routing table to forward the notification. In these systems, an event server stores subscriptions of entire ENS. Hence, the event is matched with subscriptions at all event servers which are located along

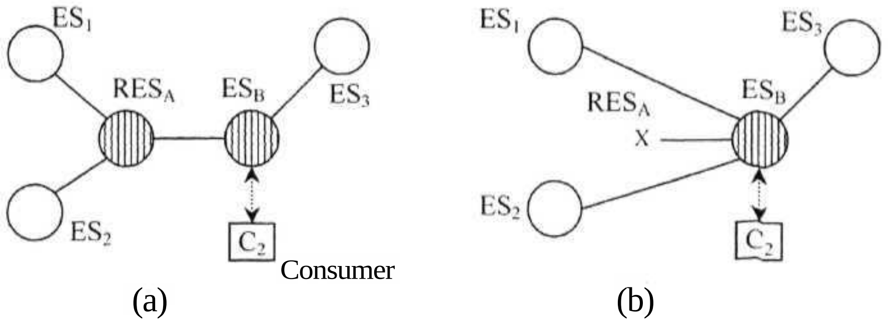
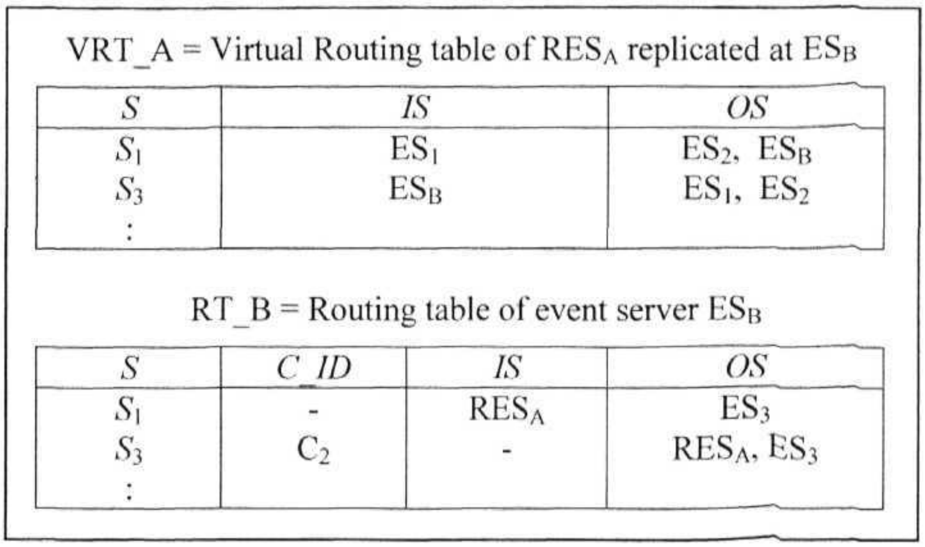


Figure 6.2: Managing failure of event server RESA

the paths between the producer which publishes the event and all the interested consumers. In our architecture, RES forwards notifications to reach all RESs. Hence it reduces the time and with no cost of matching at RESs as notifications published from a region would reach any region without explicit matching process. Further event servers in a region only matches with less number of subscriptions, thus cost of matching is also greatly reduced.

6.5 Fault-tolerance Strategy

We propose a fault-tolerance strategy for our region-based architecture descried in Figure 6.1. Our fault-tolerance strategy is based on primary-backup replication model as described in [55, 58, 71]. Although every event server can be replicated in the system, it is costly and hence replication is done for RESs of top layer in our fault-tolerance strategy. Routing table (RT) of each RES is replicated at a backup event server in its region. We choose a neighboring event server of RES as the backup event sever. Details of RT are described in Section 3.5.1. In fault-tolerance strategy, if a region leader event server RES_A fails, its neighboring RESs connect to backup event sever of RES_A. Hence the RES network of top layer is fault-tolerant to failures of RESs and inter-regional communication described in Section 6.4.1 is maintained among RESs. A backup of a routing table is called as *virtual routing table* (VRT) as it is only used in case of failure of RES. A routing table of an RES and its VRT replicated at backup event server are necessary to be consistent to avoid loss or duplication of notifications.



VRT_A = Virtual Routing table of RES_A replicated at ES_B

S	IS	OS
S_1	ES_1	ES_2, ES_B
S_3	ES_B	ES_1, ES_2
:		

RT_B = Routing table of event server ES_B

S	C_ID	IS	OS
S_1	-	RES_A	ES_3
S_3	C_2	-	RES_A, ES_3
:			

Figure 6.3: Representing routing tables at ESB

We assume that failure of a RES can be eventually detected by its neighboring RES. For example, a *heartbeat protocol* ensures that the neighboring event servers are reachable and alive [61]. To detect a link or an event server failure, a *handshaking* protocol [72] can also be used. In *handshaking* protocol, two event servers send each other an *I-am-up* message at fixed intervals. If an event server does not receive this message within a predetermined time period, it can assume that another event server has failed, that the link between them has failed, or the message has been lost.

Suppose ES_B is a backup event server of region leader RESA as shown in Figure 6.2. In Figure 6.2 (a) as RES_A does not fail, neighboring event servers (including neighboring RESs) connect to RES_A. In figure 6.2 (b), when RES_A fails neighboring event servers (including neighboring RESs) connect to ESB and ES_B executes the processes of RESA. For fault-tolerance strategy, ES_B maintains its routing table RT_B and RES_A's virtual routing table VRT_A. For example, RT_B and VRT_A are shown in Figure 6.3.

RES_A has not failed (normal condition): RESA enables its RT_A and all neighbors of RES_A are connected to it. ES_B disable VRT_A which is backup routing table of RT_A and it only executes its own processes with RT_B. Whenever a subscription is inserted in RT_A, RES_A sends the subscription to ES_B to update VRT_A. To maintain consistency of RT_A and VRT_A, RES_A updates on demand (i.e., whenever a new subscription is

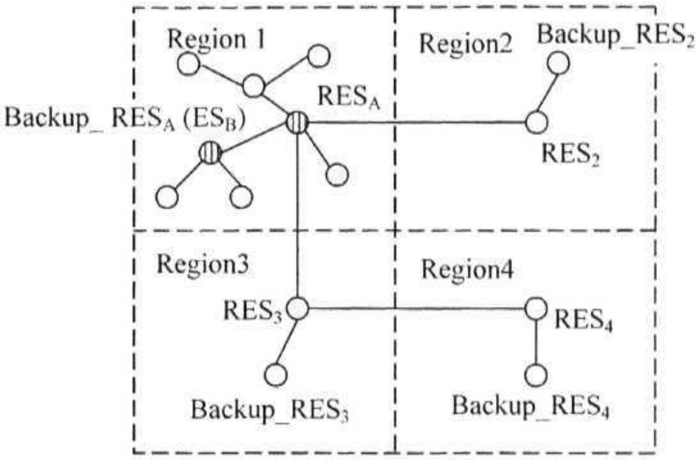


Figure 6.4: Replication of RESs in multiple regions

inserted in its routing table) to backup event server ES$_B$. Requirement for updating routing table on demand is described in Section 6.7.4.

RES$_A$ has failed: ES$_B$ enables VRT_A to execute the functions of RES$_A$. Neighboring event servers of RES$_A$ connect to ES$_B$ instead of RESA. When ES$_B$ receives a subscription/notification from neighboring event servers of ESB, ESB first executes (i.e., covering test or matching) them with RT_B. If the subscription/notification needs to be forwarded to RESA, ES$_B$ then execute it with VRT_A. For example if a notification N matches S_1 of RT_B and N needs to be forwarded to RES$_A$ (RES$_A$ $\in$ RT_B.S_1.IS),N is then matched with subscriptions of VRT_A. Similarly, if subscriptions or notifications come from neighboring event servers of RES$_A$, ES$_B$ first executes them with VRT_A then with RT_B. When neighboring event servers of RESA detects RESA has recovered, they try to connect to RESA and inform ES$_B$ to update RT_A with VRT_A.

6.6 FTAENS: Fault-Tolerance Algorithms for ENS

In this section we present algorithm for the proposed fault-tolerance strategy. A region leader (RES) in a region is a primary event server and it is allowed to interact with RESs of other regions. Each RES has a backup event server which is one of its neighboring event servers in the same region. The difference between fault-tolerance strategy for an event notification service system and existing primary-backup approaches such as [55, 58] is that backup event server in event notification service system executes its own processes although its RES does not fails as described in Section 6.5. Hence, in a region each backup event server needs to maintain both of it own routing table (RT) and virtual routing table (VRT) of its RES. Suppose RESA is a region leader and ES_B is a backup event server of RESA. As shown in Figure 6.4, RESA is interconnected with neighboring RESs of other regions. The data structures used in the algorithm are as follows:

- RT_A = Routing table of event server RESA
- VRT_A = Virtual routing table of RESA
- S_e = A set of event servers which has failed to connect
- RESNeigbor (RES,) = Neighboring RESs of RES,
- LocalPro (ESj) = Local producers of ESj
- LocalCon (ES,) = Local consumers of ES,
- Neighbor (ES,) = Neighboring event servers of ESj
- LocalES (RES,) = Event servers in the region of RESj

6.6.1 Processes of a Region Leader

A region leader RESA needs to execute the subscription and notification processes as described in Figure 6.5. For simplicity, we assume the region leader does not have local clients. (i.e., consumers and producers).

Subscription processes: RESA can receive a subscription *S* from neighboring event servers (Neighbor (RESA))

(1) RESA is in active condition (Algorithm 1. A1S1)

(2) RESA is in suspect condition (Algorithm 1. A1S3)

In case 1, RES_A inserts *S* in RT_A and the forwards *S* to neighboring event servers after checking *S* with subscriptions of RT_A. Whenever RES_A inserts a new subscription *S* in its routing table RT_A, it sends ***update** (S)* to ES_B to update VRT_A.

In case 2, RESA is not completely fails and it informs to the sender to connect to backup event server ESB.

Notification processes: RESA can receive a notification *N* from

(1) Neighboring event servers (Neighbor (RESA))

(2) Neighboring RESs (RESNeigbor (RES_A)) and

(3) Event servers in this region (LocalES (RESA)). These event servers directly send notifications received from their local producers to RES. RES forwards the notifications to its neighboring RESs.

In case 1, RESA matches *N* with subscriptions of RT_A and forwards *N* to its neighbors (Algorithm 1.A1N1).

