# GUARANTEED REAL-TIME DELIVERY OF CONTEXT – AWARE MESSAGES IN PUBLISH/SUBSCRIBE SYSTEMS

PETRUS SIPHO SHABANGU

20023042

A dissertation submitted in fulfilment of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science, Faculty of Science and Agriculture, University of Zululand

2007

# Declaration

This dissertation represents the author's own research work and has not been submitted in any form to any other institution of tertiary education for another degree or diploma. All the material used as sources of information has been acknowledged in the text.

Signature

# Dedication

I dedicate this dissertation to my family which has always been behind me especially my parents. Their words of support and encouragement have kept me going through difficult times. I also dedicate this work to my late grandfather, who was always eager to see me succeed in my studies; he believed in me and supported me incredibly.

# Acknowledgement

I would like to acknowledge my appreciation to all the staff members of the Department of Computer Science of the University of Zululand and also to the University of Zululand research committee with all the help they provided. I would also like to express my sincere gratitude to my supervisor, Prof. M.O Adigun, for his support and commitment to make this work a reality. He introduced me to research and kept me on the right track with his advice, while giving me the freedom and the possibility to pursue my research ideas.

To my fellow research students of the Department, I would like to thank them for their help. Especially Mr. Jembere, Mr. Otebolako, Mr. Iyilade, Mr. Mudali and Mr. Nyandeni for their input and directions. Special thanks go to Mr. Sibiya, Mr. Kabini and Miss Jili for their time that they had dedicated in order to solve the problems that I encountered to make the development of the prototype a reality.

I would also like to thank all the sponsors for their financial contribution in conducting this research. Especially to Telkom SA thanks for the finance you have provided me with. I would also like to thank Miss Mdletshe for her help and dedication in making sure that we meet deadlines and also for her encouraging words.

To God, the pillar of my strength, who always gives me a way through where there seems to be no way out, I am most grateful to Him.

# ABSTRACT

Publish/subscribe communication paradigm is becoming popular with its decoupling factor, and the filtering of the message stream during the process of dissemination. In a publish/subscribe communication model, a subscriber is decoupled from the publisher, in the sense that a publisher and a subscriber are physically separated from each other. This dissertation reports an ongoing attempt at guaranteeing real time delivery of messages for publish/subscribe systems in a mobile environment where subscribers continuously change location. It focuses on ensuring that messages are delivered in time and space, do not permit stale messages to be delivered but allows subscribers to select priorities based on their preferences.

This research was conducted by first, surveying the theory which gave rise to the theoretical framework used for literature review. Second, the research formulation activity consisted of model building, proving of the crafted model by using a prototyped scenario in which the proposed message delivery architecture was demonstrated and finally, a simulation that tested the reliability of the message delivery model.

The results obtained from this research testified that the emanating message delivery architecture ensured that messages are guaranteed to be delivered to registered subscribers based on subscriber preferences.

# TABLE OF CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# CHAPTER ONE

# INTRODUCTION AND BACKGROUND

## 1.1 Preamble

Embedded and distributed systems are increasingly becoming more complex and comprise components that need to communicate and exchange information. Messaging has emerged as a way of exchanging communication between components in such systems. Messaging facilitates the paradigm shift from traditional centralized applications and data stores towards data-driven systems that comprise autonomously operating components and services. There are a number of message exchange patterns (also called, interaction styles) in the literature. These include: one-way, asynchronous two-way, request-response (RPC), workflow-oriented, publish-subscribe, and composite [Behnel *et al*, 2006]. This study, however, concentrate on publish/subscribe (P/S) message exchange pattern. The publish/subscribe system is generally used where information is being pushed out to one or more parties. It is a special case of message-oriented middleware (MOM) where data producers (publishers) publish notifications to inform consumers (subscribers) about available services (events) they have subscribed to. Having the possibility of routing messages only toward specific areas or subscribing to messages originating in specific locations seems natural [Cugola and Nitto, 2005] when a P/S model of communication is adopted. One challenge in publish-subscribe systems is developing mechanisms to guarantee that messages are effectively delivered to intended

recipients in time and managing message transfer during notification dissemination to manage real-time delivery of messages.This research work addresses this challenge by exploiting context information of subscribers. Context-aware computing became prevalent with the emergence of mobile devices [Julien and Roman, 2006]. Context-awareness was introduced to be able to track the location of a subscriber. By adding location information, messages can be efficiently routed to intended subscriber by the message broker without having to search for where the subscriber is located from the subscriber database. Context, in this case, refers to the geographical context of the subscriber in terms of location. By monitoring the context of the subscriber, the service ensures subscriber's availability to the other participants. Generally, in P/S system, consumers (subscribers) express their interests by subscriptions. Publishers and Subscribers share a mediator (known as, Notification Service) that is now responsible for dealing with crucial issues like scalability, reliability, transaction support, visibility of notifications, expressivity of subscriptions, semantic data exchange, etc.

The P/S paradigm is different from the traditional point-to-point communication models in a number of ways. In publish/subscribe systems the communication between the end points is anonymous, asynchronous and loosely coupled. In other words, publish/subscribe systems decouple publishers and subscribers in space, time and flow [Behnel *et al,* 2006]. This decoupling of publishers and subscribers in time, space and flow makes publish/subscribe systems highly scalable by removing all explicit dependencies between the interacting parties. It also helps the system to adapt quickly to the dynamic environment. Decoupling in space allows the subscriber to move from one location to another without informing the publisher while decoupling in time allows for

disconnected operations of the subscriber. This decoupling in space and time thus, makes publish/subscribe systems a good choice for mobile and weakly connected environments as offered by mobile wireless networks. There is hence a pressing need to extend publish/subscribe systems to mobile wireless environments [Cugola *et al*, 2005].

Publish/subscribe systems have been effectively used to model information dissemination applications where publishers are information sources, subscribers are information destinations, and a broker entity is a router mechanism [Eugster *et al*, 2003]. The publish/subscribe systems naturally support a number of desirable features for mobile applications. They are characterized by decoupling the interacting parties, both in time and space, allowing them to communicate without being connected simultaneously or being aware of each other. They also employ an asynchronous communication style that allows mobile clients to issue requests for services, disconnect from the network, and collect their results later. Publish/subscribe systems moreover, can efficiently filter and disseminate a significant amount of data to a large number of clients. These characteristics make the publish/subscribe paradigm a very good candidate for supporting mobile applications.

## 1.2 Background to the Study

### 1.2.1 Message Exchange Patterns (Interaction Styles)

Message exchange patterns (MEP) are a type of design pattern that describe distributed communications. The patterns describe the interactions between the

different participants on the network (multiple clients, multiple servers). A Message Exchange Pattern (MEP) describes the pattern of messages [Gudgin *et al*, 2002] required by a communications protocol to establish or use a communication channel. There are two major message exchange patterns — a request-response pattern, and a one-way pattern. For example, the TCP has a request-response pattern protocol, and the UDP has a one-way pattern.

A One-Way message exchange is the simplest pattern [WS-Oxygen Tank, 2006]. It states that the sender will give a best-effort to move a message from a source location to the destination. No response to the operation is expected.

The request-response message exchange pattern is where a client asks a service provider a question and then receives the answer to the question [WS-Oxygen Tank, 2006]. The answer may come in the form of a fault/exception. Both the request and the response are independent messages and the request/response pattern is often implemented using synchronous operations for simple operations. For longer-running operations, asynchronous (with message correlation) is often chosen.

The send, receive, and reply operations may be synchronous or asynchronous. A synchronous operation blocks a process till the operation completes. An asynchronous operation is non-blocking and only initiates the operation.

The notion of synchronous operations requires an understanding of what it means for an operation to complete. In the case of remote assignment, both the send and receive complete when the message has been delivered to the receiver. In the case of remote procedure call, the send, receive, and reply complete when the result has been delivered to the sender, assuming there is a return value. Otherwise, the send and receive complete when the procedure finishes execution.

Asynchronous message passing allows more parallelism. Since a process does not block, it can do some computation while the message is in transit. In the case of receive; this means that a process can express its interest in receiving messages on multiple ports simultaneously. In a synchronous system, such parallelism can be achieved by forking a separate process for each concurrent operation, but this approach incurs the cost of extra process management.

Asynchronous message passing introduces several problems. What happens if a message cannot be delivered? The sender may never wait for delivery of the message, and thus never hear about the error. Similarly, a mechanism is needed to notify an asynchronous receiver that a message has arrived. The operation invoker could learn about completion/errors by polling, getting a software interrupt, or by waiting explicitly for completion later using a special synchronous wait call. In this research, we are going to use asynchronous communication for efficient computation.

### 1.2.2 Publish /Subscribe system

A publish/subscribe system is a collection of autonomous components, which interacts by delivering events (or messages) from sources to interested destinations. Components that generate messages are known as publishers, whereas components that consume messages are known as subscribers. As shown in Figure 1.1, the interactions among publisher and subscriber components are coordinated by a mediated entity called event service (dispatcher).

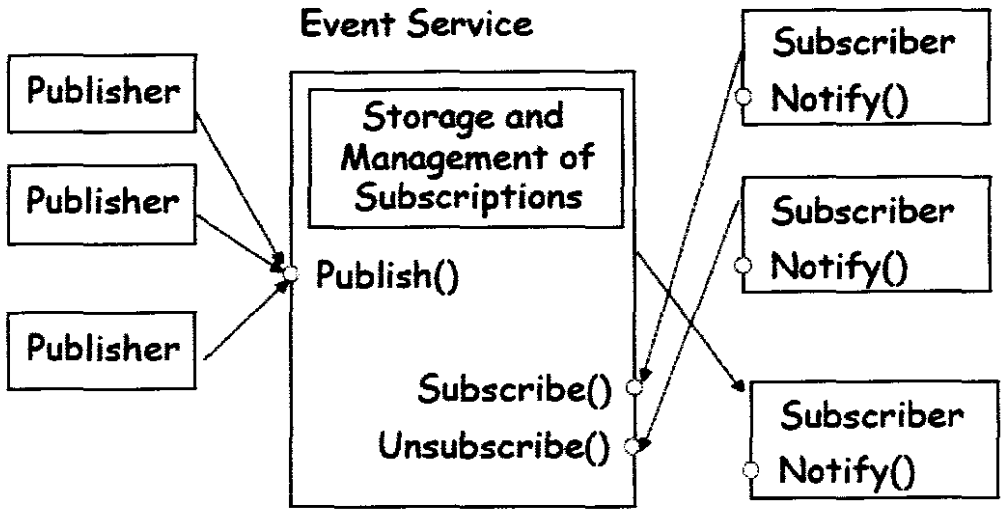



**Figure 1.1: A Simple Publish/Subscribe Architecture**

Publishers are the information providers that notify the outside world about the occurrence of certain events. When subscribers want to receive particular classes of events they express their interest by means of subscriptions. Upon the publication of a new event to the system, the event broker matches the event against all the subscriptions and then forwards it to all interested subscribers. Hence, the architecture of a publish/subscribe system relies on the mediated entity that handles the collection of subscriptions as well as the distribution of events and acknowledgements. Publish/subscribe systems are based on two different types of event subscriptions known as topic-based and content-based subscriptions. In topic-based systems, subscribers may register to one or more topics and hence receive all the events delivered to those topics. Subscribers that share the same topic will receive a copy of each event within that topic. Content-based systems, on the other hand, allow subscribers to assign certain queries on the event content as part of their subscriptions. Thus, subscribers are able to receive a specific set of events within a topic. It should be noted that events do not rely on an explicit destination address set by the publishers. They are instead routed to the end destination based on their content. The architecture of the event broker

can be either centralized or distributed. A centralized architecture consists of a single broker entity that connects several publisher and subscriber components. This central entity is potentially a performance bottleneck and a single point of failure. This affects system scalability and limits the use of centralized architectures to small scale deployments. In a distributed architecture, a number of interconnected brokers collaborate in collecting subscriptions and forwarding events to the interested subscribers. Publishers and subscribers are not attached to a single broker entity; instead, they are distributed over several interconnected brokers. This can potentially reduce the network load and alleviate system scalability. The interconnected brokers can be represented in several topologies that differ in terms of their strategies in routing subscriptions and events. Two different broker topologies are presented in Figures 1.2 and 1.3. In a hierarchical (or multicast) topology, the event brokers are organized in a forwarding tree that has a root broker and several downward brokers. Excluding the root broker, each broker is considered as a client

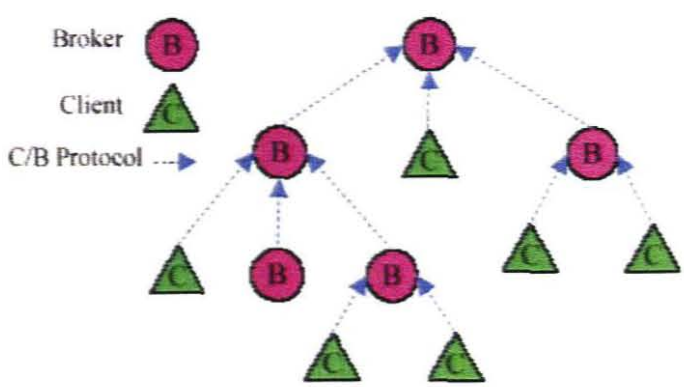



**Figure 1.2: Hierarchical Client/Broker Topology**

to the broker at the upward level of the hierarchy. Subscribers may connect to any broker regardless of the location of the corresponding publishers in the hierarchy.

Whenever a new subscription is received, the broker propagates it upward to the root broker. Each broker on the way from the subscriber to the root broker stores a copy of the subscription. When an event is received by a broker, it is forwarded to the broker's parent. The event is also matched against all the stored subscriptions. This includes any subscriptions from downstream brokers. The broker propagates the event to any interested children (subscriber/broker) only if the matching result is true. Thus, events are always forwarded upward to the root broker, and downward towards any interested subscribers. In this topology, each broker node is a critical point of failure. Also, parent brokers are potentially overloaded as they perform extra work for their children.

A peer-to-peer (or broadcast) topology consists of a set of brokers that are connected in the form of symmetrical peers. Their communication protocol supports a bi-directional flow of subscriptions and events. Each broker is responsible for a partial number of subscriptions.

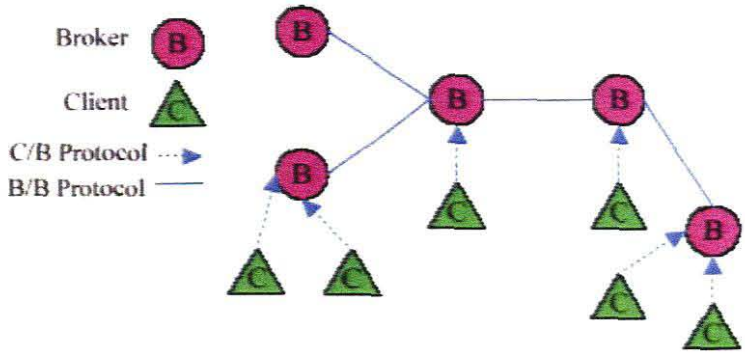



**Figure 1.3: Peer-to-Peer Broker Topology**

A publisher delivers an event to any broker that it is connected to. That broker then becomes responsible for broadcasting the event to all other brokers in the topology. When

a new event enters the system, each broker checks the event against its own subscriptions and forwards it as necessary. It is apparent that the matching and forwarding overhead is reduced in comparison with the previous topology. This is because each broker needs to match events against a portion of subscriptions. In this topology, the network will be flooded by the generated events since they travel to all brokers.

### 1.2.3 Benefits of Publish/Subscribe model

The following are the benefits of the pub/sub model:

i) *Lowered coupling*. The publisher is not aware of the number of subscribers, of the identities of the subscribers, or of the message types that the subscribers are subscribed to.

ii) *Improved security*. The communication infrastructure transports the published messages only to the applications that are subscribed to the corresponding topic. Specific applications can exchange messages directly, excluding other applications from the message exchange.

iii) *Improved testability*. Topics usually reduce the number of messages that are required for testing.

## 1.3 Statement of the problem

Publish/Subscribe systems have been adopted by many industries as a communication protocol. To ensure customer satisfaction, certain problems need to be considered whenever a publisher/producer sends messages to all interested customers. The following have been identified as major problems for message delivery in publish/subscribe systems:

i. Managing message transfer during notification dissemination to manage real-time delivery of messages to intended subscribers;
ii. The issue of message acknowledgements which lead to message duplication and data loss;
iii. Identification of home or foreign broker during the link disconnection (weak signals) situation;
iv. Reliable message delivery to subscribers in time, space and
v. Handling of temporarily disconnected subscribers during message dissemination.

Existing approaches to solving these challenges involves broadcast based, recipient based pattern, and topic based approaches [WS-Oxygen Tank, 2006] which partly addressed these problems. We propose a solution that will ensure delivery of context-aware messages, and also ensure both real-time and guarantee delivery of latest message updates, all stale messages are discarded. The delivery of messages to registered subscribers is performed based on priorities selected during subscription by a subscriber.