In case 2, RESA

(a) Sends TV to all neighboring RESs except the RES which sends *N* (Algorithm 1. A1N2).

(b) Matches *N* with subscriptions of RT_A and forwards *N* to its neighboring event servers (Algorithm 1. A1N2).

In case 3, RESA forwards *N* to all neighboring RESs (Algorithm 1. A1N3).

6.6.2 Processes of a Backup Event Server

If region leader RESA does not fail, its backup event server ES_B will only execute its own processes like other event servers. If RESA fails, ES_B needs to execute its processes and processes of RES_A. The algorithm executed at ES_B is described in Figure 6.6.

Subscription processes: While RES_A fails, ESB can receive a subscription *S* from

(1) Neighboring event servers of itself (Neighbor (ESB))

(2) Local consumers (LocalCon (ESB))

(3) Neighboring event servers of RES_A (Neighbor (RESA))

In case 1 and 2, ES_B first checks *S* against subscriptions of RT_B and forwards *S* to related neighboring event servers of ES_B. If RES_A is one of neighboring event servers to forward *S*, ES_B checks *S* against subscriptions of VRT_A and forwards *S* to neighboring event servers of RES_A (Algorithm 2. A2S3).

In case 3, ES_B first checks *S* against subscriptions of VRT_A and forwards *S* to resulting neighboring event servers of RES_A. Then ES_B checks *S* against subscriptions of RT_B if ESB is one of resulting neighboring event servers (Algorithm 2. A2S3).

Notification processes; ES_B can receive a notification *N* from

(1) Neighboring event servers of itself (Neighbor (ESB))

(2) Local producers (LocalPro (ESB))

(3) Neighboring event servers of RES_A (Neighbor (RESA))

(4) Neighboring RESs (RESNeigbor (RES_A))

(5) Event servers in this region (LocalES (RES_A)).

(LocalES (RES_A) is equal to LocalES (ESB) because they are in a region.)

In case 1 and 2, ES_B first matches *N* with subscriptions of R T_ Band forwards *N* to its neighboring event servers or delivers *N* to consumers. If TV is necessary to be forwarded to RES_A, ES_B matches TV with subscriptions of VRT_ A and forwards *N* to neighboring event servers of RES_A (Algorithm 2. A2N3).

In case 3, it is similar to case 1. But ES_B first matches *N* with VRT_A and then RT_B (Algorithm 2. A2N3).

In case 4, ES_B

(a) Sends *N* to its all neighboring RESs of RESA except the RES which sends TV (Algorithm 2. A2N4).

(b) Matches first TV with subscriptions of VRT_ A and forwards *N* to resulting neighboring event servers of RES_A. If ES_B is one of resulting neighbors to forward *N*, ES_B matches *N* with subscriptions of RT_ B and forwards *N* to neighboring event servers of ES_B (Algorithm 2. A2N4).

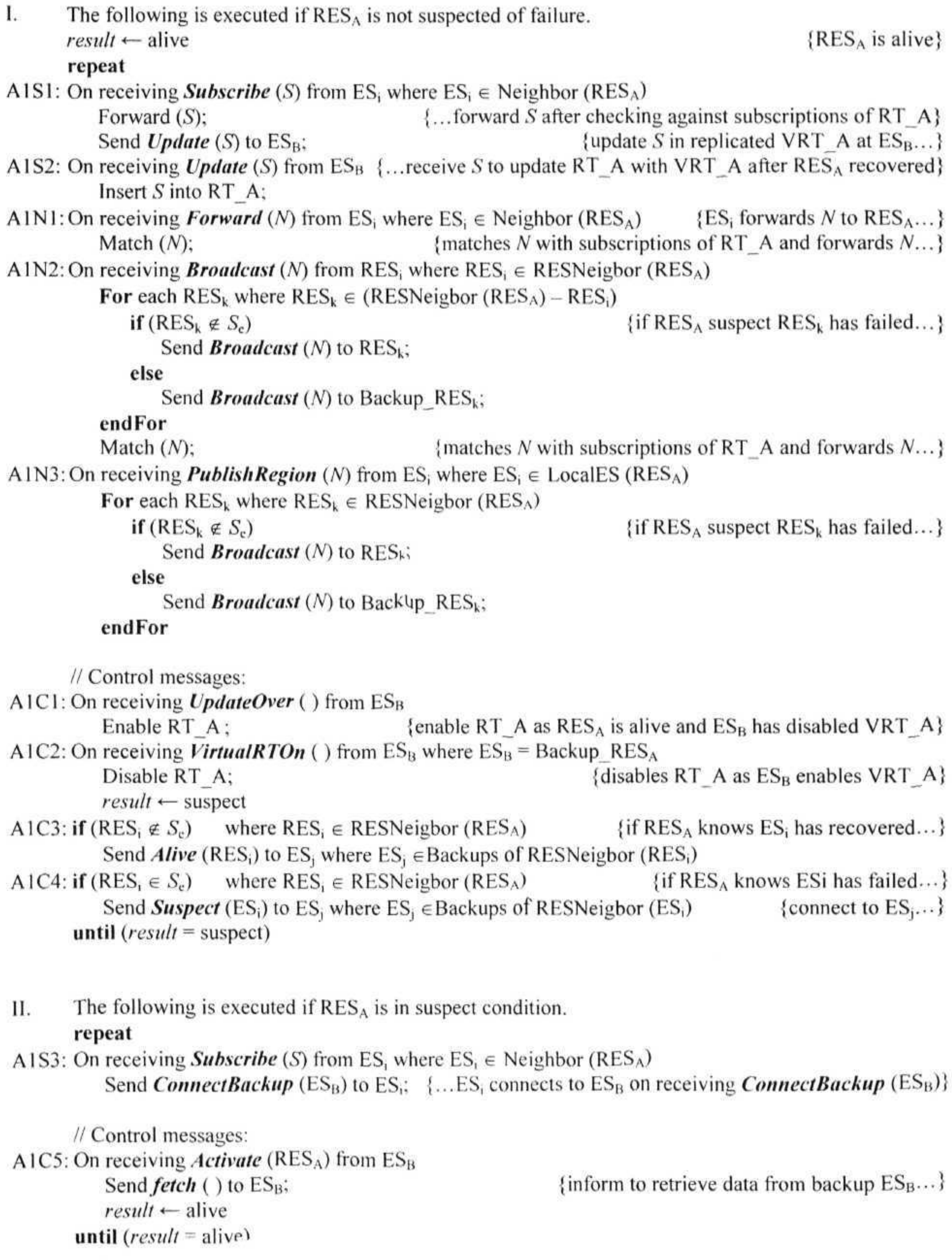
In case 5, ES_B forwards *N* to all neighboring RESs (Algorithm 2. A2N5).

6.6.3 Control Messages between RES and its Backup Event Server

Every region leader (RES) detects the failure of neighboring RESs and links between them periodically. Each RES knows the backup event servers of its neighboring RESs. When a region leader RES$_A$ fails, its backup event server ES$_B$ executes the functions of RES$_A$. Hence it is necessary the following control messages for consistency.

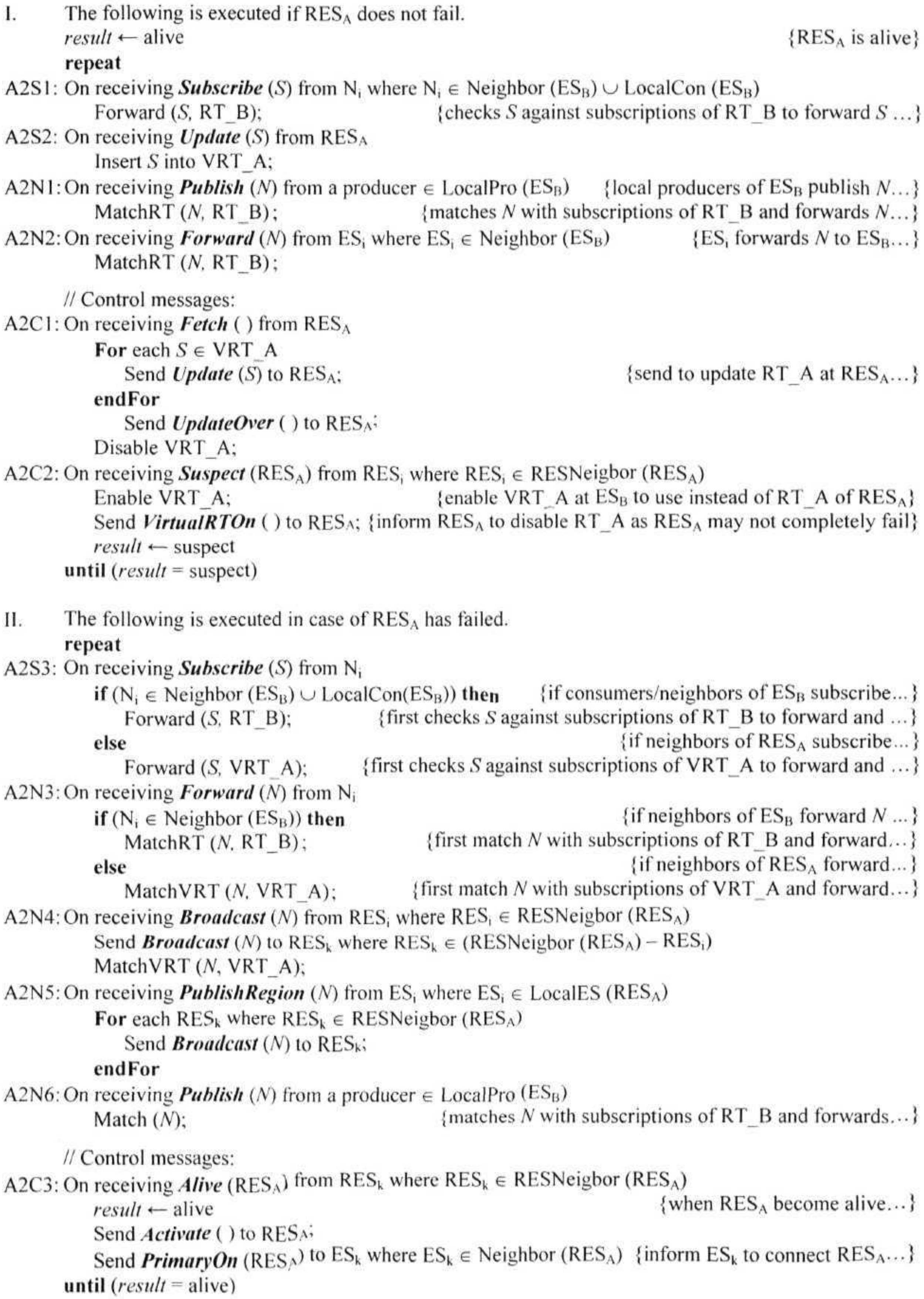
(1) **RES$_A$ fails (or is suspected of failure):** If neighboring RESs of RESA suspect that RESA has failed, it sends *Suspect* (RESA) to ES$_B$ and establishes connection with ES$_B$ (Algorithm 1. A1C4). When ES$_B$ receives ***Suspect*** (RESA), it enables VRT_A. (Algorithm 2. A2C2). For consistency, it is necessary to enable only one of RT_A (at RES$_A$) and VRT_A (at ES$_B$). Hence ES$_B$ sends ***VirtualRTOn*** ( ) to RESA (Algorithm 2. A2C2) in case of RESA may not failed completely. On receiving ***VirtualRTOn*** ( ), RES$_A$ disables RT_A (Algorithm 1. A1C2). It ensures that only one dissemination tree is active for consistency.

(2) **RES$_A$ has recovered:** If neighboring RESs of RESA detects RES$_A$ has recovered, it sends ***Alive*** (RES$_A$) to ESB (AS RESA is one of neighboring RES for some RESs, it also executes Algorithm 1. A1C3). When ES$_B$ receives ***Alive*** (RESA) from neighboring RESs of RESA, it sends ***PrimaryOn*** (RESA) to all neighbors of RES$_A$ to connect to RES$_A$ and sends *Activate* ( ) to RES$_A$ (Algorithm 2. A2C3). On receiving ***PrimaryOn*** (RES$_A$), all neighbors of RESA reconnect to RESA instead of ESB. When RESA receives ***Activate*** ( ), it send ***Fetch*** ( ) to ES$_B$ to retrieve subscriptions which ES$_B$ inserted in VRT_A during failure of RESA (Algorithm 1. A1C5). On receiving ***Fetch*** ( ), ES$_B$ sends ***Update*** *(S)* to RES$_A$ for each subscription *S* stored in VRT_A to update RT_A at RESA. After sending all subscriptions, ES$_B$ sends ***UpdateOver*** ( ) to RES$_A$ and then disables VRT_A (Algorithm 2. A2C1). On receiving ***UpdateOver*** (X RES$_A$ enables RT_A (Algorithm1. A1Cl) and executes its functions of a region leader.



```
I.      The following is executed if $RES_A$ is not suspected of failure.
        result ← alive                                                        {$RES_A$ is alive}
        repeat
A1S1: On receiving Subscribe (S) from $ES_i$ where $ES_i \in$ Neighbor ($RES_A$)
            Forward (S);                          {...forward S after checking against subscriptions of RT_A}
            Send Update (S) to $ES_B$;                         {update S in replicated VRT_A at $ES_B$...}
A1S2: On receiving Update (S) from $ES_B$  {...receive S to update RT_A with VRT_A after $RES_A$ recovered}
            Insert S into RT_A;
A1N1: On receiving Forward (N) from $ES_i$ where $ES_i \in$ Neighbor ($RES_A$)        {$ES_i$ forwards N to $RES_A$...}
            Match (N);                              {matches N with subscriptions of RT_A and forwards N...}
A1N2: On receiving Broadcast (N) from $RES_i$ where $RES_i \in$ RESNeigbor ($RES_A$)
            For each $RES_k$ where $RES_k \in$ (RESNeigbor ($RES_A$) – $RES_i$)
                if ($RES_k \notin S_c$)                              {if $RES_A$ suspect $RES_k$ has failed...}
                    Send Broadcast (N) to $RES_k$;
                else
                    Send Broadcast (N) to Backup_$RES_k$;
            endFor
            Match (N);                              {matches N with subscriptions of RT_A and forwards N...}
A1N3: On receiving PublishRegion (N) from $ES_i$ where $ES_i \in$ LocalES ($RES_A$)
            For each $RES_k$ where $RES_k \in$ RESNeigbor ($RES_A$)
                if ($RES_k \notin S_c$)                              {if $RES_A$ suspect $RES_k$ has failed...}
                    Send Broadcast (N) to $RES_k$;
                else
                    Send Broadcast (N) to Backup_$RES_k$;
            endFor

        // Control messages:
A1C1: On receiving UpdateOver ( ) from $ES_B$
            Enable RT_A ;                     {enable RT_A as $RES_A$ is alive and $ES_B$ has disabled VRT_A}
A1C2: On receiving VirtualRTOn ( ) from $ES_B$ where $ES_B$ = Backup_$RES_A$
            Disable RT_A;                                   {disables RT_A as $ES_B$ enables VRT_A}
            result ← suspect
A1C3: if ($RES_i \notin S_c$)     where $RES_i \in$ RESNeigbor ($RES_A$)            {if $RES_A$ knows $ES_i$ has recovered...}
            Send Alive ($RES_i$) to $ES_j$ where $ES_j \in$Backups of RESNeigbor ($RES_i$)
A1C4: if ($RES_i \in S_c$)     where $RES_i \in$ RESNeigbor ($RES_A$)                {if $RES_A$ knows ESi has failed...}
            Send Suspect ($ES_i$) to $ES_j$ where $ES_j \in$Backups of RESNeigbor ($ES_i$)          {connect to $ES_j$...}
        until (result = suspect)


II.     The following is executed if $RES_A$ is in suspect condition.
        repeat
A1S3: On receiving Subscribe (S) from $ES_i$ where $ES_i \in$ Neighbor ($RES_A$)
            Send ConnectBackup ($ES_B$) to $ES_i$;   {...$ES_i$ connects to $ES_B$ on receiving ConnectBackup ($ES_B$)}

        // Control messages:
A1C5: On receiving Activate ($RES_A$) from $ES_B$
            Send fetch ( ) to $ES_B$;                      {inform to retrieve data from backup $ES_B$...}
            result ← alive
        until (result = alive)
```

Figure 6.5: FTAENS: Algorithm 1 executed at region leader RESA



```
I.     The following is executed if RES$_A$ does not fail.
       *result* ← alive                                                          {RES$_A$ is alive}
       **repeat**
A2S1: On receiving ***Subscribe*** (*S*) from N$_i$ where N$_i$ $\in$ Neighbor (ES$_B$) $\cup$ LocalCon (ES$_B$)
          Forward (*S*, RT_B);                    {checks *S* against subscriptions of RT_B to forward *S* ...}
A2S2: On receiving ***Update*** (*S*) from RES$_A$
          Insert *S* into VRT_A;
A2N1: On receiving ***Publish*** (*N*) from a producer $\in$ LocalPro (ES$_B$)     {local producers of ES$_B$ publish *N*...}
          MatchRT (*N*, RT_B);                    {matches *N* with subscriptions of RT_B and forwards *N*...}
A2N2: On receiving ***Forward*** (*N*) from ES$_i$ where ES$_i$ $\in$ Neighbor (ES$_B$)          {ES$_i$ forwards *N* to ES$_B$...}
          MatchRT (*N*, RT_B);

       // Control messages:
A2C1: On receiving ***Fetch*** ( ) from RES$_A$
          **For** each *S* $\in$ VRT_A
             Send ***Update*** (*S*) to RES$_A$;                          {send to update RT_A at RES$_A$...}
          **endFor**
             Send ***UpdateOver*** ( ) to RES$_A$;
          Disable VRT_A;
A2C2: On receiving ***Suspect*** (RES$_A$) from RES$_i$ where RES$_i$ $\in$ RESNeigbor (RES$_A$)
          Enable VRT_A;                           {enable VRT_A at ES$_B$ to use instead of RT_A of RES$_A$}
          Send ***VirtualRTOn*** ( ) to RES$_A$; {inform RES$_A$ to disable RT_A as RES$_A$ may not completely fail}
          *result* ← suspect
       **until** (*result* = suspect)

II.    The following is executed in case of RES$_A$ has failed.
       **repeat**
A2S3: On receiving ***Subscribe*** (*S*) from N$_i$
          **if** (N$_i$ $\in$ Neighbor (ES$_B$) $\cup$ LocalCon(ES$_B$)) **then**      {if consumers/neighbors of ES$_B$ subscribe...}
             Forward (*S*, RT_B);            {first checks *S* against subscriptions of RT_B to forward and ...}
          **else**                                                       {if neighbors of RES$_A$ subscribe...}
             Forward (*S*, VRT_A);        {first checks *S* against subscriptions of VRT_A to forward and ...}
A2N3: On receiving ***Forward*** (*N*) from N$_i$
          **if** (N$_i$ $\in$ Neighbor (ES$_B$)) **then**                          {if neighbors of ES$_B$ forward *N* ...}
             MatchRT (*N*, RT_B);              {first match *N* with subscriptions of RT_B and forward...}
          **else**                                                          {if neighbors of RES$_A$ forward...}
             MatchVRT (*N*, VRT_A);          {first match *N* with subscriptions of VRT_A and forward...}
A2N4: On receiving ***Broadcast*** (*N*) from RES$_i$ where RES$_i$ $\in$ RESNeigbor (RES$_A$)
          Send ***Broadcast*** (*N*) to RES$_k$ where RES$_k$ $\in$ (RESNeigbor (RES$_A$) – RES$_i$)
          MatchVRT (*N*, VRT_A);
A2N5: On receiving ***PublishRegion*** (*N*) from ES$_i$ where ES$_i$ $\in$ LocalES (RES$_A$)
          **For** each RES$_k$ where RES$_k$ $\in$ RESNeigbor (RES$_A$)
             Send ***Broadcast*** (*N*) to RES$_k$;
          **endFor**
A2N6: On receiving ***Publish*** (*N*) from a producer $\in$ LocalPro (ES$_B$)
          Match (*N*);                            {matches *N* with subscriptions of RT_B and forwards...}

       // Control messages:
A2C3: On receiving ***Alive*** (RES$_A$) from RES$_k$ where RES$_k$ $\in$ RESNeigbor (RES$_A$)
          *result* ← alive                                                  {when RES$_A$ become alive...}
          Send ***Activate*** ( ) to RES$_A$;
          Send ***PrimaryOn*** (RES$_A$) to ES$_k$ where ES$_k$ $\in$ Neighbor (RES$_A$)  {inform ES$_k$ to connect RES$_A$...}
       **until** (*result* = alive)
```