## 1.4 Rationale for the study

The drive towards mobile services leads to the integration and convergence of various technologies into a wide range of innovative mobile applications. This coupled with the affordability and popularity of mobile devices amongst all communities, including even the previously technologically disadvantaged communities, makes mobile applications undoubtedly the next wave in the evolution of e-business. Given this background, the envisaged model will aid in the development of context-aware application and guarantee delivery messages for wireless applications. This will contribute significantly to the evolution of mobile applications, especially for people living in rural areas. It will also ensure reliable delivery of all messages submitted and meant for a particular receiver or subscriber, who sometimes lose vital notifications due to network breakage or temporarily disconnected subscribers on the network.

## 1.5 Research Questions

This thesis seeks to answer the following questions through this research:

i. How is delivery of messages ensured for mobile users experiencing weak network signals?
ii. How will the proposed model overcome the data loss during notification dissemination?
iii. How can we ensure high data availability in mobile computing environments where frequent disconnections may occur?

## 1.6 Research Goal and Objectives

### 1.6.1 Goal

The purpose of this research is to develop a mechanism that can guarantee real-time delivery of context – aware messages in Publish/subscribe Systems.

### 1.6.2 Objectives

The goal of this research is formulated as equivalents of the following objectives, which are to:

i. Propose and design a model for guaranteeing real-time message delivery in publish/subscribe system;
ii. Specify performance metrics such as message throughput and scalability;
iii. Implement the proposed model and simulation to determine the performance and
iv. Carry out an evaluation of the simulated system.

## 1.7 Research Methodology

**Background:** It is important that the results obtained from this study should be applicable to all publish/subscribe systems, every system should be able to adapt to link failures and also manage message overloading. A node should be able to acknowledge a message received successfully or unsuccessfully.

1. *Literature survey.* This method entails surveying the background of the area of interest and the theoretical part of this research will involve analyzing related work. This analysis will be done based on theoretical framework that will be specifically developed for this task.

Surveying the field of study, identification of a problem, related work in the area of interest, problem analysis and initial solution to problem.

2. *Model Formulation.* The theoretical knowledge gained from the related work will be used as a foundation of this research. The formulative method of this research involves model formulation and involves the following sub-tasks:

i. Identify existing scholarship that have addressed similar problem by surveying the field of study.
ii. Analysis of the problem solution using the unified modelling language (UML)
iii. Formulate a model to simplify the solution by integrating modules (section 3) proposed to ensure reliable message delivery
iv. Devise algorithms to manage message delivery in time.
v. Devise algorithms to ensure proper matching of subscriptions against publications by further exploiting the matching manager (section 3).
vi. Formulate requirements that the model must satisfy through conducting survey for existing publish/subscribe systems.

3. *Implementation/Simulation of the model and results evaluation.* The proposed matching and guaranteed message delivery module (GMDM) for guaranteeing real-time delivery of messages and management of message lost and temporarily disconnection of

subscription will be implemented as proof of concept by proposing and implementing a mobile sports scenario which will be used to publish sports score updates to subscribers in real time, and a simulation of the implementation that will be developed using VB.net varying the number of messages to ensure delivery reliability this is carried out to evaluate the results.

## 1.8 The Structure of the dissertation

The remainder of this dissertation is structured as follows: Chapter two gives an overview of existing solutions and systems in relation to content dissemination in publish/subscribe systems. We analyze and compare the characteristics of prominent publish/subscribe systems, and related solutions. Chapter three introduces our proposed system model and mechanisms to guarantee message delivery in publish/subscribe systems. Particular emphasis is put on current problems faced with disconnected subscribers, and messages lost during message dissemination and duplication of data. Chapter four presents the practical implementation of the proposed Publish/Subscribe message delivery model and discusses the results obtained from the implementation. The chapter also discusses the simulation experiment carried out to analyse the reliability of the system in terms of message throughput and scalability. Finally chapter five concludes the thesis and gives recommendations for future work.

# CHAPTER TWO

# LITERATURE REVIEW

## 2.1 Introduction

The P/S model represents an emerging paradigm for de-coupled and asynchronous connections between application components (publishers and subscribers) [Eugster *et al*, 2003]. It may support two types of models which are *mediator* and *implicit* models [Behnel *et al*, 2006]. These models support decoupling and asynchronous connections between application components in contrast to peer-to-peer model that only support coupling and synchronous connection between application components. The mediator and implicit models form the basic foundation of the architecture of this research. The message delivery guarantees and allows subscribers to move around at all times without worrying of losing messages due to frequent disconnections.

A number of studies have been conducted towards solving the problem of ensuring message delivery in publish/subscribe systems [Wang, *et al*, 2005; Cao *et al*, 2002; Podnar *et al*, 2002; Podnar and Pripuzic, 2003; Bygdas *et al*, 2001; Chand and Felber; 2004]. These studies proposed different delivery models that try to ensure message delivery to mobile subscribers without any publisher-to-subscriber message delivery guarantees. Some studies proposed brokers at different locations [Eugster, 2003; Fiege,

2003] to deal with message delivery, ignoring the issue of possible duplications. Other studies proposes algorithm to manage disconnected subscriptions [Huang, 2001; Muhl, 2002] but without guaranteeing real-time delivery of context aware messages.

Real-time message delivery aims to ensure that a message is delivered within a specified delivery time. Some messages might be published while a subscriber is disconnected from the network, this might cause the subscriber to lose messages if this type of a situation occurs frequently. Some authors proposed the use of a time to live factor [Fiege *et al*, 2003; Rowstron and Druschel, 2001; Behnel *et al*, 2006]. A time to live factor is a time given in milliseconds to an event or message to live in persistence storage. If this time expires while the subscriber is disconnected, the event is discarded from storage. Some subscribers reconnect to network without the middleware noticing the reconnection, which causes the subscriber to lose notifications.

Guaranteed message delivery ensures that a published message gets delivered to its intended subscriber. Some authors ensure this guaranteed delivery by implementing a network of brokers that collaboratively route messages from information providers to consumers [Chand and Felber, 2004]. A major challenge of such middleware infrastructure is their reliability and their ability to cope with failures in the system. The aim of guaranteed message delivery is to ensure that for every message published, a subscriber receives the message in time and space. No subscriber should receive a message he/she is not interested in, in this way messages will be ensured to be delivered to a targeted subscriber. Guaranteed message delivery goes hand in hand with matching of published events against a subscriber's interest or preferences.

## 2.2 Guaranteeing real-time delivery of context-aware messages

### 2.2.1 Overview

Studies to ensure message delivery was initiated to bring some quality of service to events published to subscribers. Traditional publish–subscribe systems make no provision for accessing past notifications. They guarantee delivery of notification to a consumer only when it is published sufficiently after the consumer has issued a matching subscription. For many applications, this behavior is inappropriate. For example, consider a client starting a stock watchlist. Without access to past notifications, the watchlist displays the current price of a stock only after the provider has published a new quote for this stock and the infrastructure has delivered this quote to the consumer. This can lead to an unacceptable delay. The challenge is to deliver all desired notifications without duplicates.

The following section introduces the framework that is used to analyze the gathering body of literature that has dealt with this aspect of message deliver and notification to subscribers.

### 2.2.2 Framework for Analysing Work Related to ensuring message delivery in publish/subscribe systems

The review of related work was conducted with the aid of the following framework consisting of characteristics identified within the literature. The aim of this framework is to help categorize and critically analyze the published body of research results.

i. **Guaranteed message delivery**

It is necessary to ensure every message published by an information producer gets to a subscriber in time and space. A message is guaranteed to be delivered to a targeted subscriber without any lost data. The message is received in full by a subscriber, and only dated message are delivered to avoid subscriber confusion or receiving unwanted (expired message) information or messages.

ii. **Context-awareness**

Context-awareness, in this case, refers to subscriber's context in terms of location. A message which is context-aware of a subscriber's location gets sent to the subscriber's current broker.

iii. **Real-time message delivery**

A message needs to be delivered at a specified time, some real time delivery systems make use of topic (categories) to specify if a message is of high priority or low priority to be delivered to an intended subscriber. In a mobile sports update system, a subscriber might prefer all soccer match update messages to outrun other published messages. Some systems might use a time to deliver a message, e.g. if message X's delivery time is 5s,

message X is considered useless and is discarded when this specified time elapses due to subscriber's disconnection.

**iv. Persistent Connections**

Mobile clients' connections frequently and unexpectedly break. Persistent connections use a simplified sliding window protocol to ensure that the connection does not lose or duplicate notifications. Both end points of the connection — the client and the broker to which the client is connected — buffer notifications that they have sent temporarily in a queue until the other end point has acknowledged them. When the mobile client re-establishes connection, both directions exchange the sequence number for the last received notification, and transmission continues from that point using available transport mechanisms.

**v. Durable versus non-durable subscription**

Durable subscriptions receive notifications of events published when they are disconnected at the time of delivery. Non-durable subscriptions lose events sent to them during their disconnection period.

**vi. Message queuing and policies**

Queuing policies limit the number of notifications that can queue. They specify when to discard buffered notifications. For example, a client can specify that notifications older than 10 minutes should be discarded or that the infrastructure should deliver only the most current notification regarding a certain data item (a stock, for example). These policies come into play not only in the case of disconnections, but also during congestion.

**vii. Why the use of a monitoring service for continuous subscriber disconnections?**

A monitoring service continuously monitors subscribers during message delivery. Messages are only channelled to connected or active subscribers, and inactive subscribers are monitored till reconnection. A monitoring service ensures or avoids unnecessary delivery of events for disconnected subscriptions, and also avoids congestion on the network.

**viii. Specification of the service to handle disconnected subscriptions**

A mechanism which ensures a disconnected subscriber's events are managed at time of delivery, and also kept in persistence storage for the duration of the disconnection. Some events may take too long; some are discarded from the storage due to the time to live expiring.

**ix. *Accessing Past Notifications***

Traditional publish–subscribe systems provide no ability to access past notifications. They guarantee to deliver a notification to a consumer only when it is published sufficiently after the consumer has issued a matching subscription. For many applications this behavior is inappropriate. Past notifications are notifications which do not expire (e.g. "Yahoo invites you to open a free mail box"), or take long to expire, and are only sent or delivered to newly subscribed consumers (subscribers).

## 2.3 Review of Work Related to Guaranteeing Message Delivery in a Mobile Environment.

This section presents research work done in this field of research. The theoretical framework presented in section 2.2 is used in a tabular form to present the literature.

### i. NDDS: The Real-Time Publish-Subscribe Middleware (Pardo-Castellote, *et al*, 1999)

**Table-2-1- Characteristics of Real-Time Messaging in Publish/Subscribe System [Pardo-Castellote *et al*, 1999]**

| Characteristics | Value |
|---|---|
| Guaranteed message delivery | No message guarantees |
| Context-awareness | Not supported |
| Real-time message delivery | Offers the best transport for distributing data quickly |
| Persistent Connections | Connections break frequently and events are lost. |
| Is the subscription durable or non-durable? | Non durable |
| Message queuing and policies | Messages are not stored, a message is quickly delivered and discarded |
| Any use of a monitoring service for continuous subscriber disconnections? | No |
| Specification of the service to handle disconnected subscriptions? | No |
| Accessing Past Notifications | Sends refresh messages |

Pardo-Castellote *et al* (1999) base their work on ensuring that the P/S Middleware copes with the real-time message delivery. They examine their work based on some characteristics such as time-critical: updates are useless if old, Idempotent: Repeated updates are acceptable, and Last-is-best: latest information is more important than retrying missed samples.

Pardo-Castellote *et al* (1999) answers three questions based on message delivery. These questions are:

- Where is the information generated?
- Where does it have to go?
- When does it have to get there?

The first question can be interpreted as the source of the information, and the second as the information consumer, which in our case is information matching, information published requires accurate matching to ensure correct destination delivery. While the third one is interpreted as time of delivery in which their work is based on.

The authors did not look at a question that addresses how the information should get to the targeted destination. Messages are not guaranteed to reach their destinations since there is no mechanism to ensure messages reach destinations in time and space.

### ii. Supporting Disconnectedness – Transparent Information Delivery for Mobile and Invisible Computing (Sutton *et al*, 2001)

This work examines the issues involved in supporting content-based messaging to both mobile devices and users using a combination of connected and mobile (possibly

disconnected) devices. The discussion in this work is in the context of the development of a proxy-server to provide disconnectedness support for the Elvin content-based messaging service (Sutton *et al* 2001).

Sutton *et al* (2001) describes the issues involved in supporting content-based messaging on mobile devices and how these issues have been overcome in the implementation of a proxy-server to provide disconnectedness support for the Elvin content-based messaging service, they also use a TTL (time to live) factor which is assigned to events of disconnected subscribers, the drawback of this work is the time taken by the proxy server to process delivery of stored events for reconnecting subscribers due to large processor overhead since no priority is being used to vary published events based on their importance and to only store important events which cannot be missed by disconnected subscribers.

This work requires a subscriber to always re-subscribe when reconnecting to the system which is not possible in some cases for mobile subscriber since they frequently disconnect from the network yet, do not wish to miss important notifications. Another drawback of this work is the storage of the expired notifications, which could result in an excessive amount of disk space usage, as many expired notifications could be stored for clients which connect infrequently.

**Table-2-2- Characteristics of Elvin Content-Based Messaging System [Sutton *et al* 2001]**

| **Characteristics** | **Value** |
|---|---|
| Guaranteed message delivery | Wait negative acknowledgement. |
| Context-awareness | Not supported |

| | |
|---|---|
| Real-time message delivery | A message is pushed to all subscribers without any given time |
| Persistent Connections | Allows subscribers to specify a time to live factor |
| Is the subscription durable or non-durable? | Durable |
| Message queuing and policies | Messages are queued both on the mobile device and also on the server |
| Any use of a monitoring service for continuous subscriber disconnections? | No |
| Specification of the service to handle disconnected subscriptions? | Uses scattered proxy servers to handle disconnections |
| Accessing Past Notifications | Not supported, deliver and discard an event. |

### iii. A Simple Architecture for Delivering Context Information to Mobile Users (Bygdas *et al*, 2001)

Bydas *et al* (2001) propose architecture for distributed context aware applications based on code mobility and tuple space technologies. Under this architecture, context information data sources are implemented using T-Spaces and made available on the Internet. Context aware applications running on wireless mobile devices dynamically install their query operators at the appropriate data sources.

The authors base their work on context information, without considering the fact that mobile users change location (in our case we refer to as brokers) frequently, causing the subscriber or mobile device user to loose urgent and important information or events.

This work does not look at how the published information gets to the intended subscriber or information consumer. Sending a message to subscribers requires many factors to be looked at such as, matching information published against subscriber's interests, dealing with unreachable subscribers, storing and managing expired events or messages, and also monitoring the status of disconnected subscriptions.

**Table-2-3- Characteristics of Delivering Context Information to Mobile Users in Publish/Subscribe System [Bygdas *et al* 2001]**

| **Characteristics** | **Value** |
|---|---|
| Guaranteed message delivery | No message guarantees |
| Context-awareness | Not supported |
| Real-time message delivery | Not supported |
| Persistent Connections | Connections break frequently and events are lost. |
| Is the subscription durable or non-durable? | Non durable |
| Message queuing and policies | Messages are not stored, a message is directed to wireless mobile devices from data sources and discarded |
| Any use of a monitoring service for continuous subscriber disconnections? | No |

| | |
|---|---|
| Specification of the service to handle disconnected subscriptions? | No |
| Accessing Past Notifications | No mechanism to access messages with a long live time |

**iv. Mobile Push: Delivering Content to Mobile Users (Podnar *et al*, 2002)**

This work is based on delivering content to mobile users using a push based style for sending information. Podnar *et al* (2002) analyzes the features of a mobile push service by investigating representative usage scenarios and propose architecture for mobile content delivery systems. The architecture is based on the publish/subscribe (P/S) paradigm which supports many-to-many interaction of loosely-coupled entities. They also define set of services that need to collaborate with the P/S infrastructure to address the dynamics of mobile environments.

Podnar *et al* (2002) proposes a solution to mobile disconnections by putting a proxy server between the Elvin [Sutton *et al* 2001] server and a mobile device to queue messages for non-active users. This presented solution by the authors implement a queuing strategy with time-to-live expiry, but it is not clear how location management and distribution are handled. Content dissemination is an increasingly popular service in environments that support user mobility. However, the solution being presented by the authors does not support subscriber migration from broker to broker. Therefore, there is a need to restructure the existing P/S communication paradigm to be applicable in mobile environments where frequent disconnections occurs, and users move from broker to the next without physically noticing the change of the brokers.

What remains a challenge is the question of how to integrate such solutions into a mobile push system, and also considering location management issues to deal with subscriber movements which results in information loss.