Figure 6.6: FTAENS: Algorithm 2 executed at backup event server

```
I.      The following is executed if $RES_A$ does not fail.
        result ← alive                                                              {$RES_A$ is alive}
        repeat
A2S1: On receiving Subscribe (S) from $N_i$ where $N_i \in$ Neighbor ($ES_B$) ∪ LocalCon ($ES_B$)
            Forward (S, RT_B);                        {checks S against subscriptions of RT_B to forward S …}
A2S2: On receiving Update (S) from $RES_A$
            Insert S into VRT_A;
A2N1: On receiving Publish (N) from a producer ∈ LocalPro ($ES_B$)     {local producers of $ES_B$ publish N…}
            MatchRT (N, RT_B);                      {matches N with subscriptions of RT_B and forwards N…}
A2N2: On receiving Forward (N) from $ES_i$ where $ES_i \in$ Neighbor ($ES_B$)          {$ES_i$ forwards N to $ES_B$…}
            MatchRT (N, RT_B);

        // Control messages:
A2C1: On receiving Fetch ( ) from $RES_A$
            For each S ∈ VRT_A
                Send Update (S) to $RES_A$;                                  {send to update RT_A at $RES_A$…}
            endFor
                Send UpdateOver ( ) to $RES_A$;
            Disable VRT_A;
A2C2: On receiving Suspect ($RES_A$) from $RES_i$ where $RES_i \in$ RESNeigbor ($RES_A$)
            Enable VRT_A;                               {enable VRT_A at $ES_B$ to use instead of RT_A of $RES_A$}
            Send VirtualRTOn ( ) to $RES_A$; {inform $RES_A$ to disable RT_A as $RES_A$ may not completely fail}
            result ← suspect
        until (result = suspect)

II.     The following is executed in case of $RES_A$ has failed.
        repeat
A2S3: On receiving Subscribe (S) from $N_i$
            if ($N_i \in$ Neighbor ($ES_B$) ∪ LocalCon($ES_B$)) then      {if consumers/neighbors of $ES_B$ subscribe...}
                Forward (S, RT_B);                {first checks S against subscriptions of RT_B to forward and …}
            else                                                                   {if neighbors of $RES_A$ subscribe...}
                Forward (S, VRT_A);            {first checks S against subscriptions of VRT_A to forward and …}
A2N3: On receiving Forward (N) from $N_i$
            if ($N_i \in$ Neighbor ($ES_B$)) then                                 {if neighbors of $ES_B$ forward N ...}
                MatchRT (N, RT_B);                   {first match N with subscriptions of RT_B and forward…}
            else                                                                      {if neighbors of $RES_A$ forward...}
                MatchVRT (N, VRT_A);            {first match N with subscriptions of VRT_A and forward…}
A2N4: On receiving Broadcast (N) from $RES_i$ where $RES_i \in$ RESNeigbor ($RES_A$)
            Send Broadcast (N) to $RES_k$ where $RES_k \in$ (RESNeigbor ($RES_A$) – $RES_i$)
            MatchVRT (N, VRT_A);
A2N5: On receiving PublishRegion (N) from $ES_i$ where $ES_i \in$ LocalES ($RES_A$)
            For each $RES_k$ where $RES_k \in$ RESNeigbor ($RES_A$)
                Send Broadcast (N) to $RES_k$;
            endFor
A2N6: On receiving Publish (N) from a producer ∈ LocalPro ($ES_B$)
            Match (N);                                  {matches N with subscriptions of RT_B and forwards…}

        // Control messages:
A2C3: On receiving Alive ($RES_A$) from $RES_k$ where $RES_k \in$ RESNeigbor ($RES_A$)
            result ← alive                                                         {when $RES_A$ become alive…}
            Send Activate ( ) to $RES_A$;
            Send PrimaryOn ($RES_A$) to $ES_k$ where $ES_k \in$ Neighbor ($RES_A$)  {inform $ES_k$ to connect $RES_A$…}
        until (result = alive)
```

Figure 6.6: FTAENS: Algorithm 2 executed at backup event server

6.7 Analysis for Fault-tolerance Algorithm

6.7.1 Time Complexity

We study two types of *time complexity* for fault-tolerance algorithm: (1) *time complexity* for starting processes of RES_A at ES_B when RES_A fails, and (2) *time complexity* for executing processes of RES_A itself when RES_A recovers. Let each message transaction between any two event servers (including RESs) in fault-tolerance algorithm takes a time period t_{com}. Assume that processing time for each message is negligible.

Lemma 6.1. Let time period of detecting the failure of RES by its neighboring RESs take t_{det}. It is a predetermined fixed time interval of *heartbeat protocol* or *handshaking protocol* as described in Section 6.5.1. After RES_A fails,

If the time required to start processes of RES_A at ES_B is $Time_1$, then

$$Time_1 \leq t_{det} + 2t_{com}$$

Proof. Fault-tolerance algorithm executes the following messages when RES_A fails.

(1) ***Suspect*** (RES_A) from neighboring RESs to ES_B (Algorithm 1. A1C4)

(2) ***VirtualRTOn*** ( ) from ES_B to RES_A (Algorithm 2. A2C2)

The maximum time required for these messages is $2t_{com}$. The time period required to know the failure of RES_A is t_{det}. Hence the maximum time to start processes of RES_A at ES_B is $t_{det} + 2t_{com}$.

Lemma 6.2. Let the number of subscriptions stored in VRT_A at ES_B during failure of RES_A is n_{subf}. When RES_A recovers,

If the time required to restart processes of RES_A at RES_A is $Time_2$, then

$$Time_2 \leq (5 + n_{subf}) \times t_{com}$$

Proof. Fault-tolerance algorithm executes the following messages when RES_A has recovers.

(1) ***Alive*** (RES_A) (Algorithm 1. A1C3)

(2) ***PrimaryOn*** (RES_A) (Algorithm 2. A2C3)

(3) *Activate* () (Algorithm 2. A2C3)

(4) ***Fetch*** () (Algorithm 1. A1C5)

(5) ***Update*** *(S)* (Algorithm 2. A2C1)

(6) ***UpdateOver*** *{)* (Algorithm 2. A2C1)

Each message takes time period t_{com} except ***Update*** (S) message. ***Update*** (S) depends on n_{subf}. Hence the maximum time required for these messages is $(5 + n_{subf}) \times t_{com}$.

Theorem 6.1. The *time complexity* of the fault-tolerance algorithm is less than equal to $\{t_{det} + (7 + n_{subf}) \times t_{com}\}$.

Proof. This follows due to Lemma 6.1 and 6.2.

6.7.2 Message Complexity

The maximum number of messages happens at backup event server of RESA during failure of RESA. Hence we investigate the *message complexity* which is the maximum number of messages communicated between backup event server ES_B and others. We investigate *message complexity* for notification and subscriptions, as the number of messages communicated depends on the type of messages. Let

N_e = the maximum number of neighboring event servers of RESA or ESB

N_{res} = the number of neighboring RESs of RESA

Lemma 6.3. On receiving a notification,

The maximum number of messages of fault-tolerance algorithm at ESB $\leq 2N_e + N_{res}$

Proof. On receiving a notification N, the number of messages at ES_B, M_{max} is as follows. The details of notification processes at ES_B are described in Section 6.6.2.

(1) If *N* is received from neighboring RESs of RES_A, M_{max} is $\{2N_e + (N_{res}-1)\}$.

(2) If *N* is received from neighboring event servers of RES_A or ES_B, M_{max} is $\{2N_e - 1\}$.

(3) If *N* is received from a local producer of ESB, M_{max} is $\{2N_e + N_{res}\}$.

(4) If *N* is received from one event servers of this region, M_{max} is N_{res}.

Hence on receiving a notification, the maximum number of messages in fault-tolerance algorithm is $2N_e + N_{res}$.

Lemma 6.4. On receiving a subscription,

The maximum number of messages of fault-tolerance algorithm at $ES_B \leq 2N_e$

Proof. On receiving a subscription *S*, the number of messages communicated at ES_B, M_{max} is as follows. The details of subscription processes at ESB are described in Section 6.6.2.

(1) If *S* is received from neighboring event servers of RES_A or ESB, M_{max} is $(2N_e\text{-}1)$.

(2) If *S* is received from a local consumer of ES_B, M_{max} is $2N_e$.

Hence on receiving a subscription, the maximum number of messages in fault-tolerance algorithm is $2N_e$.

Theorem 6.2. The *message complexity* of fault-tolerance algorithm is less than equal to $(4N_e + N_{res})$.

Proof. This follows due to Lemma 6.3 and Lemma 6.4.

6.7.3 Storage Complexity

A backup event server of RES requires more storage than other event servers and RESs. Hence we investigate the *storage complexity* at ES_B which is the backup event server of RES_A. ES_B stores two routing tables: its own RT_B and VRT_A of RES_A. Suppose ESB has the following.

N_{con} =the number of consumers at each event server

N_e = the maximum number of neighboring event servers of RESA or ES_B

K = maximum number of subscriptions submitted by each consumer

b_s = average size of each subscription (bits)

Lemma 6.5. The maximum number of subscriptions in RT_B $\leq \{(N_e + 1) \times N_{con} \times K\}$

Proof. As ES_B can receive subscriptions from local consumers and neighboring event servers, the maximum number of subscriptions received from local consumers and neighboring event servers are $N_{con} \times K$ and $N_e \times N_{con} \times K$ respectively. Hence, the maximum number of subscriptions stored in RT_B is $\{N_{con} \times K + N_e \times N_{con} \times K\} = \{(N_e + 1) \times N_{con} \times K\}$.

Lemma 6.6. The maximum number of subscriptions in VRT_A ≤ $\{N_e \times N_{con} \times K\}$.

Proof. As ES_B stores subscriptions from neighboring event servers of RESA in VRT_A, the maximum number of subscriptions in VRT_A is $\{N_e \times N_{con} \times K\}$.

Theorem 6.3. The *storage complexity* for subscriptions at backup event server, event server and region leader is as follows.

(a) The *storage complexity* at backup event server is $\{(2N_e + 1) \times N_{con} \times K \times b_s\}$ bits.

Proof. This follows due to Lemma 6.5 and 6.6 as the backup event server ES_B maintains RT_B and VRT_A.

(b) The *storage complexity* at each event server (except backup event server) is $\{(N_e + 1) \times N_{con} \times K \times b_s\}$ bits.

Proof. This follows due to Lemma 6.5 as each event server maintains the maximum number of subscriptions like RTJB of ES_B.