**Table-2-4- Characteristics of Push-Based System for Delivery Content to Mobile Users [Podnar *et al*, 2002]**

| **Characteristics** | **Value** |
|---|---|
| Guaranteed message delivery | Proposes the use of content dispatchers which channel the events to subscribers |
| Context-awareness | Introduces a location management service to map unique subscriber identifier to the current IP Addresses. |
| Real-time message delivery | It specifies not delivery time of a message |
| Persistent Connections | Connections break frequently and events are lost. |
| Is the subscription durable or non-durable? | Non durable – drops content for unreachable subscribers |
| Message queuing and policies | Uses a flexible queuing policy |
| Any use of a monitoring service for continuous subscriber disconnections? | No |
| Specification of the service to handle disconnected subscriptions? | No |
| Accessing Past Notifications | Content dispatchers keep events for new subscribers. |

v. **Reliable Message Delivery for Mobile Agents: Push or Pull (Cao *et al*, 2002)**

This work differs slightly from previous work by ensuring reliable message delivery looking at message processing delays and traffic congestion, which is part of what is being investigated in this study, avoiding overloading the middleware with messages which causes congestions. This work also looks at different communication patterns and requirements of real-time message processing, specific applications can select different message delivery approaches to achieve the desired level of performance and flexibility.

However, messages are not guaranteed to be reliably delivered to their destination mobile devices or users. Cao *et al* (2002) compared message delivery when using Pull and Push approaches. However, the authors agree that the simple push mode cannot guarantee reliable message delivery. It is possible that when a message is forwarded to the address as kept in the middleware, the target subscriber may have left for another broker. They then propose a synchronized push mode to avoid message loss and the chasing problem caused by subscriber mobility. This synchronized push mode will be used for synchronization between message forwarding from the broker and subscriber migration. The proposed approach of Cao *et al* (2002) requires a subscriber to wait for an ACK (acknowledgement) message before it can migrate to another broker or location. This might cause a subscriber to wait for a long time to receive for an ACK message which might have been delayed due to message overload and congestion over the network.

The proposed approach will not work in a mobile environment, since subscribers move frequently changing their current location. The proposed approach by Cao *et al* (2002) requires a subscriber's participation which is not always guaranteed in publish/subscribe

communication, since a publisher and a subscriber are decoupled and the communication is asynchronous. This decoupling factor does not guarantee physical participation of the subscriber, therefore a mechanism to ensure and guarantee message delivery is required since message delivery is not guaranteed in a mobile environment for decoupled participants (subscriber and publisher).

**Table-2-5- Characteristics of Push/Pull Reliable Message Delivery in Publish/Subscribe [Cao *et al*, 2002]**

| **Characteristics** | **Value** |
|---|---|
| Guaranteed message delivery | Reliable message delivery |
| Context-awareness | Not supported |
| Real-time message delivery | Uses timeframes to determine time of delivery |
| Persistent Connections | Failure or link breakage cannot be detected |
| Is the subscription durable or non-durable? | Non durable – drops content for unreachable subscribers |
| Message queuing and policies | Uses a send and wait approach- wait for an Acknowledgement |
| Any use of a monitoring service for continuous subscriber disconnections? | No |
| Specification of the service to handle disconnected subscriptions? | No |
| Accessing Past Notifications | New subscriptions wait for new publications |

**iv. XNET: A Reliable Content-Based Publish/Subscribe System (Chand *et al*, 2004)**

This work is based on ensuring recovery after failures, and also ensuring reliability of events published. Chand and Felber (2004) propose several approaches to fault tolerance so that a system can recover from various types of router and link failures. They also ensure that the shared state of the system (*i.e. all registered subscriptions*) is consistent with the actual consumer population at all times. This work ignores the fact that frequently sent events or a subscriber's connection and disconnection from the system might cause congestion to the network, and also causing overhead in which information has to be kept in persistent storage and later be retrieved to ensure message delivery.

Therefore the fact that subscribers connect and disconnect from the brokers cannot be ignored and needs to be looked at in terms of the number of subscribers reconnecting at the same time, and also message or event delays during notification. The message delay increases with the subscriber population, this increase prove to be the result of subscriber disconnection ignorance. Chand, and Felber (2004) consider reliability of less important because undelivered messages have no impact on the consistency of the content routing system.

The authors propose the use of Crash/Recover and Crash/Fail-over schemes to deal with link failures. The Crash/Recover scheme requires the implementation of several databases at each broker to deal with lost messages or events, which causes disturbance at the brokers, and this might results in lost messages. ACK messages are considered to be a mechanism to ensure reliable delivery, the authors are not considering a fact that an ACK message when used in a distributed system is delayed. This might cause the system to

frequently retransmit events time and again because an ACK message might have been delayed or discarded due to network congestion or message overhead in which the system might not adapt to the increasing messages which are sent and the ACK message flowing on the network. The Crash/Fail-over mechanism requires the subscriber to wait for a confirmation messages that the subscriber's information is being redirected to another broker which the subscriber migrates to, since there in a distributed environment in a publish/subscribe scenario subscriber connect and reconnects to different brokers at different times. This reconnection might cause a subscriber to wait indefinitely for confirmation messages which might have been delayed due to messages being exchanged on the network.

**Table-2-6- Characteristics of XNET System in Publish/Subscribe**

**[Chand and Felber, 2004]**

| **Characteristics** | **Value** |
|---|---|
| Guaranteed message delivery | Reliable message delivery |
| Context-awareness | Not supported |
| Real-time message delivery | Timely delivery |
| Persistent Connections | Crash/Recover and Crash/Fail-over scheme |
| Is the subscription durable or non-durable? | Durable |
| Message queuing and policies | Messages are queued for recovery |
| Any use of a monitoring service for continuous subscriber disconnections? | No |

| | |
|---|---|
| Specification of the service to handle disconnected subscriptions? | No |
| Accessing Past Notifications | Events are stored on a Server for delivery |

## 2.4 Surveying of Real Implementation of Publish/Subscribe Systems

In this section, we specifically deal with real implementations of publish/subscribe systems. We take into account in details the most popular publish/subscribe systems, in particular by specifying their characterizing features with respect to the general solutions presented above. Each system will be positioned, according to the classification framework defined above.

### 2.4.1 TIB/RV

TIB/RV [Oki *et al*, 1993] is one of the first commercial systems to implement the publish/subscribe paradigm. TIB/RV is a topic-based system that relies on the abstraction of an event channel ideally connecting all subscribers interested in the same topic. In TIB/RV, brokers are structured in a two-level hierarchical architecture. Brokers at the lowest level of the hierarchy are called rendezvous daemon. Each network host on which a publisher or a subscriber resides has to run daemons. Subscriptions are assigned at this level following an ADA approach: each daemon holds the subscriptions for the host on which it runs. Events are diffused through network-level broadcast.

Event diffusion spanning a Wide Area Network (WAN) is realized through the other type of brokers, namely rendezvous router daemons, constituting the higher level of broker hierarchy. Each local network is represented at wide-area level by a single router daemon that receives all

the events directed from the local to other networks and multicasts in the local network notifications received from other networks. Router daemons form an application-level network, in which daemons are connected in couples through Transmission Control Protocol (TCP) connections. Event routing is realized by building multicast trees among router daemons. Each daemon maintains a tree for each subject. A router daemon is added to a tree if there exists at least a subscriber for that subject in the local network represented by that daemon. Since, in order to build trees, daemons have to know both the entire network topology and the current subscription configuration, we can classify the mechanisms used for wide-area diffusion in TIB/RV as a no assignment policy.

### 2.4.2 Scribe

An alternative approach for topic-based systems is the one proposed in Scribe [Castro *et al* 2002], a research system designed at Microsoft Research. Scribe is built upon an overlay network infrastructure called Pastry [Rowstron, Druschel 2001]. Pastry allows for efficient large-scale routing of messages in an application-level network of brokers. Each broker in Pastry is assigned a unique identifier in the network and messages can be routed to a specific broker by simply specifying its identifier. Scribe is actually an application written using Pastry and represents a P/S interface for it. Subscription routing, event routing and notification routing are implemented in Scribe by leveraging Pastry's direct routing capabilities. Scribe is a nice example of a system adopting a pure FDA approach. In the following, we explain how the assignment policy is implemented. The basic idea is that each topic is assigned a random identifier and the Pastry node with the identifier closest to the topic becomes the target broker for that topic. A multicast tree is built for each topic, rooted at the corresponding target broker.

When a new node subscribes for a subject, its subscription is routed by Pastry to the corresponding target broker that updates the tree structure in order to include the new subscriber. When an event is published for a subject, event routing is performed through Pastry, by directly routing the event to the target broker for that subject. The target broker is simply addressed by the subject's identifier. When an event arrives at the target broker, matching reduces to identifying the correct multicast tree and notification routing is performed by diffusing the notification through such tree.

### 2.4.3 Gryphon

Gryphon is a content-based system, developed at the IBM Watson research center. Besides being a real implemented system, Gryphon represents the reference framework for all the research in content-based P/S carried out at IBM Watson. Relevant research results include a matching algorithm with sub-linear complexity [Aguilera *et al*, 1999], an efficient event routing protocol performing partial matching at each broker [Banavar *et al*, 1999], and a routing protocol that satisfies exactly-once message delivery [Bhola *et al*, 2002]. Not all the results in these papers are implemented in the actual system[1]. However, they represent important steps in the evolution of the research in pub/sub systems. We then, base our analysis and classification of the Gryphon system upon these papers.

The idea behind the content-based multicast algorithm presented in [Banavar *et al* 1999] is to realize a distributed version of the matching algorithm presented in [Aguilera *et al* 1999]. It is basically a testing network algorithm realized using a tree data structure. In the distributed version, the tree spans all the brokers and the matching of a single constraint is performed at

[1] see the Gryphon web site (http://www.research.ibm.com/gryphon/) for details

each routing step. This original and effective idea allows for very good performance results. However, the algorithm relies on a very strong assumption: in order to build and keep the diffusion tree updated, each subscription update occurring at a broker must be reflected in the whole system. As we already pointed out, subscription flooding can be very harmful for the overall performance of the system. A revised version of the Gryphon multicast algorithm is presented in [Bhola *et al*, 2002]. Here the focus is on addressing reliable delivery of subscriptions, by realizing fault-tolerant broker architecture. Brokers are organized in a tree structure, rooted at publishers and with subscribers on the leaves. Reliability is addressed in two directions: subscriptions are partially replicated to manage faults of access points, while during event diffusion each broker keeps track of message loss. Finally, we cite the work of another part of the Gryphon team, devoted to the research in the application of network-level multicast. Relevant results have been produced also in this area, presented in [Opyrchal *et al* 2000] [Riabov *et al* 2002] [Riabov *et al* 2003] that we already commented above.

### 2.4.4 SIENA

Another important contribution to the research in content-based pub/sub is the SIENA system [SIENA]. SIENA focuses on providing efficient and scalable notification routing over a wide-area network. Brokers communicate exclusively through application-level connections without exploiting either network-level multicast or overlay network infrastructures. The assignment approach followed by SIENA is ADA. However, it lacks an explicit addressing mechanism for brokers such as in Scribe and does not use multicast for event diffusion. One main focus of SIENA event routing algorithm is to avoid flooding events blindly over the entire network.

The idea behind SIENA's routing algorithms is to build logical paths for events from all possible publishers to all subscribers. Paths are built with a subscription routing process, which for each subscription present in the system, creates a diffusion tree spanning all brokers, so that each broker knows in which direction it has to route the event in order to reach matching subscribers. This mechanism allows the pruning of all the parts from the event routing process to the parts of the broker's network that does not contain subscriptions matching that event. Subscription propagation is constrained by exploiting a containment relationship: informally, a subscription contains another subscription if all events matching also match. When a subscription update occurs, the new subscription is not propagated by a broker if this broker has already propagated a containing subscription before. Another technique to aid the event routing process introduced in SIENA is the advertisement. An advertisement is issued by a publisher to declare the set of events it is going to produce. Advertisements are also considered in building routing paths, to further reduce the set of involved brokers. With respect to a pure ADA approach, SIENA algorithm increases subscription redundancy in order to create efficient routing paths for events, but a complete replication is necessary only for most general subscriptions. The SIENA algorithms have become a reference solution for the problem of routing content-based events and subscriptions in an application-level network (content-based routing problem). In [Muhl 2002], a general theoretical framework for content-based routing is proposed as well as some variants over the original SIENA algorithm and a performance evaluation.

### 2.4.5 Hermes

Hermes [Pietzuch, Bacon 2003] is one of the most recent proposals in the pub/sub research. It is presented as a pub/sub middleware rather than a simple system, because it encompasses also other functions such as type checking and security. In the context of our analysis/framework, Hermes gathers in an interesting way several ideas from the systems described above. Hermes is a system based on a FDA approach, with a type-based subscription model. It differs from Scribe in that it is implemented as a network of brokers rather than exploiting an overlay network infrastructure. The FDA policy is realized considering the type of a subscription. A subscription is assigned to a broker whose identifier matches the hash of the type name. The FDA policy organizes subscriptions in coarse-grained clusters (types) and it may happen that the system contains more brokers than types, leaving some brokers not assigned to any type. At the same time, differently from SIENA, only a single copy of each subscription is retained in the system, at the corresponding target broker. Event and notification routing are implemented with SIENA-like algorithms. Paths are created for routing events belonging to a specific type to the corresponding target broker. Notification routing follows the same idea, with a diffusion tree for each type, rooted at the target broker and having each subscriber as a leaf. Content-based filtering of subscription is performed directly at recipients. Thus in Hermes, there is only one spanning tree for each type, whereas in SIENA subscription routing builds in practice a spanning tree from each possible publisher to all subscribers.

## 2.5 Minimal Message delivery delays and Handling message overload due to congestion

The following section details the framework used to analyze the literature that has accumulated in ensuring minimal delays of events, and also ensuring reduced or no message overhead.

### 2.5.1 Ensuring minimal event delays and handling message overhead in publish/subscribe environments.

The following has been identified as the cause of message loss and duplications in publish/subscribe: delivery delay, message congestion, reliability and message duplication.

#### 2.5.1.1 Delivery delay

The message delay might be minimum or maximum. A minimum delay might be as a result of low data flowing between the publisher, middleware, and subscriber sites, while maximum delay is caused by a large data availability resulting in message delay due to the overhead.

#### 2.5.1.2 Message congestion

Congestion in this case results from high data availability at the broker and the system cannot afford to carry all the data at the same time. The message congestion happens as a result of continuous publication of messages to the middleware and it becomes difficult to handle because some messages are sent more than once. Other systems might prefer a once-off delivery mechanism or several message delivery attempts to ensure successful delivery.

#### 2.5.1.3 Reliability

The number of messages sent and received determines the delivery reliability of a system. A system is considered reliable if it can deliver 100% of all published messages or events to an intended subscriber without loss of any message or a part of the message becoming discarded. The time of message publication and delivery also determines the reliability of a system. If a published message or event is required to be delivered within a specific time, that message must be delivered to the subscriber within that time frame for the system to be reliable and trusted.

#### 2.5.1.4 Message duplication

Filtering of message is required to ensure one copy is delivered. A message being sent more than once should be detected at the middleware or subscriber broker before the actual delivery to intended subscriber.

### 2.5.2 Review of Message Delivery delays and duplication in Publish/Subscribe issues in a Mobile Environment.

We now review existing work in P/S which support mobile clients and manage delivery issues:

#### (i) Supporting Mobile Clients in Publish/Subscribe Systems (Wang *et al*, 2005)

With the increasing popularity of wireless communication networks and mobile handheld devices, there is a pressing need to extend traditional distributed applications to mobile

environments. The authors propose a novel solution to support mobile clients in publish/subscribe systems and use a two-phase handover (2PH) protocol for managing delivery. In comparison with existing solutions, their solution can guarantee the exactly-once and ordered event delivery to mobile clients with very low cost, but the concern is reconnection, since it requires a mobile clients to wait for a long time to get the undelivered events upon reconnection. Concurrent movement of multiple clients is considered and mechanisms are provided to resolve the possible conflicts during the handover processes.

Another concern of the work by Wang *et al*, 2005 is that when an event is published and forwarded to a broker, the brokers tend to take too long handing over the events to the relevant location of the subscriber. This handover causes message delay causing it to reach the subscriber after the specified time of delivery. Since it does not support duplicates, a message is sent and assumed to be successfully delivered to a subscriber but may not be delivered especially when subscriber change location due to migration.

**Table-2-7- Characteristics of Mobile Clients in Publish/Subscribe System [Wang *et al*, 2005]**

| **Characteristics** | **Value** |
|---|---|
| **Delivery delay** | Maximum delay |
| **Message congestion** | Message overload at the Brokers |
| **Reliability** | Not fully reliable – message might be lost |
| **Message duplication** | Exactly-once message delivery mechanism |

### ii. Exactly-once Delivery in a Content-based Publish-Subscribe System (Bhola *et al*, 2002)

The authors present general knowledge model for propagating information in a content-based publish-subscribe system. The model is used to derive an efficient and scalable protocol for exactly-once delivery to large numbers (tens of thousands per broker) of content-based subscribers in either publisher order or uniform total order. The protocol they proposed allows intermediate content filtering at each hop, but requires persistent storage only at the publishing site. It is tolerant of message drops, message reordering, node failures, and link failures, and maintains only "soft" state at intermediate nodes.