(c) The *storage complexity* at a region leader RES is $\{N_e \times N_{con} \times K \times b_s\}$ bits.

Proof. This follows due to Lemma 6.6 as the maximum number of subscriptions stored at RES is the same as VRT_A.

6.7.4 Updating of Replicated Subscriptions

In fault-tolerance strategy, backup event server ES_B maintains a virtual routing table of RES_A (VRT_A) besides its own routing table RT_B as presented in Section 6.5. RT_A stored at RES_A and VRT_A stored at ESB are necessary to be consistent. In this section we will describe when and how routing table replications is to be performed and maintaining consistency among replications can be achieved. Hence, whenever it receives a new subscription RES_A updates the subscription to ES_B as follows.

If a subscription is lost in a routing table, the consumer who subscribed the subscription will not receive corresponding notifications which match the subscription. If the replication technique uses the *periodically updates* (i.e., at constant time interval)

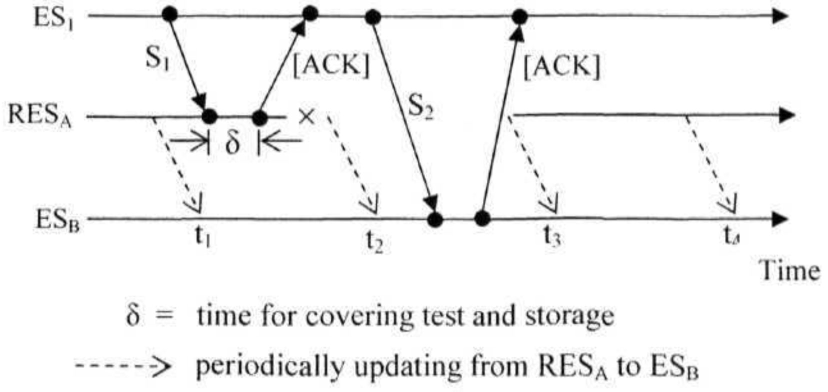

ES_1
S_1
[ACK]
[ACK]
RES_A
×
δ
S_2
ES_B
t_1
t_2
t_3
t_4
Time
δ = time for covering test and storage
-----> periodically updating from RES_A to ES_B

Figure 6.7: Loss of subscription *S* in periodically update

between RES_A and its backup event server ESB, some subscriptions may be lost in its backup event server ES_B as shown in Figure 6.7. For example, a subscription *S* is in RT_A of RES_A, but it is not in VRT_A at ES_B as RESA has failed before *S* is sent to ES_B. Hence, a RESA should update *on demand* (i.e., update whenever a new subscription is inserted in its routing table) to backup event server ESB as described in Figure 6.8. When each RESA receives a subscription *S*, it inserts *S* in its routing table and sends acknowledgement of receiving *S* to the sender. Hence if a RES_A fails before a subscription *S* is inserted in the routing table (i.e., RES cannot reply acknowledgement to sender of S_1), the event server which sent *S* will resend *S* to backup event server. It ensures that subscriptions are not lost when RES_A fails.

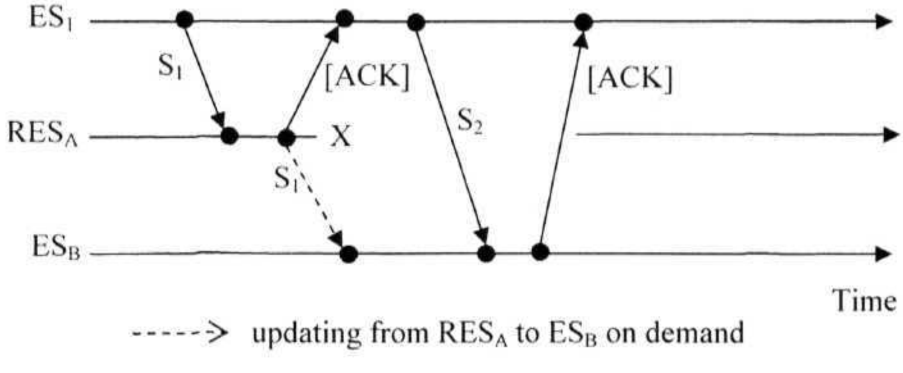


Figure 6.8: Update of subscriptions on demand

6.8 Study on Architectures and Routing Tables

Event notification service system is usually defined on a connected network (without regions) with routing tables at each event server [13, 18, 61, 27]. We have proposed region-based architecture for fault-tolerance strategy. The region-based architecture reduces the sizes of routing tables and thus matching time at event servers also decreases. It is interesting to study the relation between architecture and size of routing tables. For this purpose we have undertaken a simulation.

The main objective of this section is to evaluate on sizes of routing tables with respect to the number of subscriptions. In this simulation study, we do not model explicitly the number of consumers and simulation scenarios include sequences of issuing subscriptions and publishing notifications. In practice, subscriptions issued by consumers have similarity (i.e., some subscriptions are same or covered by each other). For example in a stock trading application, the number of possible different stocks is much less than the total number of subscriptions. We investigate the effects that are caused by varying the degree of Subscription Similarity Ratio (SSR). SSR = 0.5 means that the 50% of subscriptions are the same or covered by each other, i.e., possible different subscriptions in an application is 50% of total subscriptions issued by consumers to an event notification service system. SSR = 0.75 means the similar subscriptions of an application is 75% of total subscriptions in the system (i.e., possible different subscriptions is 25% of total subscriptions).

6.8.1 Simulation Setup

Experiments of simulation were done on the same specifications of PC and in Java as described in Section 4.7.1. We constructed experiments to compare two architectures: one is our *multi-region-based network* (region-based architecture) and another is *single-region-based network.* It is obvious that a network of event servers of ENS system can form either a hierarchical or a peer-to-peer architecture. Hierarchical architecture for ENS system is appreciated than the peer-to-peer one because in case of the latter sizes of routing tables are almost equal. In case of larger network of event servers and larger number of subscriptions, the size of routing table is large and so many such tables are expensive to maintain. Hence most of ENS systems [13, 14, 18, 27] prefer hierarchical architecture as the routing tables at lower levels of hierarchy is usually sparse. But routing tables at higher levels grow faster and become expensive. In hierarchical network of ENS system, subscriptions are propagated only upwards in the tree and event servers store them in routing tables. Hence, the issue is always under study by researchers and we study the routing table sizes for classical network assuming as single region and multi-region-based network. In simulations, for both cases we have taken hierarchical architecture for interconnecting event servers. Multi-region-based network reduces the sizes of routing tables and thus load of event server also decreases.

The number of event servers in the entire network is 50 and we divided five regions for multi-region-based network. Hence each region consists of 10 event severs interconnected by hierarchical network. We choose root event server as region leader RES of each region because center node of hierarchical network is the root node. In single-region-based network, total number of event server is 50 and root event server is used as a single source of publication of events/notifications. Similarly in multi-region-based network, notifications are published from a root event server of a region. We use low rates of subscribing and publication without hitting any processing capacity. In experiments, subscriptions are equally distributed among the event servers (i.e., consumers are equally located in each event servers). We vary the number of subscriptions sent from each event server and hence the total number of subscriptions in the system is 50 times of subscriptions of an event server for both architectures. The routing table sizes of root event servers and sum of routing tables (stored at 50 event servers) in the network are measured in experiments. We also investigate the effect of SSR on routing table sizes by using two values of 0.75 and 0.5. The values of results reported in the plots arc the mean of 5 independent runs.

6.8.2 Simulation Results

Figure 6.9 and 6.10 show the comparison of sizes of routing tables at root event servers of two architectures by varying the total number of subscriptions (as described in Section 6.8.1) with parameter SSR values 0.75 and 0.5 respectively. The result we expect from this simulation is a reduction in the number of subscriptions of routing table at root event servers in multi-region-based network. This is also an indication of the *storage complexity* of root event server. In multi-region-based network, subscriptions are only forwarded within a region. Hence, the routing table size at root event server of multi-region-based network is less than those of single-region-based network.

Figure 6.11 shows the comparison of sum of routing tables of all event servers in the system for two architectures. In multi-region-based network, since subscriptions are localized within each region, sum of routing tables are related to event servers of each region. In single-region-based network, sum of routing tables are related to all event servers of the entire network. The experiments also investigate the effect caused by varying the degree of SSR among the subscriptions of consumers. The SSR has more effect in single-region-based network as subscriptions are forwarded to many event servers of large network. The above experiments demonstrate that multi-region-based network significantly reduces the routing table sizes (storage complexity) and hence the subscription forwarding messages (message complexity) also decrease.