Bhola *et al*, 2002 discuss the guaranteed delivery service of the Gryphon system. Gryphon is a scalable, wide-area content-based publish-subscribe system, employing a redundant overlay network of brokers [Aguilera *et al*, 1999] [Banavar *et al*, 1999].

A guaranteed delivery service provides exactly-once delivery of messages to subscribers. Each publisher is the source of an ordered event stream. A subscriber who remains connected to the system is guaranteed a gapless ordered filtered subsequence of this stream. A filtered subsequence is gapless if, for any two adjacent events in this subsequence, no event occurring between these events in the original stream matches the subscriber's filter. The guarantee is honored as long as the subscriber remains connected, even in the presence of intermediate broker and link failures.

The main contributions of this work are:

The introduction of knowledge graph abstraction that models propagation of knowledge from publishers to subscribers through filter and merge operations, and propagation of demands for knowledge in the reverse direction. A protocol based on this knowledge model that tolerates broker crashes and dropped / re-ordered messages, that does not

require hop-by-hop reliability of messages, and that requires stable storage only at the publishing broker and only soft-state everywhere else.

However, Bhola *et al*, 2002 does not focus on disconnected subscriptions which tend to receive dated messages during reconnection. Based on the fact that dated messages are not discarded and removed from the storage this tend to disrupt the dissemination of new incoming events which have high priority for delivery. Therefore there is a need to thoroughly look at mechanisms to ensure outdated messages are discarded and removed.

**Table-2-8- Characteristics of Content-based Exactly-Once Delivery**

***[Bhola et al, 2002]***

| **Characteristics** | **Value** |
|---|---|
| **Delivery delay** | Maximum delay |
| **Message congestion** | Outdated messages are not discarded |
| **Reliability** | Messages might be lost due to congestion |
| **Message duplication** | Exactly-once message delivery mechanism |

### iii. Publish/Subscribe Systems on Node and Link Error Prone in Mobile Environments (Oh *et al*, 2005)

Mobile and ubiquitous environments are prone to errors due to wireless link disconnection, power exhaustion on mobile devices, etc. Oh *et al*, 2005 analyze performance and effectiveness of publish/subscribe systems when confronted with the failure of client and server nodes and disconnection of communication links, which is common in mobile environments.

Thus, it is essential to consider how the failure and unavailability of mobile devices and wireless networks affect performance and the use of the service. The authors analyze the influence of errors on mobile devices, servers, and wireless links, and compare them to other interaction-based models such as client-server model and polling models, and also estimate performance in error prone environment and effectively adopt publish/subscribe systems by using their analysis. The results of Oh *et al*, 2005 show that publish/subscribe systems are more durable than client/server models in error-prone mobile and ubiquitous environments.

*Failure of client (subscriber disconnection)*: After a client recovers from failure, client can obtain any event at anytime if durable database exists, which logs data (or events) to be used by client after recovery from failure. However, the size of the durable database is limited. If the durable database can have a maximum of $n$ data (events), a client can be provided up to $n$ recent events which occur before recovery. However, data is not available to the client after its recovery from a failure when the client requires more prior event than $n$ recent event.

The authors [Oh *et al*, 2005] ignore the message delay time between failure recovery and notification, and this cannot always be ignored in a mobile environment where subscribers migrate from one broker to the next. The authors [Oh *et al*, 2005] mentioned that subscribers after successful reconnection subscribers might have to wait for too long before receiving events published during their disconnection time (offline time). A mechanism or service to ensure immediate event delivery to reconnected subscribers is required to allow subscribers to receive all events published during their offline time, so as to reduce or eliminate the delivery delay.

Table-2-9- Characteristics of Node and Link Error Prone Mobile Environment [Oh *et al*, 2005]

| Characteristics | Value |
|---|---|
| **Delivery delay** | Maximum delay |
| **Message congestion** | During recovery from failure |
| **Reliability** | Reliable for connected subscriptions |
| **Message duplication** | Possible duplications due to delays |

### iv. Handling Overload in Publish/Subscribe Systems (Jerzak and Fetzer 2006)

A new approach for handling overload in Publish/Subscribe systems is presented in this work [Jerzak and Fetzer, 2006]. The authors [Jerzak and Fetzer, 2006] addresses the issues of how to cope with (1) with the limitations imposed by the external environment (e.g., network congestion or link failures) and (2) the limitations resulting directly from within the service itself (e.g., high message load). These two issues are handled in a graceful way while ensuring that the most valuable information is given the highest chance of a successful delivery.

Link and message shedding mechanisms are proposed which require routers to be able to asses the profitability of both links and messages they manage/forward. The sole purpose of the introduced charges is to allow the router to decide which message has the highest or lowest price and hence decide about shedding it or forwarding. This is a non-trivial task, especially if we keep in mind the fact that the P/S communication scheme is fully decoupled and there can be potentially hundreds of subscribers interested in the given message that are unknown to the router and vice versa. For link shedding, the selection of

the link is done according to the sum of the advertised maximum prices on the stored subscriptions, while Message shedding is a more fine grained mechanism requiring additional algorithms and data structures built into the HAPS (Highly-Available Publish/Subscribe system) router.

The authors [Jerzak and Fetzer, 2006] ignored the delivery time of a message, the message is delayed since the system checks for previous failures that occurred during notification. A published message is placed into a path with the best minimal failure rate, the time it takes to determine the failure rate, and delivery time shows that even a published event that was supposed to be delivered within a specified time is not assured to be delivered in time due to message shedding and also determining previous path failures.

**Table-2-10- Characteristics of Overload in Publish/Subscribe System [Jerzak and Fetzer, 2006]**

| Characteristics | Value |
|---|---|
| **Delivery delay** | Maximum delay |
| **Message congestion** | Message Shedding |
| **Reliability** | Not fully reliable – message might be lost |
| **Message duplication** | Not defined |

### 2.5.3 Summary of the Delivery Semantics Associated with Events for Quality of Service

### 2.5.3.1 Delivery semantics

The first kind of characteristics are the delivery semantics associated with events; an expression of quality of delivery.

*Unreliable*: When such an event is published, there is no guarantee that it will be received by any subscriber. There is only a best-effort attempt to deliver it. This is assumed by default.

*Reliable*: Once successfully published, a reliable event will be received by any subscriber that is "up for long enough". A subscriber which does not halt (whether prematurely by failure or intentionally) will eventually deliver every such event. *Certified*: With such events, even if a subscriber temporarily disconnects or fails, it will eventually deliver the event.

*Totally ordered*: Events can furthermore, be notified in a total order to the subscribers: e.g., two subscribers' s1 and s2 which deliver two events e1 and e2 both deliver e1 and e2 in the same order (we also term this subscriber-side order).

*FIFO ordered*: Two events e1 and e2 that are published through the same object are delivered to all objects whose subscription matches both e1 and e2, and in the same order they were published (publisher-side order).

*Causally ordered*: These types of events are delivered in the order in which they are published.

### 2.5.3.2 Transmission semantics

Further semantics, called transmission semantics, can be associated to events. These govern the handling of events when they are in transit, also with respect to other events.

*Priority:* Events can have priorities, that is, the delivery of an event can be delayed due to another event with higher priority.

*Timely:* Similarly, certain events might "expire". In other terms, events can become obsolete and should be discarded.

These different semantics are not all mutually exclusive. For instance, events can be certified and have some notion of priority, or be certified and totally ordered at the same time. It appears that contradictions reside for instance between reliable and simultaneously timely limited events, as well as between orders (total, FIFO, or causal order) and priorities. In the above cases, the first type takes precedence. Figure 5.4 illustrates the dependencies between the different semantics. Note that for any kind of order expressed by an event type, its instances satisfy that order with respect to instances of the same type, its subtypes, and super types with that same order only.

Next, we give an overview for supporting mobility in publish/subscribe systems. Each publish/subscribe system is reviewed to determine if it supports mobility.

## 2.5.4 Mobility Support in Publish/Subscribe Systems

Most of the existing publish/subscribe systems have been designed and optimized for stationary environments where publishers and subscribers are static, and the

infrastructure itself stays fixed. The mobility-related operation is dealt with at the application layer through a sequence of subscribe requests: A subscriber from the application layer first defines new subscriptions and unsubscribe prior to disconnecting from the publish/subscribe system. After reconnecting to the system, the subscriber needs to re-subscribe to make the system aware of its subscriptions. However, the subscriber will not receive notifications that have been published during the time of disconnection. It is argued in [Podnar *et al*, 2002] that the publish/subscribe middleware itself must offer the mobility support by ensuring seamless reconnection to a new broker and by preserving notifications published during disconnection. The authors in [Zeidler *et al*, 2003] agree that mobility-related issues should be addressed by the publish/subscribe middleware, and not delegated to the application layer. Some publish/subscribe systems (Table-2-11) incorporate solutions to the problem of client mobility: The common solution stores notifications published during disconnection in a special subscriber queue and deliver the notifications after subscriber reconnection. The existing solutions extend the established stationary publish/subscribe systems (Table-2-11) to cope with client mobility while keeping the infrastructure stationary [Caporuscio *et al*, 2003; Fiege *et al*, 2003]. The position paper [Huang *et al*, 2001] takes an orthogonal approach. The authors analyze the requirements of mobile publish/subscribe systems, and discuss centralized and distributed system architectures tailored to mobile environments.

Based on our view of the common solutions to store notifications published during disconnection in a special subscriber queue we differ in a way to avoid deadlock and starvation by exploring the use of a durable database which only keep up to date events and discards outdated events to avoid long waiting by subscribers to receive events

missed during their disconnection period. Our approach of adopting the durable database ensures reliable delivery and non deadlock processes which is further discussed in section 5 of our results evaluation.

| | CORBA event service | CORBA notification service | JMS | TIB/ Rendezvous | JEDI | Siena | DACS | Hermes | REBECA |
|---|---|---|---|---|---|---|---|---|---|
| subscription criteria | channel-based | content-based | content-based | subject-based | content-based | content-based | type-based | type/attribute-based | content-based |
| notification structure | typed Any | structured messages | structured messages | set of typed attributes | set of untyped attributes | set of typed attributes | Java objects | instances of an event type | set of attributes |
| filtering criteria | not specified | constraint language | message selectors | not specified | regular expressions | constraint language | structural reflection | filter expressions | compound expressions |
| notification persistency | no | yes | yes | no | no | no | no | no | yes |
| architecture | not specified | not specified | centralized/ distributed implementations | distributed | distributed (hierarchical graph) | distributed (hierarchical/ acyclic graph) | distributed | distributed (general graph) | distributed (acyclic graph) |
| routing strategy | not specified | not specified | flooding | flooding | reverse path forwarding | reverse path forwarding | not specified | core-based trees | reverse path forwarding |

**Table-2-11- The requirements of mobile publish/subscribe systems**

**Mobility support in CEA:** The Cambridge Event Architecture (CEA) [Bacon *et al*, 2000] uses a mediator which receives notifications on behalf of a subscriber during disconnections. The mediator acts as a subscriber proxy, and can register interest in subscriber's location. When the subscriber reconnects to the system, the mediator will get a notification with the new subscriber's location, and then deliver the queued messages to the subscriber. The proposed solution is indeed interesting because it relies on the publish/subscribe infrastructure itself to transmit the information about the changing subscriber's locations. However, it involves a serious security risk. A malicious party could take the role of a mediator, track subscribers, thereby jeopardizing location privacy, and delivering bogus notifications after subscriber reconnection.

**Mobility support in JEDI.** JEDI [Cugola *et al*, 2001] offers two mobility-related operations: *moveIn*, and *moveOut*. A subscriber uses *moveOut* to disconnect from a broker and *moveIn* to reconnect possibly to a new broker. A client can detach from the system, serialize its current state, and later on reconnect. The old broker stores events on behalf of the subscriber during the disconnection period and transmits them to a new broker upon reconnection. The approach solves the queuing problem, however, no details regarding the handover procedure from the old to the new broker, or the change of the delivery path is given. Authors in [Cugola *et al*, 2001] propose a rather complex solution that updates the delivery tree in a hierarchical distributed architecture: It uses a dynamic dispatching tree that has a leader responsible for subscribers with the same subscription. This solution requires a complex protocol and further analysis is needed since no evaluation study is currently available.

**Mobility in Siena.** The authors of Siena present a support service for mobile, wireless clients of a distributed publish/subscribe system in [Caporuscio *et al*, 2003]. The mobility service enables the movement of subscribers between different access points of a publish/subscribe system. The service uses client proxies and a special client library to manage subscriptions and notifications on behalf of a subscriber while the subscriber is disconnected and during the handover between different access points. A client proxy runs as a special component at an access point and stores messages for a disconnected subscriber in a special queue. The client library mediates subscriptions, and initiates a move-out procedure: It submits subscriptions to the client proxy that subscribes using the client's subscriptions and stores incoming messages in a special buffer. The client uses the move-in function to reconnect to the system. It contacts the local client proxy and submits the address of the old proxy.

The old and the new proxy start a special handover procedure that transfers messages from the old proxy to the new one and then to the subscriber. The mobility service implements a special synchronization mechanism to avoid lost notifications. The main principle is quite simple: when transferring subscriptions from A to be active on B, the system needs to make sure that subscriptions are active on B before terminating subscriptions on A. It is possible that during the procedure both A and B will receive the same message. The mobility service implementation permits that a subscriber receives duplicate messages. The presented system is independent from the underlying publish/subscribe middleware. The client library wraps the target publish/subscribe API

and needs to be implemented specially for each API by adding the move-in and move-out functions, and by overriding the subscribe function of the original API.

Caporuscio *et al*, (2003) offers results of an experiment that proves the applicability of the implementation. The evaluation is limited since the experiment has been performed on a broker network consisting of three broker nodes, a single publisher, and a single mobile subscriber that moves only once. The experiment includes the performance evaluation if a subscriber uses a GPRS network - which has been simulated to access the publish/subscribe service.

**Mobility in REBECA.** The approach taken within the project REBECA is to extend and modify the existing publish/subscribe system to support mobile and location-dependent applications [Fiege *et al* 2003; Zeidler, *et al* 2003]. The mobility service aims to support two different types of mobility: physical mobility and logical mobility. Physical mobility is similar to terminal mobility. A client is physically mobile and roams between different network domains. It can disconnect from the system and later on reconnect possibly to another broker in a different network. Its subscriptions are valid and the system stores notifications published during the disconnected period. Logical mobility is related to geographical location. As a client changes its geographical position, its subscriptions dynamically change because the published information is location-dependent. The algorithm that is developed for physical mobility is designed for a distributed network of brokers. It applies the "queuing" approach. The old broker stores notifications for a disconnected subscriber. When the subscriber connects to a new broker, it re-issues its subscriptions, but keeps no record of the old broker address. The algorithm finds the old broker by locating a broker that is at the junction of delivery paths for the new and the old

broker. It is clear how this junction broker is found if simple routing is used. Each broker stores active subscriptions for all subscribers with the subscriber identifier, and since the subscription from the old broker is still active in the system, it is simple to find the junction broker leading to both the old and the new broker. The notifications stored by the old broker are routed through the junction to the new broker and delivered to the subscriber. With simple routing the routing tables can become rather large because all brokers have the knowledge about all subscriptions. Routing algorithms that use covering and merging are better suited for mobile environments where subscriptions change more often than in static scenarios. The proposed algorithm needs further extensions in case routing based on covering or merging is applied since the process of finding a broker junction is not straightforward. The designed algorithm appears to be rather complex and there are currently no evaluation results that shows its applicability and performance.

**Mobility in Elvin.** Mobility support for Elvin [Carzaniga, 2000] is one of the first implementations offering mobility to subscribers in a publish/subscribe system. It enables subscriber's nomadic mobility without modifying the original Elvin server. The proposed solution puts a proxy server between the original Elvin server and a mobile device. The central proxy server queues messages for disconnected subscribers and delivers them upon their reconnection. The presented solution implements a queuing strategy with a time-to-live expiry. A subscriber must always connect to the central proxy server which can become a performance bottleneck and induce significant network traffic due to triangular routing if a subscriber connects to the system from another network.

**JMS-based systems supporting mobility:** Recently, some of the systems that implement the JMS specification offer support for mobility [Softwired, 2002; ObjectWeb, 2003;

Yoneki *et al*, 2003]. Such systems offer a lightweight JMS-compliant API for Java-enabled mobile terminals that can be used to implement JMS-based publishers and subscribers. iBus//Mobile [Softwired, 2002] is a commercial MS-compliant implementation. It integrates a special gateway that serves as a mediator between a JMS provider, and JMS clients. It offers support for native clients with no JMS support. Native clients can publish and receive SMS or MMS messages that are transformed into JMS messages that can interact with the JMS provider. iBus//Mobile supports TCP, UDP, HTTP, and HTTPS as transport protocols for JMS messages. JORAM [Yoneki *et al*, 2003] is an open source project that has recently published a client API called kJORAM that adjusted to J2ME devices. Pronto [Yoneki *et al*, 2003] is an academic project. It provides a JMS-compliant middleware system that supports mobility of JMS publishers and subscribers and implements a mobile JMS API that can run on resource-limited devices. It incorporates a mobile gateway that supports JMS in wireless networks and employs SMS, or mail as transport mechanisms for native devices that do not support Java and JMS.

### 2.5.5 Requirements of Mobile Clients in a Mobile Environment for Message Consumption

Mobile clients have specific requirements due to their scarce resources and temporary connectivity. Usually, mobile clients connect over wireless links, which are especially susceptible to overloading due to their restricted bandwidth. As they move, mobile clients can connect to different access points using various networking technologies (Bluetooth, Wireless LAN, and so on). Therefore, continuous information delivery requires seamless handover. Moreover, both processing and wireless networking consume considerable

electric power and available battery capacity probably represents the greatest limiting factor in mobile devices today. Consequently, mobile clients might get overloaded if the infrastructure delivers all occurring updates to them. This is particularly true for a periodic push, in which the load corresponds to the rate at which the information changes. Therefore, we decided to reduce the load by reducing the number of update messages mobile clients must deal with. In our approach, the infrastructure propagates only those update messages necessary to fulfil consumers' specific requirements. This includes only pushing data to the consumer if it sufficiently deviates from the data received previously. The infrastructure can save a lot of resources by doing so, because consumers usually do not depend on receiving all intermediate updates. For example, consumers should receive a new stock quote only if they received the last update at least 10 seconds before, or if the price changed by at least two dollars. Using Rebeca's content-based subscription mechanism, a client has full control over the notifications it will receive. Clients use the same mechanism to specify, for example, constraints on notification frequency. The Rebeca infrastructure uses this information to avoid propagating notifications unnecessarily. Rebeca applies further policies when network congestion occurs. For example, if several updates have queued up at an infrastructure node, it forwards only the most recent update. The mechanisms Rebeca uses to deal with client mobility differ from previous ones in that the application does not control mobility; instead, the middleware handles it transparently.

Based on the review of related work discussed in this chapter we observe that there is still a lot of work to be done for message delivery management in publish/subscribe systems, especially in mobile environments where subscriber-context is an issue. Most of the work

discussed in this chapter still needs a service to discover or monitor frequent disconnection of subscribers, message delivery delays, the delivery of messages based on the priority preference, the delivery of stale/outdated messages to registered subscribers and guaranteed delivery of messages. In this work we develop a mechanism or architecture that guarantees real-time delivery of messages to subscribers based on the requirements the subscribers specify during subscriptions. We also propose modules to manage priority, matching management and also discovery of outdated messages which are discarded from the durable database. We also, during implementation, consider the issue of message duplication by implementing a service that does not duplicate a message to a subscriber. Most of the issues identified are implicit in the proposed architecture and on the issue of message delivery reliability we developed a simulator to evaluate message delivery.

This Chapter has brought together related work and we also constructed a theoretical framework in which we discussed and analyzed each work and also table the related work using different characteristics of publish/subscribe pattern. The next chapter proposes solutions to the problems discussed in this chapter and we also construct message delivery algorithms to ensure reliable and efficient delivery.

# CHAPTER THREE

# MODEL DESIGN

## 3.1 Introduction

The increasing popularity of information services that rely on content delivery in mobile environments motivates the need for mechanisms that guarantees real-time message delivery to mobile users in a specific time limit without loosing part of the message, and also avoids the issue of message duplications by also ensuring no messages are lost during dissemination. A service will be more reliable, efficient and scalable if it enables the delivery of personalized and customized content to mobile users timeously. Furthermore, it must monitor disconnected subscriptions in order to keep undelivered messages in persistent storage, in which later the message will be retrieved and forwarded to the relevant subscriber after successful reconnection. Subscribers define the characteristics of content that is of interest to them in order to receive notification when such content becomes available. We list and analyze the requirements of guaranteeing message delivery in a Publish/Subscribe mobile environment. The possible design principles that can be handled by our model are analyzed and defined in this chapter. Our new model should accommodate mobile devices that participate in a mobile environment to ensure reliable, efficient message delivery.

This chapter is organized as follows: Section 3.2 discusses message queues as it was used in our model for managing messages for delivery. Section 3.3 introduces our definition of

real-time delivery model and also defines their design goals. Section 3.4 defines requirements for guaranteeing message delivery in publish/subscribe systems for mobile environments. Finally, the remaining sections deal with the development of mechanisms to guarantee message delivery for mobile users in a publish/subscribe environment.

## 3.2 Message Queues

The most common alternative to pub/sub for realizing interactions with multiple recipients is the message queue paradigm. The Message Queue is an abstraction that is particularly used in the industry, with many popular existing implementations, such as IBM WebSphereMQ [IBM 2003], Microsoft Message Queue [Platt 2001] and part of the JMS specification [Sun Microsystems, 2003]. The reason is that the message queue paradigm can easily provide transactional or reliability guarantees, thanks to the fact that messages are persistently stored within the queues. Moreover, it is often used as the basis in asynchronous invocation of software components (such as COM+ Queued Components, Message-Driven Enterprise Java Beans or Web Services). All the communications in this paradigm are filtered by the queue that covers a role similar to the Notification Service in pub/sub. The difference is that each participant may have its own queue and a one-to-many interaction could require addressing several queues. Another feature that lowers the level of decoupling attained through message queue compared to the one provided by pub/sub is that consumers must explicitly pull messages from the queue. However, push-style callback is often present also with a one-to-many delivery, making the message queue paradigm similar to a persistent form of pub/sub.

## 3.3 Real-Time message delivery model

### 3.3.1 Design Goals

The main goal of this work is to set up/develop research mechanisms for message guaranteed delivery that meet a uniform deadline. The time to receive a message, after it has been sent, can be divided into two components: first, the time it takes the monitor service to request subscriber status and second the time at which the event arrives at the subscriber. The first component is the path connectivity delay and it depends on the physical distance between the source and the receiver, and the network conditions. The second component depends on the time of notification by the Event manager, and delivery to a subscriber, and it might also be determined by the speed of the Notifier service. The delay is fixed but different for each receiver. However, using the priority factor attached to events that are of higher priority can control the time deadline. Events of higher priority preempt events of lower priority. By this we mean that, if an event X has a smaller amount of time to be delivered to its destination than event Y, then, event X has a higher priority than event Y as well as other events queued for delivery. Intuitively, the farther away the receivers are in the priority list, the longer the waiting time. By implication, a soccer match update cannot have the same priority as the product on sale in a particular clothing store. Therefore during a soccer match, all other events with a lower priority will wait for higher priority events to be delivered to their destination before they are delivered. Furthermore, some events with shorter waiting time might end up being discarded because their time expired while still on the priority queue taking their turn.

There are two secondary goals (or design criteria) on which the ultimate design is based. These two design criteria are presented next.

- *Support for Scalability.* Possible source implosion situations must be minimized to allow the publish/subscribe applications scale well with large groups. Source implosion situations can happen if many receivers try to establish individual point to point communications with the source simultaneously, such as when the receivers reply to their reconnection status.
- *Enforcing fairness.* Heterogeneity in publish/subscribe applications implies that some receivers may be very close to the source and others very far, in terms of delay between the source and the receivers. Fairness means that the farthest receiver should be able to receive from sources at the same time as closer receivers. Therefore, the farthest receiver determines the value of the target delivery time.

## 3.4 Message Delivery Guarantee System Requirements

In order for Publish/Subscribe customers to receive event notification in a more concise manner than what is currently available, message delivery guarantees needs to be adopted in the publish/subscribe communication paradigm. This will allow customers to subscribe to the system and set their preferences during the activation of the subscription in order to be notified of changes to events relevant to their interest, and also to ensure delivery guarantees that messages will not be lost.

Message delivery guarantees to mobile device users must address challenges such as message duplications, message loss, and that useful messages do not get discarded. . Mobile device users will be able to enjoy information services on offer knowing that all

messages will be kept in a persistent storage when they are temporarily disconnected from the system anytime and anywhere. Message delivery guarantees should also consider people that are based in rural areas where disconnection is the order of the day due to poor signal strength. This category of people would be better served if access to information services is not denied them due to the unreliability of the cellular network.

Therefore, this dissertation reports a proposed publish/subscribe approach to message delivery in a mobile environment. The approach is then strengthened to guarantee messages are delivered in real-time unless the message expires before it is delivered.

The worldwide hunger for content information service thrives on the precondition that such system delivers only highly personalized and customized content in accordance with user preferences and current presence status. This may bring about the creation of a "branded" dissemination service that is invulnerable to spam. The service could become a trusted intermediary between content publishers and subscribers that filters information according to user's needs. The following design requirements have been identified for the proposed solution.

**Push-based content delivery:** Service users must be able to specify the type of content they want to receive, and be served with the published information as soon as it is available. The push-style content delivery eliminates the burden of querying for information at regular intervals and is in accordance with the stochastic nature of content creation and publication.

**Content filtering and personalization:** The Content filtering process relies on user subscriptions and minimizes the number of received message that are not of interest. This

feature requires that services are personalized and adapted to user context. The associated effect is that information overload on a user is reduced to the barest minimum.

**Scalability:** This requirement implies that service is optimized for the particular application area with respect to the number of publishers and subscribers in the system, and the size and frequency of published content.

**Message Priorities:** The published event is either classified as of high or low priority by the publisher of the event. A high priority event is an event that cannot be missed by a subscriber and event of this kind are stored and delivered to any subscription under it, but gets discarded after expiry. Low priority events are events which are of no importance to a subscriber and can be discarded at any time, especially when there is no subscription under it. We therefore conclude that starvation will not results in this manner as every subscription is treated independently and the parties involved (subscriber and events) will not lead to any deadlock as every subscriber receives only events he/she subscribed for. The starvation issue is also managed by the monitor service and event manager which only direct events to intended subscribers.

## 3.5 Message Delivery Guarantees in Publish/Subscribe Systems

Figure 3.1 shows the Publish/Subscribe Message Delivery Architecture designed to guarantee message delivery to subscribers in a mobile environment. The architecture employs the Recipient list pattern to carry out message delivery management such that a published message is matched against user subscriptions. This is done to filter out subscribers who are not interested in the publication. The recipient list stands for the list of subscribers registered to receive the published message.

The architecture (Figure 3.1) comprizes three sites namely, the publishing and subscription, message delivery management and notification site. These sites communicate to make message delivery a reality, and also manage message loss and duplications. The publishing and subscription site comprizes publishers (event producers), who produce and make events available to subscribers and subscribers who registers their intents to receive notifications for published events. The message delivery management site act as a mediator between publishers and subscribers by managing event flow from the publishing site to the subscriber site. The message delivery management site comprizes two modules namely, matching manager and guaranteed message delivery module (GMDM). The Matching manager module uses the published events and the subscribed preferences to match and filter out unwanted messages. The Matching manager module is described in greater details in Figures 3.9 and 3.10. GMDM guarantees real time message delivery to subscribers and also manages all undelivered events. The module is comprised of three services namely, an Event manager, a Monitor, and a Notifier service which interact to ensure reliable message delivery. GMDM is described in details in Figure 3.6 and 3.7.

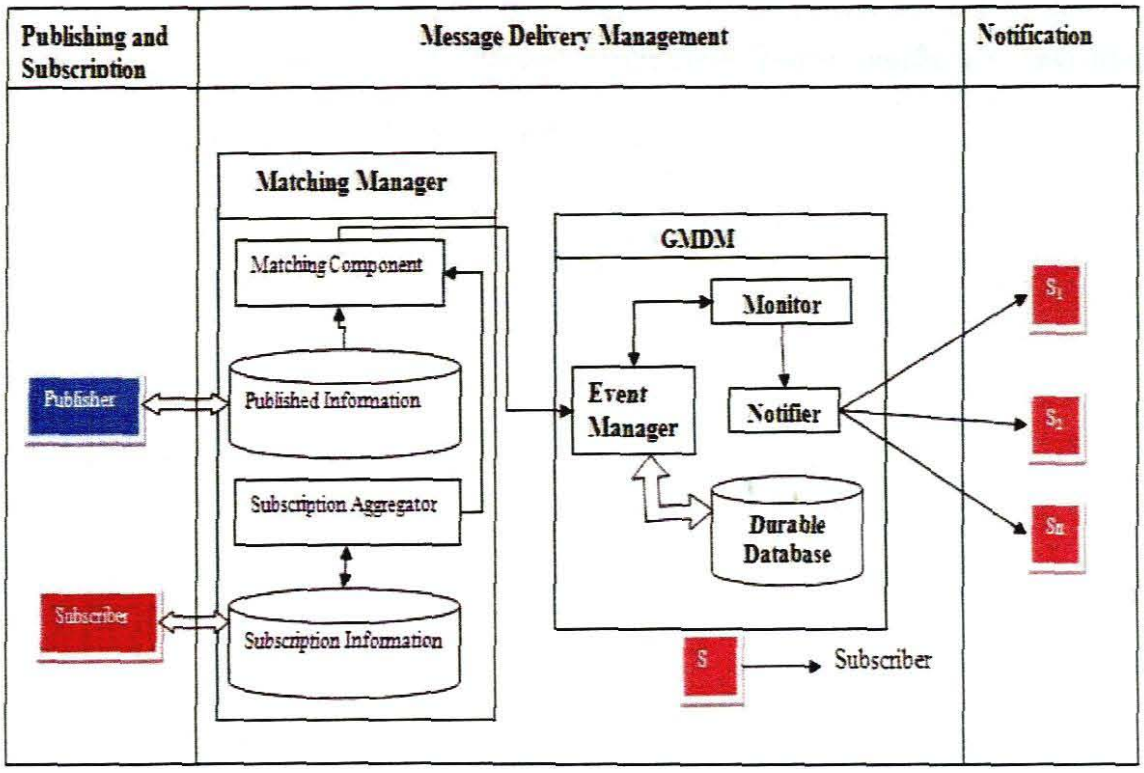



**Figure 3.1: Publish/Subscribe Message Delivery Architecture**

**SubscriberUI:** this class creates a subscriber profile during subscription and store the created profile to the subscription information.

**ProfileManager:** this class has a set method to prepare a subscriber's status, i.e. subscription status, its connections, also manages subscriptions. it also directly interact with the event manager.

**EventManager:** this class is an event manager class that manages all published events by ensuring that events are channeled to interested subscribers, and also ensures safe storage of undelivered events in a persistent storage.

**MonitorService:** this class is responsible for monitoring and keeping subscriber status on the network by triggering its movement and reporting back to the EventManager class.

**NotifierService:** this class accepts and disseminates events to subscribers, receives subscriber details such as Sub_Phone# (subscriber phone number) , Sub_Status (subscriber status) and the published event.

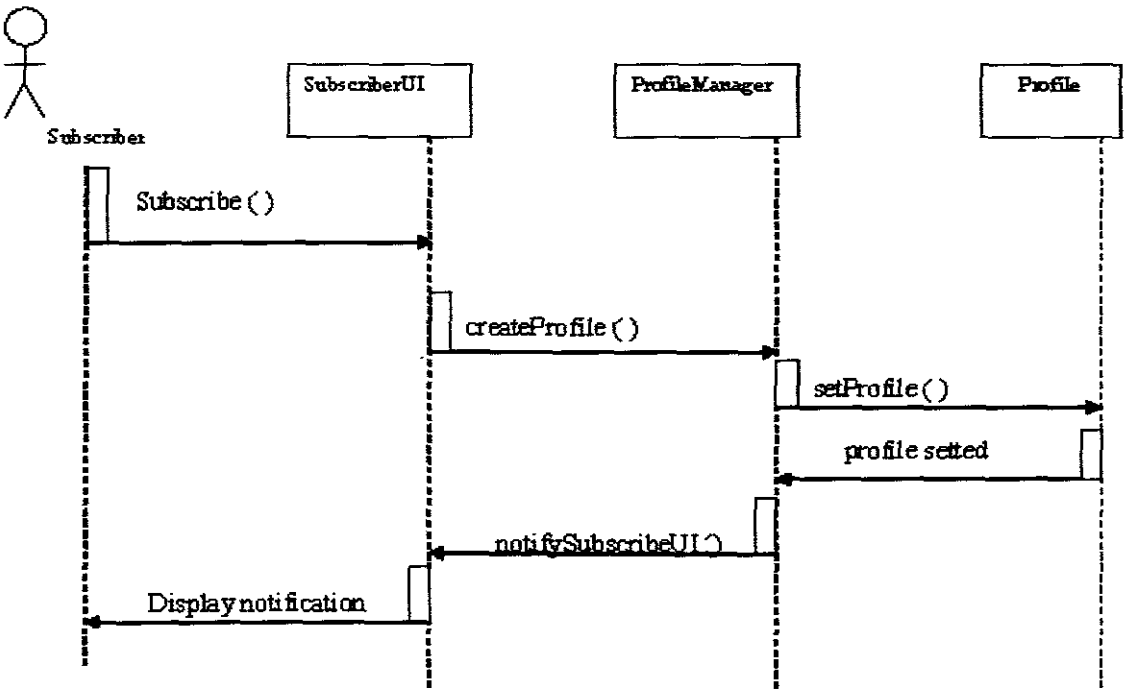



**Figure 3.2: Sequence diagram for registering a subscription**

Figure 3.2 illustrates how to register a subscription. For notifications to be published and delivered, the system has to acquire at least one registered subscriber to act as the information consumer of the published events. The subscriber uses a mobile device to subscribe to the system and his/her profile is created and stored in the subscription information database as shown in figure 3.1

In figure 3.1 during subscription the subscriber provides her personal details, preferences and also priority preferences for delivery purposes.