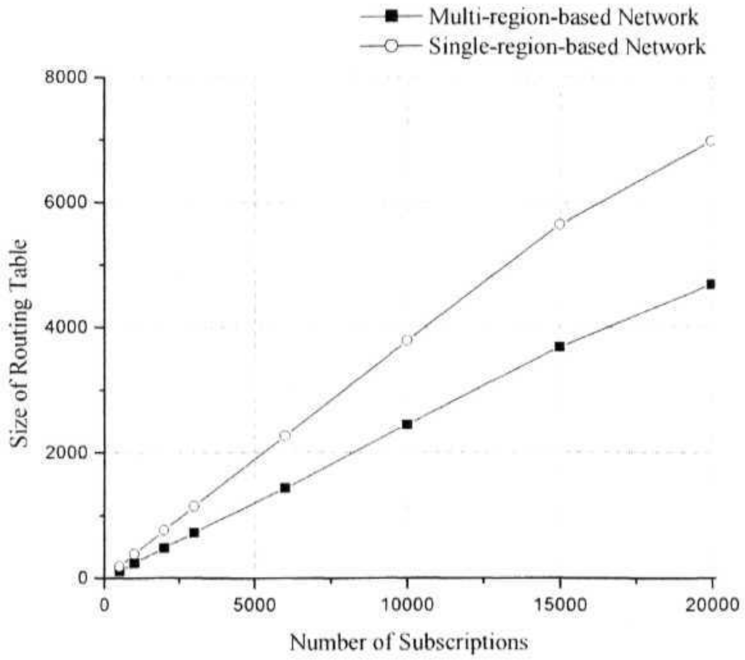


Figure 6.9: Average routing table size of root event server with SSR=0.75

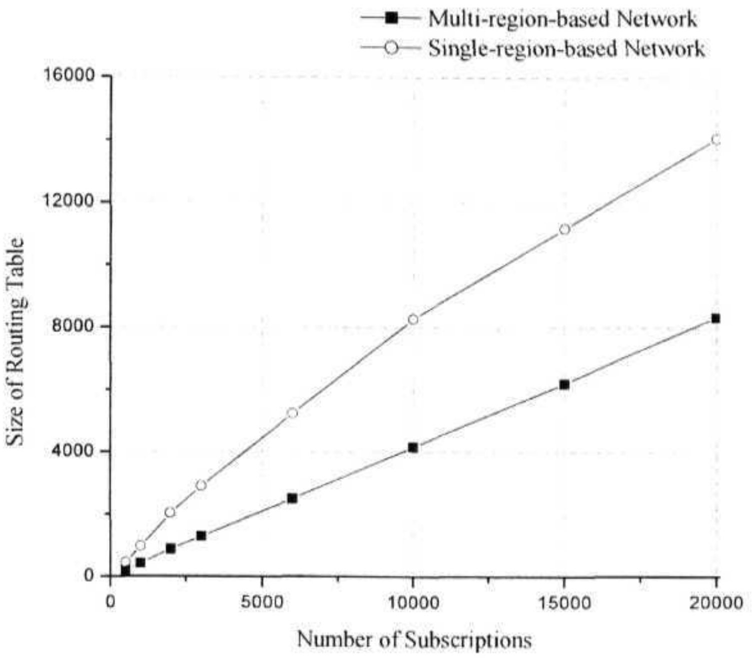


Figure 6.10: Average routing table size of root event server with SSR=0.5

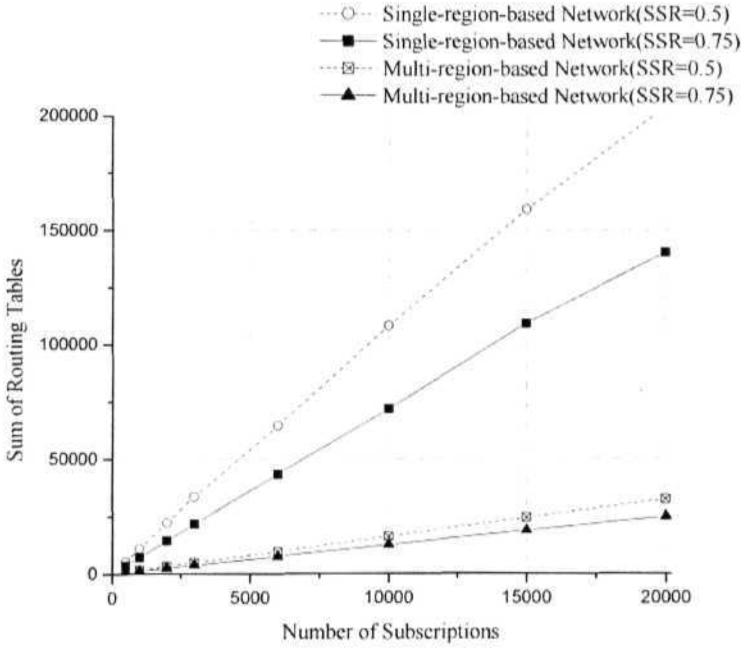


Figure 6.11: Multi-region-based vs. Single-region-based Network (sum of routing tables)

6.9 Summary

In a large-scale event notification service system, failures of event servers and links between them are not uncommon. In this chapter, we describe a fault-tolerant region-based architecture for a large-scale event notification service system and a fault-tolerance strategy for this architecture. Firstly, we identify the types of failures in event notification service systems. Then we present a fault-tolerant region-based architecture by dividing the whole network of ENS system into a few regions and advantages of region-based architecture. The region-based architecture consists of three layers: physical network, event server network and RES network. Each Region Leader (RES) of RES network is responsible to communicate other regions. In region-based architecture, a notification is forwarded to reach all RESs although there is no interested consumer for the notification. In hierarchical architectures of ENS systems as described in [13, 18], a notification is also forwarded to root event server although no interested consumer exists. For a notification published from a region, if an interested consumer exists in another region, region-based architecture has benefits on latency as each RES forwards the notification to reach all regions without matching with subscriptions.

We propose a fault-tolerance strategy for region-based architecture. In fault-tolerance strategy, each RES is replicated at backup event server which is a neighboring event server of the RES. The backup event server executes the processes of RES when RES fails. Hence the fault-tolerance strategy makes the system resilient to failures of RESs and links between them. A failure in a region has no effect on other regions because a notification can reach to every region. The data (routing table) replication is only done for region leaders and thus it is not costly in the system as the number of region leaders is very less than the total number of event servers of the entire network. Then, we present an analysis for the fault-tolerance algorithm to observe the *time complexity*, *message complexity* and *storage complexity*. Finally we study the effectiveness of region-based architecture with experiments of simulation. The experiments demonstrate that the region-based architecture is clearly superior to the hierarchical architecture in routing table sizes at event servers.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the work presented in this thesis and then conclude with a discussion of related research issues that remain open for future work. The architecture of distributed systems is dominated by client/server platforms relying on synchronous request/reply. This architecture is not well suited to implement information-driven applications like news delivery, stock quoting, weather and traffic reports, and dissemination of auction bids due to limitations of traditional request/reply systems. These limitations include tight coupling, synchronous and one-to-one communication. In contrast to that, an event notification service system operates with independence being decoupled to information-driven applications because event producers and consumers in real life need not be always be synchronous. Moreover, the characteristics of event notification service systems allow the system to adapt quickly to frequent connections and disconnections of mobile nodes and characteristics of mobile network. Therefore, event notification service system should be the first choice for implementing such applications for mobile computing environment.

A basic event notification service that provides only communication support is not sufficient because the requirements of event notification service systems differ based on application domains and environments. Event notification service system is widely recognized as being well suited to interconnecting the components of mobile applications since it naturally accommodates a dynamically changing population of components and the dynamic reconfiguration of the connections between them. In this thesis, we first propose a conceptual architecture of event notification service system for mobile computing environment. We identify three important research problems when developing event notification service system for mobile environment. For these research problems, we proposed three services. The service for *ordering of events* ensures to maintain causal ordering of events and to avoid loss and duplication of events in mobile environment. *Just-in-time delivery of events* service supports delivery of notifications to a mobile consumer as soon as it reaches a new location by reducing the delay of handoff process. *Resilient dissemination of events* service is proposed to minimize the impact of failures of event servers and links in the network. We give a summary of the research presented in this work in Section 7.1. In Section 7.2, we outline the open issues with regard to event notification services, and make some proposals for future work.

7.1 Summary of the Work Done

In Chapter 2, we begin with an overview of event notification service system and introduce routing strategies and subscription styles. We then identify the eight research issues for further investigation. The related technologies of event notification service systems are surveyed and classified based on architecture, subscription style and support for mobility. After that, we describe some results of important research issues of event filtering, event ordering and security.

We identify requirements of an event notification service system in Chapter 3. Requirements consist of two parts: typical requirements for every notification service system and additional requirements depending on different applications and environments. We present the designs of event notification service systems for two types of mobile network: *wireless network with base station* (e.g., GSM cellular network) and *ad-hoc network* without base station (e.g., Bluetooth). As *wireless network with base station* is widely used in practice, we present a conceptual architecture of an event server in event notification service for this network. This architecture consists of routing tables, subscription manager, notification manager and communication manager. The services we present in our work are based on this basic architecture.

The research problem of *event ordering* in event notification service system is addressed by our causal ordering algorithm described in Chapter 4. We define causal ordering for event notification service systems and it consists *of subscription ordering* and *notification ordering*. We present a causal ordering algorithm with two parts: static module and handoff module. Static module includes subscription forwarding algorithm and notification delivery algorithm and they operate on static network of event servers. Handoff module is executed when a mobile consumer/producer connects to a new event server and it also includes two handoff algorithms for consumers and producers. Handoff module provides to avoid loss and duplication of subscriptions/notifications when a mobile consumer or producer disconnects and reconnects to current event server or a new event server. We prove the *safety* and *liveness* properties of the causal ordering algorithm for its correctness. We analyze the causal ordering algorithm to investigate *message overhead* and the *handoff complexity*. In simulated experiments, it demonstrates that the causal ordering algorithm ensures causal ordering of subscriptions/notifications and avoids loss and duplication of subscriptions/notifications with a relatively low cost of latency for delivering them. Hence, the research problem defined in Section 1.2.1 is worked out.

We argue that mobile consumers need to receive notifications within a reasonable delay at new locations for deadline-bound applications such as stock trading, auctioning and monitoring of traffics, etc. Our just-in-time delivery service provides that a mobile consumer receives notifications as soon as it reaches a new location. The (normal) handoff process of our causal ordering algorithm (Chapter 4) provides to maintain causal ordering of subscriptions/notifications and to avoid losses and duplication of them. But this normal handoff cannot avoid delays for transferring subscriptions and notifications from previous event server to new event server, and re-subscribing the subscriptions for new paths of notifications at new event server. We introduce a location-based pre-handoff technique for just-in-time delivery of events (notifications) to mobile consumers who migrate to another location area and connect to new event servers in Chapter 5. The location-based pre-handoff algorithm is based on regional map and location tables of mobile hosts. It is a proactive strategy which initiates updates of routing tables and location tables of location areas to which a mobile consumer may visit during its travel. We observe time complexity, message complexity and storage complexity of location-based pre-handoff algorithm. Complexity analysis and experimental results prove that location-based pre-handoff algorithm effectively reduces the latency for delivery of notifications at a new location of a mobile consumer with a reasonable cost of message complexity. Experiments also show that correct movement prediction (CMP) of mobile consumers greatly affects the message complexity. Hence, the research problem defined in Section 1.2.2 is worked out.

The final service developed in this thesis addresses resilient dissemination of events. It is motivated by the requirements for fault-tolerance in event notification service systems. Fault-tolerance is an important feature in a large-scale event notification service system as link and node failures are expected to be more in wide-area networks than local-area networks. In Chapter 6, we introduce the two main parts: region-based architecture and a fault-tolerance strategy on the region-based architecture. Region-based architecture is designed to minimize the impact of faults occurring in an event notification service system. Moreover it reduces the size of routing tables at event servers and latency for delivery of notifications to consumers. In region-based architecture, a region leader (RES) is selected for each region, and the region leader is responsible to communicate with other regions. The fault-tolerance algorithm consists of two parts: processes of a region leader and processes of backup event server of the region leader. The replicated data structure at backup event server is a routing table of its region leader. The region-based architecture and the fault-tolerance strategy ensure that a system failure in a region has no effect on other regions and a notification can reach all regions. We present an analysis for the fault-tolerance algorithm to observe the *time complexity*, *message complexity* and *storage complexity*. The experiments demonstrate that the region-based architecture reduces the sizes of routing tables and latency of delivery of notifications to consumers. Hence, the research problem defined in Section 1.2.3 is met.