The message delivery sequence model at delivery time involves four participating objects, namely: the Event manager, the Monitor service, the Notifier service, and the Mobile device. The Event manager receives as input, subscriber information and publication information from the Matching manager module architecture (Fig.3.9). At the time of receiving the input, the manager requests subscriber's connection status from the Monitor service which is a service that keeps track of published information waiting for guaranteed delivery, and also to avoid network congestion and traffic. The Monitor service uses the information provided by the Event manager to determine the subscriber status, which it reports back to the Event manager which then reacts according to the response. In Figure 3.3, the response from the Monitor service is active, which informs the Event manager that the mobile subscriber is active. The Event manager sends the event information containing the destination ID, and the publication information to the Notifier service being the service that ensures a message is delivered to the destination, which in our case is the Mobile device.

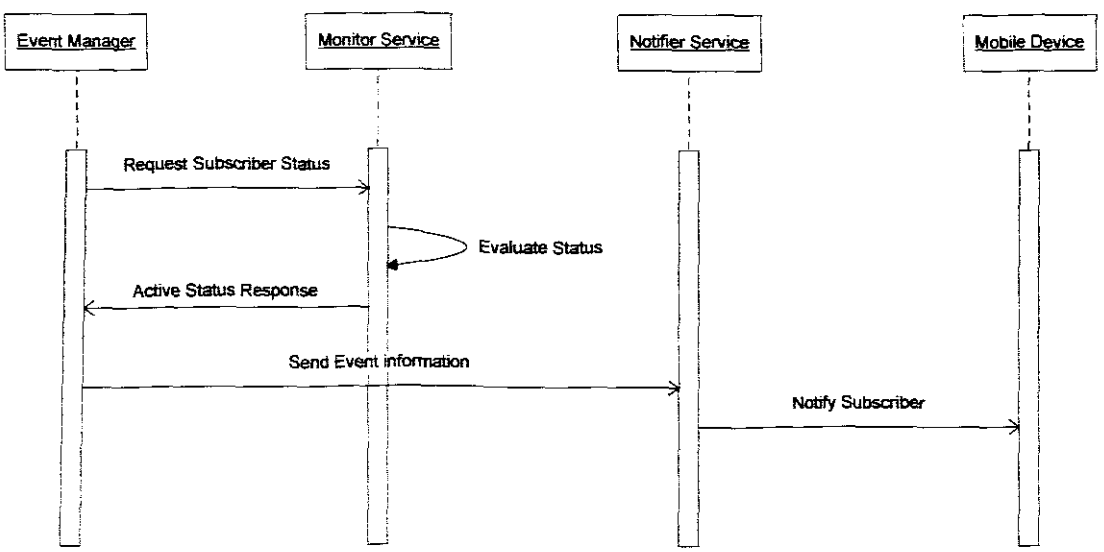



**Figure 3.3: Sequence Diagram for Successful delivery**

A problem which may arise during notification period by the Notifier service is the runtime disconnection. The runtime disconnection in our knowledge has not been looked at in publish/subscribe paradigm, and in our case the Notifier service acts as both the message sender service, and the emergency service.

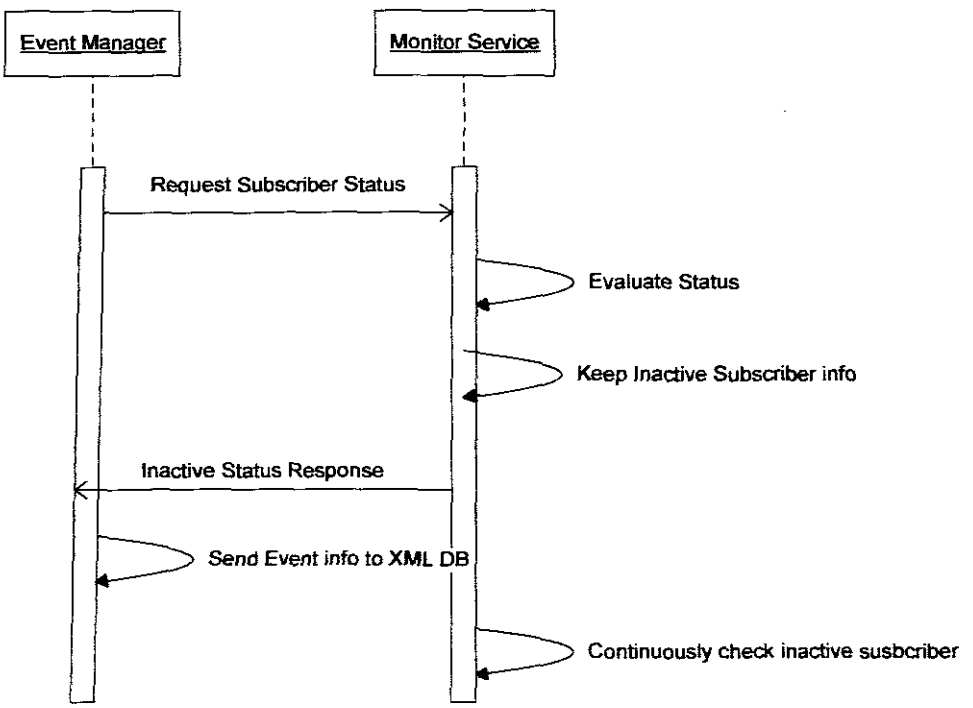



**Figure 3.4: Sequence Diagram for Unsuccessful delivery**

First, it acts as the message sender illustrated in Figure 3.4 by notifying a subscriber of published events. Second, it also acts as the emergency service that handles runtime disconnection by detecting the disconnection, then notifies the event manager which stores the published information with the destination ID in the XML database. Finally, Notifier informs the Monitor service about which disconnected service to continuously monitor. In the event that a subscriber's status is detected to be inactive (Figure 3.4) the Monitor service keeps the subscriber's information for continuous monitoring.

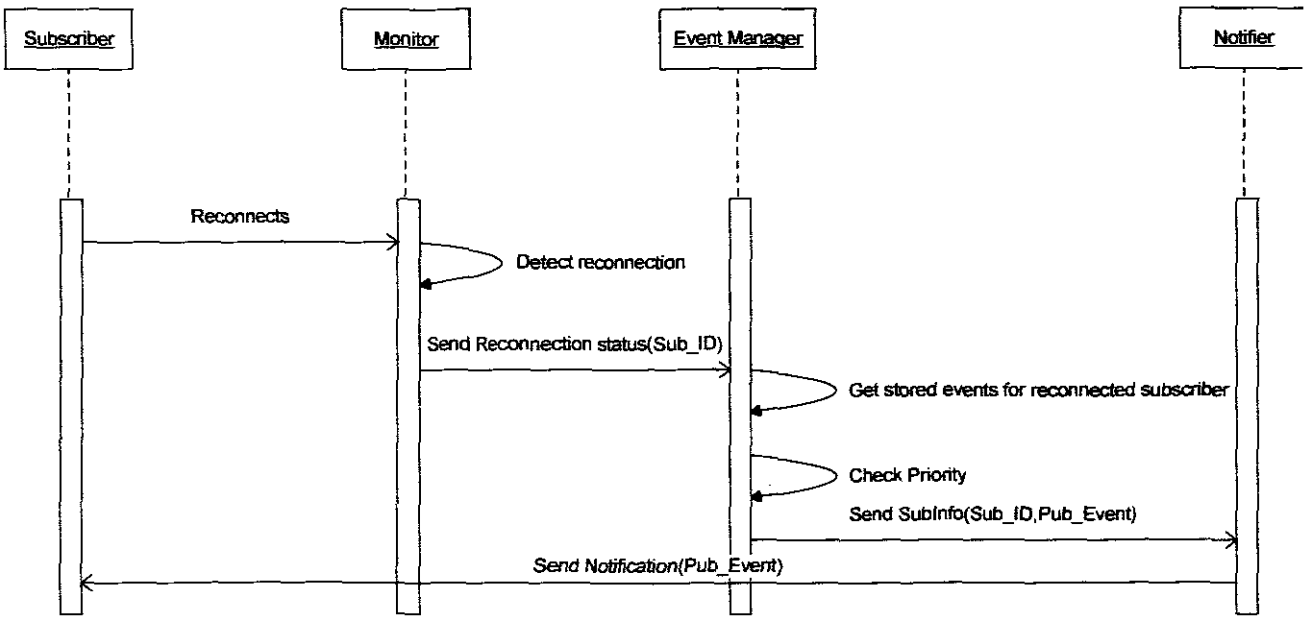



***Figure 3.5: Sequence Diagram for Subscriber reconnection***

Figure 3.5 shows the sequence diagram of a subscriber reconnecting to the network after it previously disconnected. When the subscriber's connection signal to the network becomes strong or when a subscriber switch on his/her mobile device it triggers the monitor service which detects the reconnecting subscriber and notifies the event manager by sending the reconnection status information such as sub_ID (subscriber identification number). The event manager will then get stored events for the reconnected subscriber and also checks for the priority of each stored events and forward the events to the notifier which delivers them according to the priorities, from higher to lower priority.

Figure 3.6 shows the Activity diagram for managing message delivery, and also ensuring continuous monitoring of disconnected subscribers.

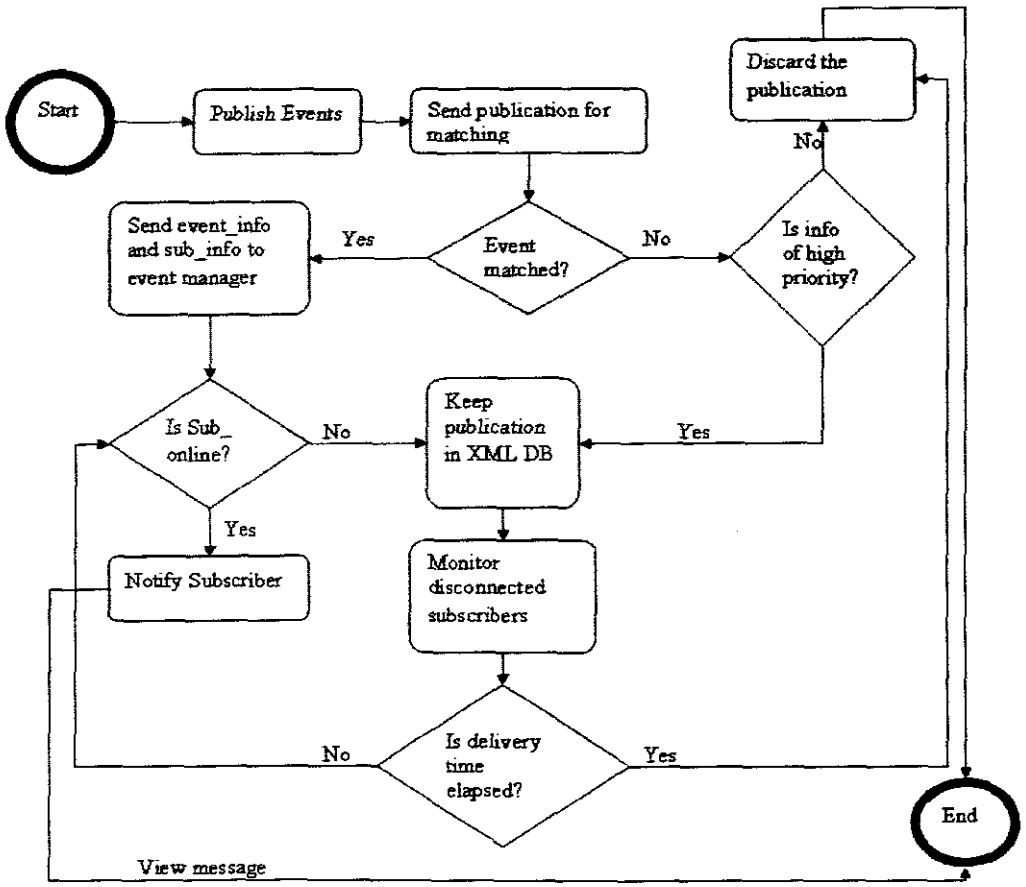



**Figure 3.6: Activity Diagram showing how a message delivery is managed.**

As subscribers always connect and disconnect from the network continuously there is a need to monitor this disruption and manage it to ensure successful delivery and also to ensure that disconnected subscribers always receive messages lost during the disconnection period. Now the activity diagram in Figure 3.6 diagrammatically simplify the flow of activities that take place during publication to notifying subscribers, and also the storage of messages not received. An event get published and matched against subscriber's preferences, if an event is matched is then received by the event manager to handle the online status of the subscriber. In this way, all subscribers find to be online receive the published event and the event is also stored in the durable database (xml database) for offline subscribers.

In case of offline subscribers, the monitoring component monitors reconnecting subscribers for delivery of stored events. Only unexpired events stored are delivered to reconnecting subscribers. Whenever there is no matching subscription to a published event, a priority variable is checked for the event, and only high priority events are stored for future subscriptions and discarded after they expire.

The Event manager keeps the event information into the XML database for later retrieval when the disconnected subscriber's status changes to active.

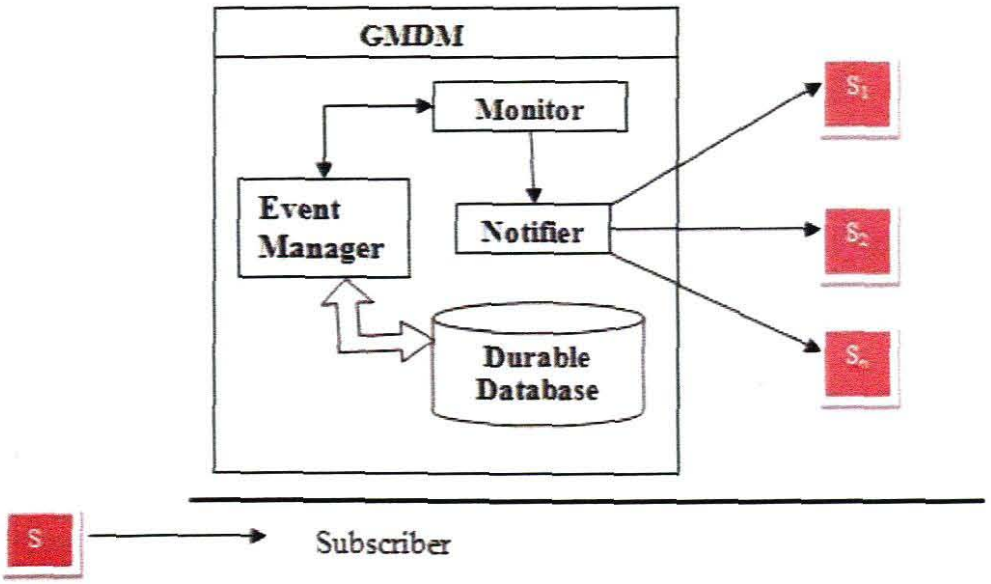



**Figure 3.7: Guaranteed Message Delivery Module (GMDM)**

### 3.5.1 The Guaranteed Message Delivery Module

The message delivery architecture crafted in this work consists of a matching manager and GMDM. The GMDM (Figure 3.7) ensures real time delivery of a message based on the priorities specified by a subscriber during subscription. The module comprises three components, the monitor, notifier and the event manager component.

The module stores data by implementing the durable subscription pattern that first saves messages for an inactive subscriber and later delivers them when the subscriber reconnects. In this way, a subscriber will not lose any messages even though it is disconnected. A durable subscription has no effect on the behavior of the subscriber or the messaging system while the subscriber is active (e.g., connected). A connected subscriber acts the same whether its subscription is durable or non-durable. The difference is in how the messaging system behaves when the subscriber is disconnected.

The guaranteed message delivery process entails the following sub-tasks:

- Managing incoming events by ensuring successful delivery
- Monitoring subscriber reconnection after unsuccessful delivery due to offline subscription.
- Sending events to interested subscribers who are online during time of notification.
- Ensuring undelivered events are kept in a persistent storage for later retrieval

#### 3.5.1.1 Monitor component

The monitor component's role is to serve as a broker-between the event manager and the notifier during message delivery. The monitor is the one service in the GMDM that guarantees that a normadic subscriber will receive its subscription when it returns.

#### 3.5.1.2 Notifier component

The Notifier component is mainly responsible for delivering messages to subscribers ready to receive a message. The Notifier service can also act as a monitor service. During notification of subscribers, it might be possible that a subscriber may disconnect or lose a signal due to mobile subscriber's movements. This makes the Notifier service to also act as a monitor by redirecting back the event to the Event Manager to keep the event in a persistent storage for later retrieval when the disconnected subscriber connect again.