7.2 Future Work

In this section, we point out open issues and suggest promising directions for future work. Much work remains to be done to improve what we have achieved so far and to explore new solutions. An event notification service system can be extended with a variety of additional services. We present here some plans for future works that are well out of the scope of this thesis and might follow up from this research.

Event Composition. Event composition is about generating new events by detecting patterns (e.g., sequences) of occurring events. At present, new applications emerge that call for a service that integrates event-information from different sources (producers). One example is a traveler information system that uses data about public transport, highway traffic news, and weather information. To support these applications, it is necessary to integrate information that is provided by different sources using different event types and semantics. The causal ordering used in event ordering service described in Chapter 4 cannot be used for event composition as event composition is necessary global order of events from different sources to make up the composite event. Hence, one of the main problems with composite events in distributed systems is the absence of a global clock. For composition of events from fixed sources, their physical clocks can be synchronized by clock synchronization protocol like NTP (Network Time Protocol). NTP provide reference time injected by GPS time servers. If the event sources are located on mobile hosts, NTP cannot be used to synchronize the reference time through wireless links. Hence the global reference time can be achieved only if mobile hosts are equipped with GPS receiver.

Location-based Service. Event notification service systems can support several application fields. Just-in-time delivery service described in Chapter 5 can be extended for location-based services (e.g., information on traffic jams or free parking spaces). An investigation on this problem has practical relevance. For location-based services, an event server needs to put location information in subscriptions issued by mobile consumers before forwarding the subscriptions. Changing location of a consumer might be seen as an event that triggers the delivery of information. Location dependent applications include yellow pages, maps and traffic report. These applications may need to know the directions of mobile hosts. For example, a mobile consumer driving on a highway, the notification delivered to him depends on information of future destination rather than information based on the location from which subscription was submitted.

The experimental results location-based pre-handoff algorithm for just-in-delivery service show that correct movement prediction (CMP) (i.e., knowing exact future locations) of moving mobile consumers can greatly reduce message complexity and space complexity. The location-based pre-handoff algorithm of just-in-time delivery can be extended to execute pre-handoff process at an exact location area to which a mobile user will move by usage of direction of the moving mobile host besides its location. The direction of a mobile host can be predicted by using its previous cell and current cell. For mobile devices with installed GPS (Global Positioning System) receiver, an event server can easily get exact location and direction of a mobile host. Hence, location-based pre-handoff approach for just-in-time delivery service can be improved to reduce message complexity by using direction towards which a mobile host is moving.

Multiple Receivers. Event notification service system should provide means for users to connect to the system with a wide variety of devices running a diverse collection of system software (operating system). These devices are different in size, performance, and power consumption. Mobile users may have some form of PC or laptop which they want to use in conjunction with mobile devices such as PDAs and mobile phones. Moreover, the size of a notification should be small for mobile devices and some large size of information (several MB) cannot be downloaded to mobile phones. Hence, volume of a notification is to be dynamically decided according to the sizes of devices and their profiles.

Bibliography

[1] M. Aguilera, R. Strom, D. Sturman, M. Astlcy, and T. Chandra. Matching Events in a Content-based Subscription System. *In Proceedings of 18th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* pages 53-61, 1999.

[2] S. Alagar, and S. Venkatesan. Causal Ordering in Distributed Mobile Systems. *IEEE Transactions of Computers,* 6(3), March 1997.

[3] M. Altinel, and M. .1. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of 26th VLDB Conference*, 2000.

[4] R. Arkins. *Perisieni Elvin for Mobile Devices.* BlnfTech honours thesis, The University of Queenslands, November 2000.

[5] B. R. Badrinath, A. acharya, and T. Imielinski. Structuring Distributed Algorithms for Mobile Hosts. In *Proceedings of the 14th International Conference on Distributed Computing Systems,* June 1994.

[6] R. Baldoni, R. Beraldi, L. Querzoni, and A. Virgillito. A Self-Organizing Crash-Resilient Topology Management System for Content-based Publish/Subscribe. In *Proceedings of International Workshop on Distributed Event-Based Systems (DEBS '04)*, Edinburgh, Scotland, UK, May 2004.

[7] C. Bornhövd, M. Cilia, C. Liebig, and A. Buchmann. An Infrastructure for Meta-Auctions. In *Proceedings of Second International Workshop on Advance Issues of E-Commerce and Web-based Information Systems (WECWIS'OO),* San Jose, California, June 2000.

[8] L. F. Cabrera, M. B. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Services. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, Elmau, Germany. IEEE Computer Society, May 2001.

[9] A. Campailla, S. Chaki Edmund Clarke, S. Jha and H. Veith. Efficient Filtering in Publish-Subscribe Systems Using Binary Decision Diagrams. In *Proceedings of ICSE 2001,* pages 443-452, 2001.

[10] M. Caporuscio, A. Carzaniga, and A.L. Wolf. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. *IEEE Transactions on Software Engineering,* 29(12), pages 1059-1071, 2003.

[11] M. Caporuscio, A. Carzaniga, and A. E. Wolf. An Experience in Evaluating Publish/Subscribe Services in a Wireless Network. In *Proceedings of Third International Workshop on SoftM'are and Performance,* Rome, Italy, July 2002.

[12] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Content-based Addressing and Routing: A General Model and its Application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, January 2000.

[13] A. Carzaniga, D. S. Rosenblum, and A. E. Wolf. Design and Evaluation of a Wide-area Event Notification Service. *ACM Transactions on Computer Systems,* 19(3), pages 332-383, August 2001.

[14] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks.* PhD thesis, Politecnico di Milano, Italy, December 1998.

[15] M. Cilia, L. Fiege, C. Haul, A. Zeidler, and A. P. Buchmann. Looking into the Past: Enhancing Mobile Publish/Subscribe Middleware. In *Proceedings of 2nd Int'l Workshop on Distributed Event-Based Systems (DEBS '03),* ACM Press, San Diego/CA, USA, pages 1–8, 2003.

[16] B. P. Crow, I. Widjaja, J. G. Kim, and P. T. Sakai. IEEE 802.11 Wireless Local Area Networks. *IEEE Communications Magazine,* pages 116-126, September 1997.

[17] G. Cugola, D. Frey, and A. L. Murphy. Minimizing the Reconfiguration Overhead in Content-based Publish-Subscribe. In *Proceedings of the 19th ACM Symposium on Applied Computing (SAC04),* Nicosia (Cyprus), pages 1134-1140, March 2004.

[18] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI Event-based Infrastructure and its Application to the Development of the OPSS WFMS. *IEEE Transactions on Software Engineering,* 9(27), pages 827-850, September 2001.

[19] G. Cugola and E. Di Nitto. Using a Publish/Subscribe Middleware to Support Mobile Computing. In *Proceedings of the Workshop on Middleware for Mobile Computing*, Heidelberg, Germany, November 2001.

[20] S. DasBit, and S. Mitra. Challenges of Computing in Mobile Cellular Environment-a Survey. *Journal of Computer Communications*, Elseiver, 26(18), pages 2090-2105, December 2003.

[21] J. Elson, L. Girod, and D. Estrin. Fine-Grained Network Time Synchronization using Reference Broadcasts. In *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, Boston, MA, pages 147-163, December 2002.

[22] P. T. Eugster, R. Guerraoui, and C. H. Damm. On Objects and Events. In *Proceedings of OOPSLA 2001,* Tampa Bay, USA, October 2001.

[23] F. Fabret, A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. *In SIGMOD Record,* 30(2), pages 115-126, June 2001.

[24] F. Fabret, H. Jacobsen, F. Llirbat, J. Pereira and D. Shasha. Webfilter: A High-throughput XML-based Publish and Subscribe System. In *Proceedings of 26th VLDB Conference,* Rome, Italy, 2001.

[25] L. Fiege, G. Mühl, and A. Buchmann. An Architectural Framework for Electronic Commerce Applications. In *Proceedings of Informatik 2001: Annual Conference of the German Computer Society,* Vienna, Austria, September 2001.

[26] L. Fiege, G. Mühl, and F. C. Gartner. A Modular Approach to Build Structured Event-based Systems. In *Proceedings of ACM Symposium on Applied Computing (SAC),* Madrid, Spain, 2002.

[27] L. Fiege, and A. Zeidler. Supporting Mobility in Content-based Publish/Subscribe Middleware. In *Proceedings of ACM/IFIP/Usenix International Middleware Confonference (Middleware 2003)*, volume 2672 of LNCS, Springer-Verlag, pages 103-122,2003.

[28] L. Fiege A. Zeidler A. Buchmann, R. Kilian-Kehr, and G. Mühl. Security Aspects in Publish/Subscribe Systems. In *Proceedings of Third International Workshop on Distributed Event-Based Systems (DEBS'04)*, 2004.

[29] M. J. Franklin, and S. B. Zdonik. Data In Your Face: Push Technology in Perspective. In *Proceedings ACM SIGMOD International Conference on Management of Data,* ACM Press, Seattle, Washington, USA, pages 516-519, June 1998.

[30] R. K. Ghosh, S. K. Rayanchu, and H. Mohanty. Location Management by Movement Prediction Using Mobility Patterns and Regional Route Maps. In *Proceedings of IWDC'03*, volume 2918 of LNCS, Springer-Verlag, pages 153-162, 2003.

[31] R. E. Gruber, B. Krishnamurthy, and E. Panagos. The Architecture of the READY Event Notification Service. In *Proceedings of J 9th IEEE International Conference on Distributed Computing Systems Middleware Workshop,* Austin, Texas, USA, May 1999.

[32] E. Hanson, M. Chaabouni, C. Kim, and Y. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *SIGMOD Record,* 19(2), pages 271-280, June 1990.

[33] D. Harel. *Algorithmics: The Sprit of Computing,* Second Editions, Addison-Wesley, 1993.

[34] Y. Huang, and H. Garcia-Molina. Publish/Subscribe in a Mobile Environment. In *Proceedings of the 2nd ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDEOl),* Santa Barbara, CA, May 2001.

[35] Y. Huang, and H. Garcia-Molina. Publish/Subscribe Tree Construction in Wireless Ad-hoc Networks. In *Proceedings of 4th International Conference on Mobile Data Management (MDM 2003),* volume 2574 of LNCS, Melbourne, Australia, Springer-Verlag, pages 122–140, 2003.

[36] B. Hughes, and V. Cahill. Towards Real-time Event-based Communication in Mobile Ad Hoc Wireless Networks. In *Proceedings of 2nd International Workshop on Real- Time LANS in the Internet Age 2003 (ECRTS/RTL1A03).* Porto, Portugal, pages 77-80, 2003.

[37] J. Kaiser, and M. Mock. Implementing the Real-time Publisher/Subscriber Model on the Controller Area Network (can). In *Proceedings of Second International Symposium on Object-Oriented Distributed Real-Time Computing Systems,* 1999.

[38] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM,* 21(7), July 1978.

[39] Chao-Ping LI, and Ting-Lu Huang. A Mobile-Support-Station-Based Causal Multicast Algorithm in Mobile Computing Environment. In *International Conference on Information Networking (ICOIN-11),* Taipei, Taiwan, January 1997.

[40] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of 4" International Conference on Cooperative Information Systems (CoopIS' 99),* pages 70-78, September 1999.

[41] C. Liebig, B. Boesling, and A. Buchmann. A Notification Service for Next-generation Systems in Air Traffic Control. In *GI-Workshop*: Multicast-Protokolle und Anwendungen, German Computer Society, Braunschweig, Germany, May 1999.

[42] Chit Htay Lwin, Hrushikesha Mohanty, and R. K. Ghosh. Causal Ordering in Event Notification Service Systems for Mobile Users. In *Proceedings of ITCC'04,* IEEE CS Press, Las Vegas, USA, pages735-740, April 2004.

[43] Chit Htay Lwin, Hrushikesha Mohanty and R. K. Ghosh. Just-in-time Delivery of Events in Event Notification Service Systems for Mobile Users. In *Proceedings of INTELLCOMM 2004,* volume 3283 of LNCS, Bangkok, Thailand, pages 190-198, November 2004.

[44] Chit Htay Lwin, Hrushikesha Mohanty, R. K. Ghosh, and Goutam Chakraborty. Resilient Dissemination of Events in a Large-Scale Event Notification Service System. In *Proceedings of IEEE International Conference on e-Technology, e-*

Commerce and e-Service (EEE 2005), Hong Kong, China, pages 502-507, March 2005.

[45] Chit Htay Lwin, Hrushikesha Mohanty, and R. K. Ghosh. A Survey on Event Notification Service Systems. In *Proceedings of International Conference on Information Technology (CIT 2003)*, Bhubaneswar, Orissa, India, pages 314-319, December 2003.

[46] Chit Htay Lwin, Hrushikesha Mohanty, and R. K. Ghosh. Event Notification Service Systems and Mobile Commerce: A Survey. In *Proceedings of International Workshop on Mobile Commerce in association with International Conference on Emerging Technologies (ICET '03)*, Bhubaneswar, Orissa, India, pages 71-101, December 2003.

[47] M. Mansouri-Samani, and M. Sloman. GEM: a Generalised Event Monitoring Language for Distributed Systems, *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2), pages 96–108, June 1997.

[48] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of International Workshop on Parallel and Distributed Algorithms*, Elsevier Press, North Holland, pages 215-226, 1989.

[49] R. Meier. Communication Paradigms for Mobile Computing. *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, vol. 6, pages 56-58, 2002.

[50] Z. Miklós. Towards an Access Control Mechanism for Wide-area Publish/Subscribe Systems. In *Proceedings of 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, IEEE Press, Vienna, Austria, 2002.

[51] S. Mullender. *Distributed Systems*. Second Edition. Addison Wesley, 1993.

[52] G. Mühl, L. Fiege, and A. Buchmann. Filter Similarities in Content-based Publish/Subscribe Systems, In *Proceedings of International Conference on Architecture of Computing Systems (ARCS'02)*, volume 2299 of LNCS, pages 224-238, 2002.

[53] G. Mühl, L. Fiege, F. C. Gärtner and A. P. Buchmann. Evaluating Advanced Routing Algorithms for Content-based Publish/Subscribe Systems. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, & Simulation of Computer & Telecommunications Systems (MASCOTS'02)*, Texas, USA, pages 167-176, October 2002.

[54] Object Management Group. The Common Object Request Broker Architecture: Core Specification, Revision 3.0. Specification, December 2002.

[55] R. Oliveira, J. Pereira, and A. Schiper. Primary-backup Replication: From a Time-free Protocol to a Time-based Implementation. In *Proceedings of 20th IEEE Symposium on Reliable Distributed Systems (SRDS'01)*, New Orleans, La., USA, pages 14-23, October 2001.

[56] OpenQueue. http://openqueue.sourceforge.net, last visited: 18th March 2005.

[57] M. R. Pearlman, and Z. J. Haas. Determining the Optimal Configuration for the Zone Routing Protocol. *IEEE Journal on Selected Areas in Communications,* 17(8), August 1999.

[58] F. Pedone, and S.FrOolund. Pronto: A Fast Failover Protocol for Off-the-shelf Commercial Databases. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Niirnberg, Germany, pages 176-185, October 2000.

[59] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-based Publish/Subscribe Systems. In *Proceedings of Fifth IFCIS International Conference on Cooperative Information Systems (CoopIS'2000)*, Eilat, Israel, September 2000.

[60] G. P. Picco, G. Cugola, and A. L. Murphy. Efficient Content-Based Event Dispatching in the Presence of Topological Reconfigurations. In *Proceedings of the 23*rd *International Conference on Distributed Computing Systems (ICDCS)*, ACM Press, pages 234-243, 2003.

[61] P. R. Pietzuch, and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proceedings of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, Vienna, Austria, pages 611-618, July 2002.

[62] P. Pietzuch, and J. Bacon. Peer-to-Peer Overlay Broker Networks in an Event-Based Middleware. In *Proceedings of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03),* 2003.

[63] P. R. Pietzuch. Hermes: *A Scalable Event-based Middleware.* Ph.D Thesis, University of Cambridge, UK, February 2004.

[64] I. Podnar, M. Hauswirth, and M. Jazayeri. Mobile Push: Delivering Content to Mobile Users. In *Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'()2),* Vienna, Austria, pages 563-570, 2002.

[65] R. Prakash, M. Raynal, and M. Singhal. An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments. *Journal of parallel and Distributed Computing,* 41, pages 190-204, 1997.

[66] J. T. Quah, and G. L. Lim. Push selling—Multicast Messages to Wireless Devices Based on the Publish/Subscribe Model. *Journal of Electronic Commerce Research and Applications,* Elseiver, 1, pages 235-246, 2002.

[67] M. Raynal, A. Schiper, and S.Toueg. Causal Ordering Abstraction and a Single Way to Implement it. *Information Processing Letters,* 39(6), 1991.

[68] W. Rjaibi, Klaus R. Dittrich, and D. Jaepel. Event Matching in Symmetric Subscription Systems. In *Proceedings of GASCON Conference,* 2002.

[69] Sax Project Organization. SAX: Simple API for XML, Version SAX 2.0.1, http://www.saxproject.org/, last visited: 18th March 2005.

[70] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content-based Routing with Elvin 4. In *Proceedings of AAUG2K,* Canberra, Australia, June 2000.

[71] S. Shah, K. Ramamritham, and P. Shenoy. Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers. *IEEE Transactions on Knowledge and Data Engineering,* 16(7), July 2004.

[72] A. Silberschatz, P. B. Galvin, and G. Gagne. *Chapter 15, Operating System Concepts.* Sixth Edition, John Wiley & Sons, 2002.

[73] E. Simon. *Distributed Information Systems ~ from client/server to distributed multimedia*. McGraw-Hill, 1996.

[74] C. Skawratananond, N. Mittal, and V. K. Garg. A Lightweight Algorithm for Causal Message Ordering in Mobile Computing Systems. In *Proceedings of 12th ISC A International Conference on Parallel and Distributed Computing Systems*, pages 245-250, 1999.

[75] W. Stallings. *Wireless Communication and Networks*. Pearson Education, 2002.

[76] D. Sturman, G. Banavar, and R. Strom. Reflection in the Gryphon Message Brokering System, Technical report, IBM T J. Watson Research Center, 1997.

[77] Sun Microsystems. Java Message Service Specification. Version JMS 1.1, http://java.sun.com/products/jms/, last visited: 18th Mach 2005.

[78] Sun Microsystems. Java 2 Platform Enterprise Edition. Version J2EE 1.4, http://java.sun.com/j2ee/, last visited: 18th March 2005.

[79] P. Sutton, R. Arkins, and B.Segall. Supporting Disconnectedness- Transparent Information Delivery for Mobile and Invisible Computing, In *Proceedings of IEEE International Symposium on Cluster Computing and the Grid (CCGrid'01)*, May 2001.

[80] TIBCO. Talarian. http://www.tibco.com/, http://www.talarian.com, last visited: 18th March 2005.

[81] p. Triantafillou, and A. Economides. Subscription Summarization: A New Paradigm for Efficient Publish/Subscribe Systems. In *Proceedings of 24th International Conference on Distributed Computing Systems (1CDCS' 04)*, IEEE CS Press, 2004.

[82] P. Verissimo. *Ordering and Timeliness Requirements of Dependable Real-time Programs*, Chapter (17) of Distributed Systems, 2nd Edition, ACM Press, (1993).

[83] W3C. XML Path Language (XPath) 2.0. W3C Working Draft World Wide Web Consortium, http://www.w3.org/TR/2005/WD-xpath20-20050211/, February 2002.

[84] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security Issues and Requirements for Internet-scale Publish-Subscribe Systems. In *Proceedings of 35th Hawaii International Conference on System Science (HICSS' 02)*, 2002.

[85] V. W.-S. Wong, and V. C. M. Leung. Location Management for Next Generation Personal Communication Networks. *IEEE Network, Special Issue on Next Generation Wireless Broadband Networks*, 2000.

[86] Li- Hsing Yen, Ting-Lu Huang, and Shu-Yuen Hwang. A Protocol for Causally Ordered Message Delivery in Mobile Computing Systems. In *Proceedings of the Second International Mobile Computing Conference,* Hsinchu, Taiwan, R.O.C, pages 35-41, 1997.

[87] A. Zeidler, and L. Fiege. Mobility Support with REBECA. In *Proceedings of 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW '03),* IEEE CS Press, pages 354-36, 2003.