#### 3.5.1.3 Event Manager Component

The Event Manager component is the sensor that keeps the monitor component informed when information is published and also checks if any subscriber has interest in the published message. This component implements the Guaranteed Delivery pattern [Enterprise Integration Patterns, 2005]. It also triggers the Notification service to notify all subscribers identified as active (online) and are capable of receiving published messages. With Guaranteed Delivery, the messaging system uses a built-in data store to persist messages. Each computer the messaging system is installed on has its own data store so that the messages can be stored locally. When the sender sends a message, the send operation does not complete successfully until the message is safely stored in the

sender's data store. Subsequently, the message is not deleted from the data store (durable database) until it is successfully forwarded to the relevant subscriber. In this way, once the sender successfully sends the message, it is always stored on persistent storage on at least one computer until it is successfully delivered to and acknowledged by the receiver.

#### 3.5.1.4 *Component interaction at the Guaranteed Message Delivery Module*

The GMDM: Guaranteed Message Delivery Module (Figure 3.7) is responsible for message delivery guarantees achieved by using the guaranteed delivery pattern [InterpriseIntegrationPatterns, 2005].

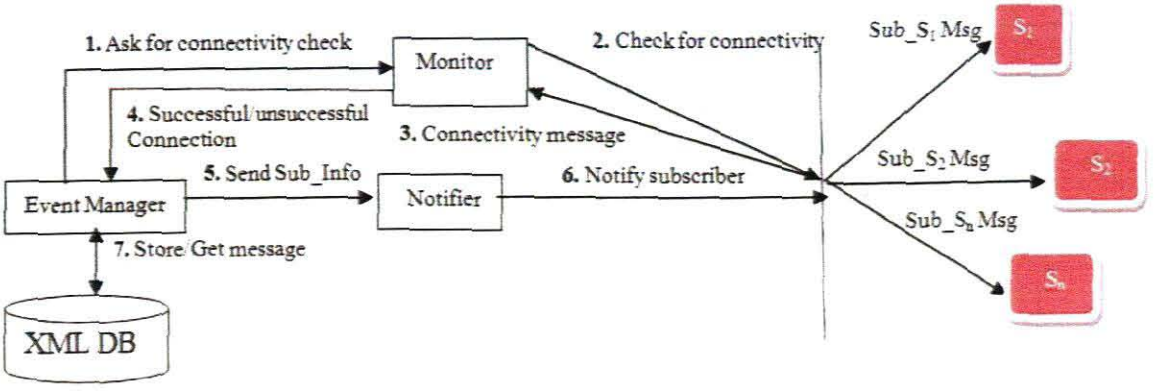



***Figure 3.8: Flow of messages at the GMDM***

Components of Figure 3.8 demonstrates how the GMDM can be used in a sports update system such as a soccer score dissemination service, where subscribers of this service are notified of score updates for soccer matches playing on the same day. Soccer games are played according to existing match fixtures. A subscriber will have to indicate which matches they are interested in at the time of registration. As the score board changes,

notifications will be issued to qualified subscribers. Each subscriber is treated as a normadic node, such that score updates are pushed to only those who are connected at any particular point in time. However, score updates are guaranteed to be delivered unless a subscription expires before the subscriber could reconnect. The system avoids delivery of stale information by deleting from durable databases all stale messages yet-to-be delivered. Duplication is also prevented in the system such that information updates are not delivered more than once. Once a new update arrives, the existing copy is replaced by the new one.

The algorithm in Table 3.1 shows how the components communicate to successfully manage and deliver an event. This is only a high-level design.

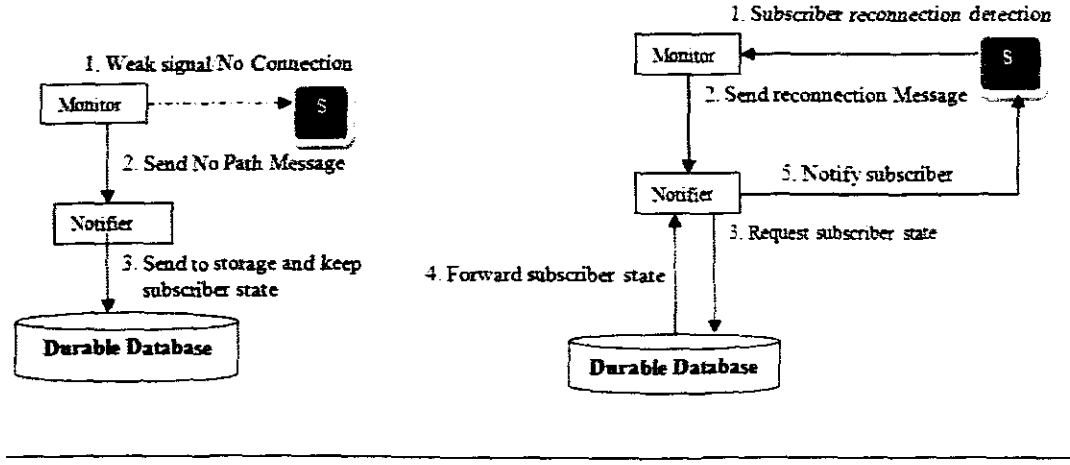
**Table 3. 1: Control Structure for events delivery and management of offline subscribers**

| **Event Delivery Management Algorithm** |
|---|
| Event Manager receives delivery destination/s for a published event<br><br>**if** (Event_Topic == X, and Destination == SubY, and Sub_Prefence == X)<br><br>Event Manager instructs the Monitor service to check SubY status<br><br>**if** (SubY_status = connected)<br><br>Monitor service notifies Event Manager<br><br>Event Manager forward info to Notifier service<br><br>Notifier service send event X to SubY<br>SubY listen for incoming events<br><br>**if** (SubY disconnect during notification)<br><br>**if** (Event not received)<br><br>Notifier service return event info to Event manager<br><br>Event Manager send info to Persistent storage<br><br>**Else**<br>Event delivered successfully<br>Wait for events from server |

```
        End If
    End If

  Else if (SubY_status = disconnected)
          Event Manager send event info to persistence storage

  End if

End if
```

Table 3.1 gives an overview of the control structure for event delivery management.

### 3.5.2 Managing undelivered messages




*a) Handling disconnected subscribers* *b) Reconnection state*

**Figure 3.9: Management of undelivered events**

Figure 3.9 depicts a diagram that shows the handling of connection and disconnection of subscribers during message dissemination. Figure 3.9a shows a subscriber not connected to the network during publication of an event which is being approved by the monitor component and also how the message get to be stored in the durable database which keeps undelivered events till they expire and be discarded from the storage. Figure 3.9b shows an event that takes place after Figure 3.9a, in this process a subscriber reconnects

to the system as it is detected by the monitor component and all stored and unexpired events are disseminated and delivered to the reconnected subscribers. When a subscriber loses its connection all messages meant for this subscriber are kept in a durable database with a time to live stamp. For every disconnected subscriber, a state is kept and retrieved later to assist the notifier to know which messages to be delivered when the subscriber reconnects at a later stage. Figure 3.9b shows how the events are delivered to a subscriber who just reconnected after loosing connection. Every event kept in the database is later retrieved and sent to a relevant subscriber who was offline at time of message delivery, and all messages or events entering the durable database are assigned a time to live stamp, this means that all messages whose time to live expires before reconnection are discarded from the database.

### 3.5.3 The Matching Manager Module

The matching manager module functions like a mediator, it receives messages from a publisher (Figure 3.10), and matches the information with the subscriber preferences. This module (Figure 3.10) comprises of a Matching component, and a subscription aggregator. The matching component adopts a pattern called a message filter pattern which filters unwanted data and only allows relevant data to be sent and is responsible for matching all subscriptions against subscriber's preferences (topic). A publisher publishes messages into the Matching manager which also comprises of a database which keeps the published information which is used by the matching module to match all published information with subscriber's interest. The subscription aggregator manages subscriber's interests and interacts with the Matching manager by supplying subscriber interests to be

matched against published information. The subscription aggregator interacts with the database, which keeps the subscriptions with subscriber interests (preferences). The matching manager interacts with the Guaranteed Message Delivery Module (Figure 3.7) which places or notifies the subscribers through notifications. After the message is matched and assigned a priority value, the message is then sent or forwarded to the Guaranteed Message Delivery Module. The Matching manager module uses two databases, the first database which stores all published information, and the second database which keeps subscriber profiles, including subscriber's interests.

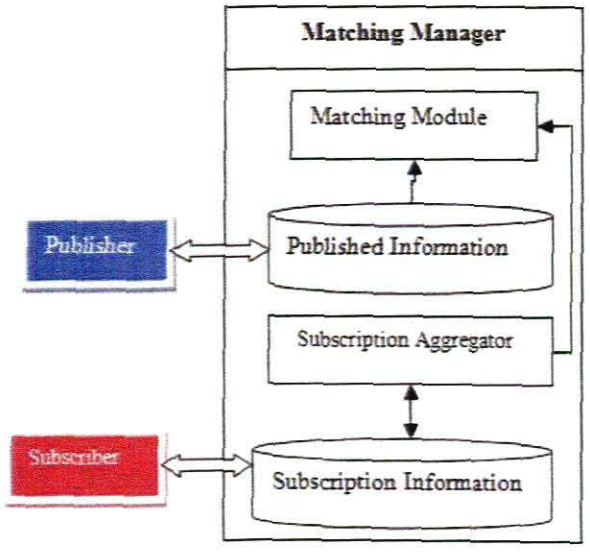



**Figure 3.10: The Matching Manager Module Architecture**

The matching manager (figure 3.10) is the engine of the whole system. This manager comprises of four communicating parts: the matching module, published information, subscription aggregator, and the subscription information. The publisher which is an external entity from the system, publishes events which are detected and stored in the published information database for use during matching process. The published

information is used by the matching module to match the event against subscription information. This matching is carried away by the matching module and the subscription aggregator. Whenever a subscriber subscribes to the system, the subscriber's profile is stored in the subscription database which communicates with the subscription aggregator to match the subscriptions against the published information. Whenever matching is successful, the matched information is communicated with the GMDM (Figure 3.7). The matching manager allows published events to be correctly matched with stored subscriber's subscriptions. If incorrect matching is done, the incorrect messages will be delivered to subscribers, some subscribers might receive messages of events they never registered for. In this way the matching manager ensures that publications match the correct subscriptions.

Figure 3.11 shows the flow of messages in the Matching Manager Architecture (Figure 3.10). After matching (figure 3.11), the matched data is sent to GMDM (Figure 3.7) and messages in GMDM flows as shown in figure 3.8.

### 3.5.4 Matching Manager Module Components

The matching manager of figure 3.10 matches all incoming events against subscriptions as shown in figure 3.11. Subscriptions are subscriber information containing subscriber preferences, such as a topic or category to which a subscriber has shown interest on.

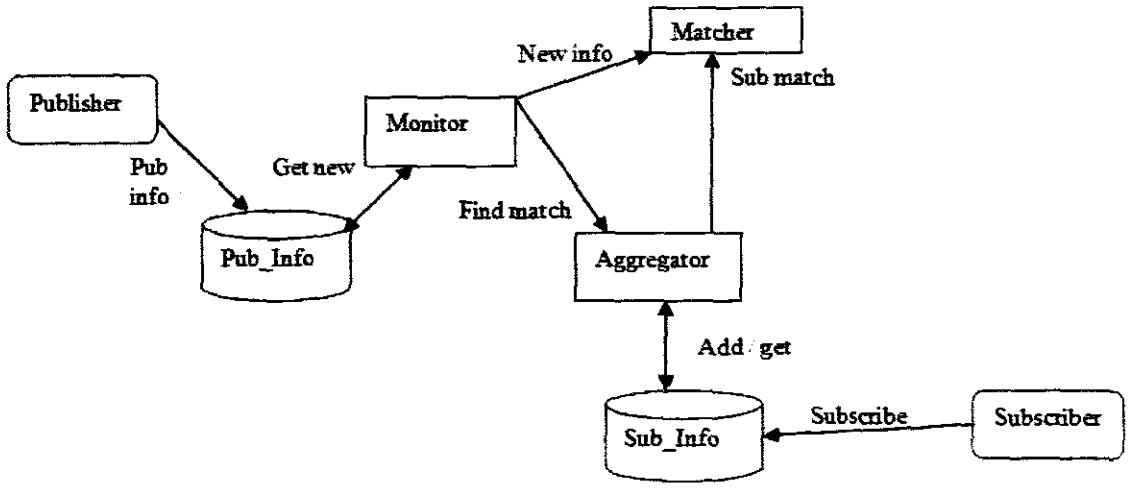



**Figure 3.11: Data Flow in the Matching Management Process**

The Matching process entails the following sub-tasks (This process is based on Figure 3.11):

- Subscriber registration and selection of preferences (Subscriptions)
- Publication of messages by a publisher to registered subscribers
- Monitoring events entering the Published information database (Pub_Info)
- Aggregator retrieves and forwards all new events in the Pub_Info database to the Matcher component as shown in figure 3.11
- Matching events against subscriptions in the Subscription database (Sub_Info)

Information published is matched against subscriber's preferences (Figure 3.11). Figure 3.11 shows how the matching will take place. The Matching process consists of four services namely, the Subscription, Monitoring, Matching and the Aggregation. Below we briefly summarize each service according to its function in the Matching of information.

Subscriber registration or subscription occurs the same way as discussed in section 3.5 figure 3.1

### 3.5.4.1 Monitoring Service

The Monitor service observes new events entering the Published information database, if new events are found, the Monitor service interact with the aggregator to get the new event, and forward it to the Matcher service. The Matcher service matches the information against subscriptions stored in the Sub Info database (a database that keeps subscription information).

### 3.5.4.2 Matching Service

The Matcher service performs matching of subscriptions against publications, for every event entering the Pub Info database (a database that keeps published information) is retrieved, and matched against subscriptions from the Sub Info database.

These services of the Matching manager could also be used in our sports update system for the soccer dissemination service. Whenever a score update for a playing match is published, the message is stored into the published information database and the monitoring service triggers the publication and makes use of the published event to send it to the matcher for matching with all subscribers shown interest for the soccer category. If matching is successful the matched information is forwarded to the GMDM (Figure 3.7) for delivery to matched subscribers for the soccer score update category. The published soccer score update flows according to Figure 3.8.

**Table 3. 2: Control Structure for matching events against subscriptions**

| **Matching Algorithm** | **Priority Handling Algorithm** |
|---|---|
| **if** (New_Event found)<br>Monitor service get and forward event<br>**if** (New_Event_Topic == x)<br>Matcher service checks Sub_Info<br>**if** (Sub_Preference == x)<br>Matching successful<br>**Else**<br>Matching unsuccessful<br>Event kept in the Pub_Info DB<br>**End if**<br>**End if**<br>**End if** | **if** (New_Event = Higher Priority_Value)<br>**if** (Notifier Service not busy)<br>Block incoming Events<br>Process Event with Higher Priority<br>**Else**<br>Permit Pre-emption<br>Process Event with Higher Priority<br>**End if**<br>**Else**<br>Use FIFO Processing<br>**End if** |
| *New_Event_Topic* → Publication category<br>*Sub_Preference* → Subscriber's subscription interests.<br>*Pub_Info DB* → Database that keeps published information | *FIFO* → First In First Out<br>*Higher Priority_Value* → A value signalling pre-emption of running processes |

Table 3.2 gives an overview of the control structure for identifying a new event that has just entered the system, and needs to be matched against subscriptions, and also an algorithm to identify an event with a higher priority to give it first preference during processing, and preempt the running process. The event with a higher priority might be an event with a time stamp, an event with a time stamp is an event that needs to be delivered to its destination for a specified time. For Example if event X on the queue has a time stamp of 7s, and there is a new event Y with a time stamp of less than 7s, then Y

will be delivered, since it has a shorter time to live. Events are not determined by their life period on the queue for delivering, but by the time they have left for delivery. In a sports update dissemination service where a subscriber preferred that rugby updates should precede soccer updates, this means that when a soccer and rugby update messages are published while the subscriber is disconnected and the soccer update is published before the rugby update. When the subscriber reconnects the rugby update will be delivered first irrespective of which message was published before the other. In the case of messages with the same priority the first come first served (FCFS) is used to process the messages.

#### 3.5.4.3 Aggregator Service

The Aggregator service acts as an agent, and an advantage of using it comes with its ability to integrate communication of two or more services for quick and efficient management of message retrieval and processing and also act as a communication link between the Matcher service and the Sub_Info database. This service is initiated by the Matcher service, which alerts it to get subscription information from the Sub_Info database. The Aggregator service forwards the information to the Matcher service to perform matching against publications.

### 3.5.5 Class Diagram

Figure 3.12 is the class diagram that represents the classes of the system implementation together with their attributes and operations (methods). The implementation of the system is covered by nine classes as follows:

**Publisher:** This class represents a bean that defines properties and behaviors for handling information of the publisher entity.

**Subscriber:** This class represents a bean that defines properties and behaviors for handling information of the subscriber entity.

**Message**: This class is responsible for storing messages in an XML storage format which acts as a durable database for storing undelivered messages. The XML storage is only for subscribers who are not connected or offline at time of delivery. The Message class defines methods for saving, deleting and retrieving messages.

**Preferences:** This class represents a bean that defines properties and behaviors for handling topic publications. This class is a convergence of four properties the title, category, body and a keyword

**SubscriberProfile:** This class is a derivation from the super class Profile. Added to this, it is a specialized bean that defines properties and behaviors for handling information pertinent subscriber profile.

**SubNotifier:** This class is responsible for notifying subscribers of published messages. It's only activated when there is a message to deliver and it is instructed by the event manager to do so.

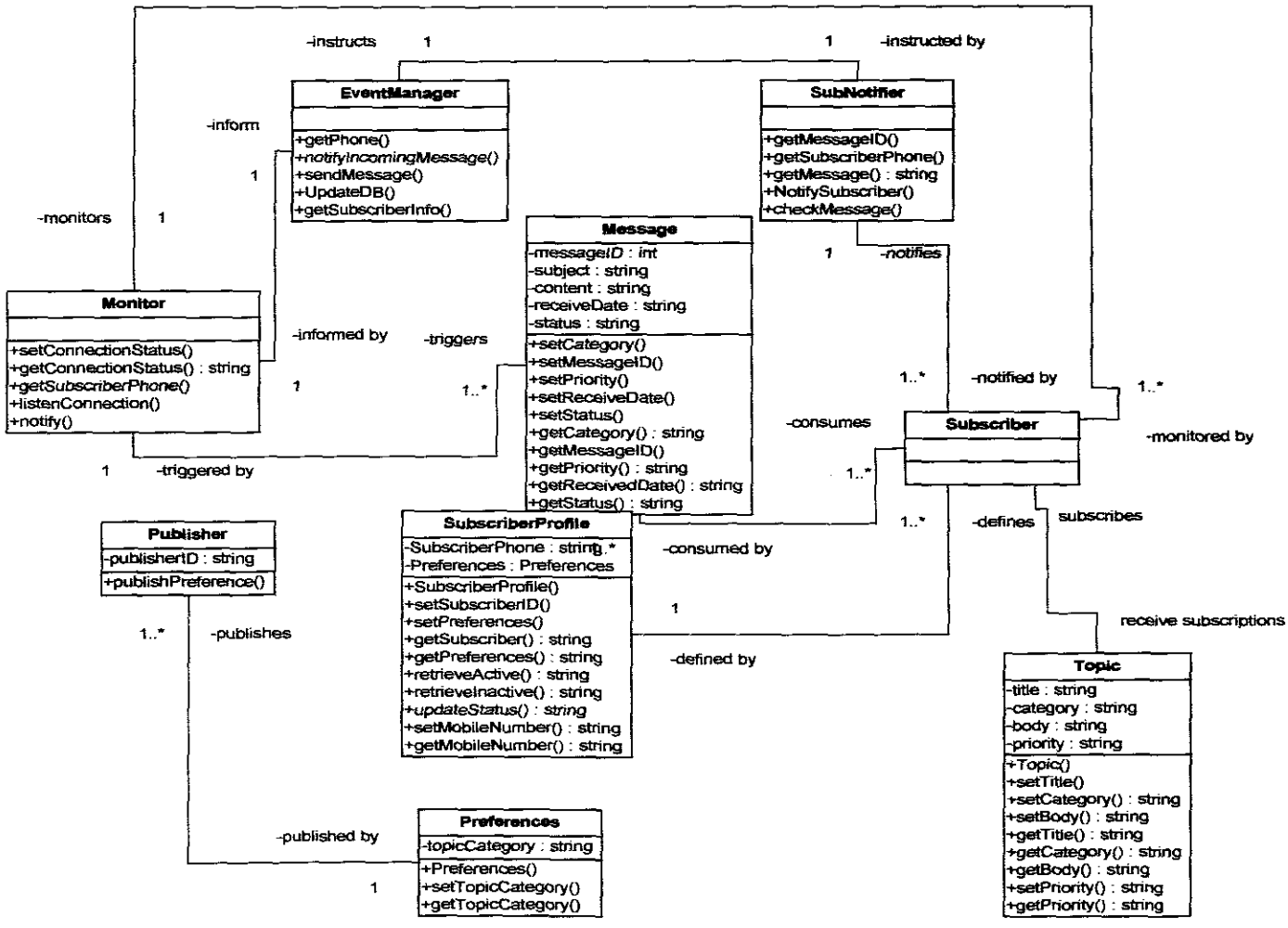
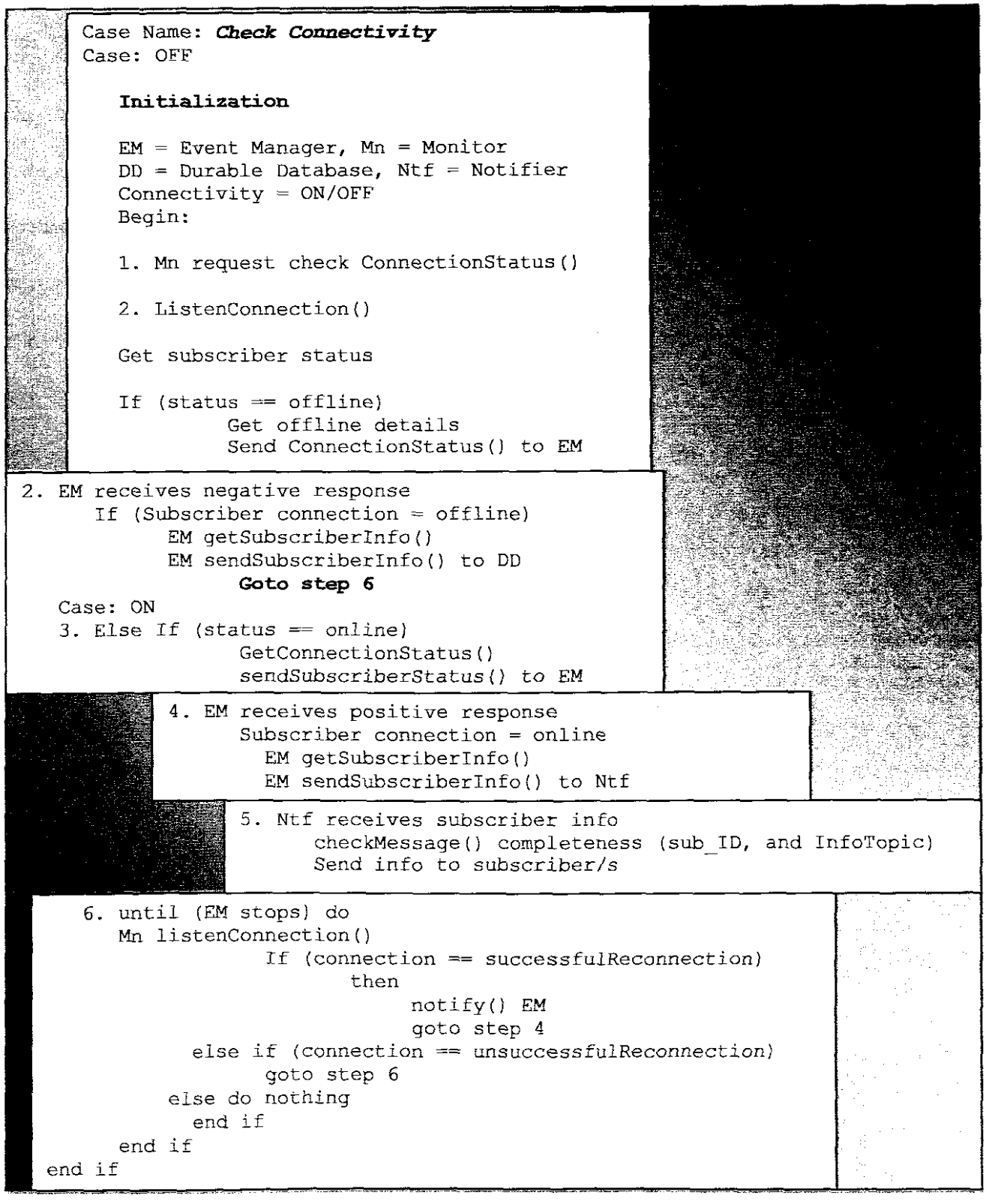



**Figure 3.12: The Publish/Subscribe Message Delivery Class Diagram**

**Event Manger:** This class is responsible for initiating the communication between itself and the monitor class, and also instructs the notifier class to notify subscribers, and in case of undelivered events it direct them to the XML database for future retrieval.

**Monitor:** This class is responsible for monitoring the migration of subscribers from broker to broker and also ensures that disconnected subscribers do not lose events published to them during their disconnection period.

**Figure 3.13: Component Interaction algorithm for Guaranteed Delivery**



```
Case Name: Check Connectivity
Case: OFF

   Initialization

   EM = Event Manager, Mn = Monitor
   DD = Durable Database, Ntf = Notifier
   Connectivity = ON/OFF
   Begin:

   1. Mn request check ConnectionStatus()

   2. ListenConnection()

   Get subscriber status

   If (status == offline)
            Get offline details
            Send ConnectionStatus() to EM

2. EM receives negative response
     If (Subscriber connection = offline)
          EM getSubscriberInfo()
          EM sendSubscriberInfo() to DD
               Goto step 6
  Case: ON
  3. Else If (status == online)
               GetConnectionStatus()
               sendSubscriberStatus() to EM

   4. EM receives positive response
        Subscriber connection = online
          EM getSubscriberInfo()
          EM sendSubscriberInfo() to Ntf

      5. Ntf receives subscriber info
           checkMessage() completeness (sub_ID, and InfoTopic)
           Send info to subscriber/s

  6. until (EM stops) do
     Mn listenConnection()
               If (connection == successfulReconnection)
                     then
                          notify() EM
                          goto step 4
          else if (connection == unsuccessfulReconnection)
               goto step 6
        else do nothing
          end if
     end if
end if
```

This chapter proposed an architecture for message delivery, and also discussed modules of the architecture and mechanisms to clarify on how each module contribute to the real-time and reliable delivery of messages to registered subscribers. We also proposed and constructed a matching manager component to ensure reliable matching of published messages with subscriber preferences, and we proposed and also constructed a guaranteed message delivery module/mechanism to handle the frequent disconnection of subscribers and to ensure guaranteed delivery to registered subscribers. The next chapter is a proof of concept in which we implement the solutions we discussed in chapter three and we also run a simulation to guarantee reliable message delivery.

# CHAPTER FOUR

# MODEL IMPLEMENTATION

## 4.1 Introduction

The previous chapter presented the overall development of a proposed architectural model for guaranteeing message delivery for mobile subscribers who continuously move from broker to broker, and also presented some delivery algorithms that consider priority for messages that need to be delivered urgently. The previous chapter also presented different components and services that are involved during message delivery, and also the algorithm that shows the interaction of these components to ensure timely delivery. The focus of this work is on developing mechanisms to provide guaranteed real-time delivery of published events to mobile subscribers who changes location or move from broker to broker, which need to be considered too during dealing with issues of ensuring reliable delivery. This chapter presents the design, implementation and evaluation of the proposed message delivery model discussed in Chapter 3, and also the message delivery reliability is also evaluated in this chapter.

## 4.2 Description of the Implementation

The Guaranteed message delivery model is implemented as a web application that supports publication of messages through simple web forms processed by simple servlets running in a web container. The published messages are disseminated to potential subscribers through SMS. SMSs are sent to potential subscribers using the WMABridge API. The WMA Bridge API enables J2SE (Java 2 Standard Edition) applications to easily interface with MIDlets defined by the J2ME specification through messaging. Undelivered SMS messages are temporally stored in an XML database, when the subscriber connect all SMS messages belongs to that particular subscriber are then delivered to the subscriber.

The proposed model is crafted as a three tier implementation architecture which is comprised of the client, logic and the information tier. The client tier is a convergence of two MIDlets that are developed under J2ME specification and tested using the Sun Wireless Toolkit 2.3 version. The first MIDlet defines an unconnected subscriber when the SMS message gets published. The second MIDlet defines an interface for receiving incoming messages published by legitimate publishers. The logic tier is a convergence of various packages structured into three standard components of business logic, access logic and presentation logic. The logic tier runs under a web container basically the servlet container which is Apache Tomcat Application Server 5.0.The web container is a module that handles processing of servlet components, managing various aspects such as state management, concurrency control, thereby giving the developers the freedom of developing an application without worries of system dependent mapping and calls. The information tier is a collection of XML documents which is referred to as an XML

database. The collection is accessed through JAXP for reading and writing to XML documents. The implementation adopted the use of DOM Parser which is capable of structuring information into a hierarchical tree.

## 4.3 Implementation Model

Figure 4.1 illustrates a Publish/Subscribe implementation model which comprises three tiers, namely the client tier, middleware tier and Enterprise information tier. The client tier can either

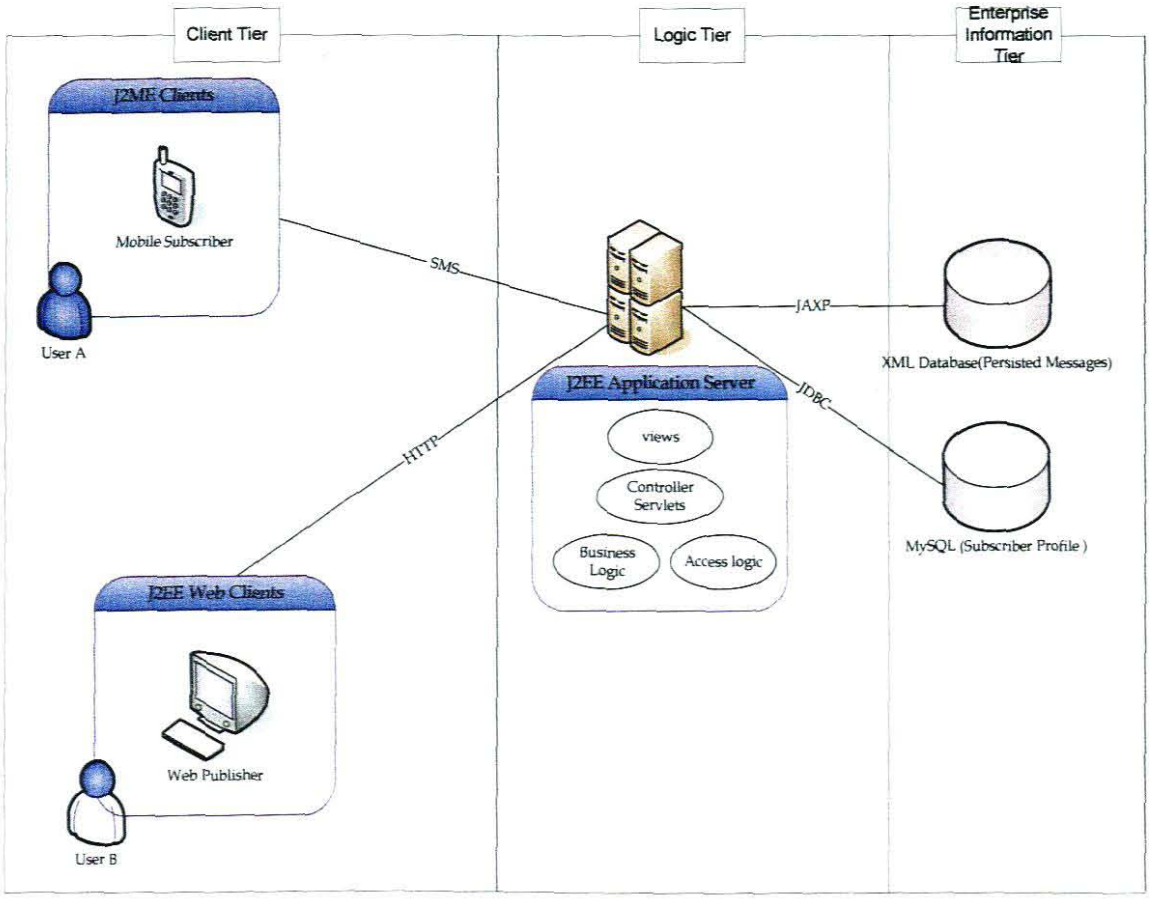



**Figure 4.1: The System Implementation Model**

be J2ME or web clients communicating with web components in the J2EE Application Server

within the logic tier. The logic tier is a convergence of web components running in the web container (Servlet engine) and business components forming the application logic. Lastly the information tier is implemented in terms of an XML database which an XML file persisting published messages, and the MySQL database which keeps subscriber profiles. The application interfaces with the XML database using JAXP (Java API for XML Processing).

## 4.4 Runtime Interaction Components during Message Delivery

Figure 4.2 depict the runtime interaction components which guarantee message delivery, the figure shows five communication sites, the Publishing, middleware, notification, monitoring, and mobile subscriber site. Below we clearly describe communication in each site stated above:

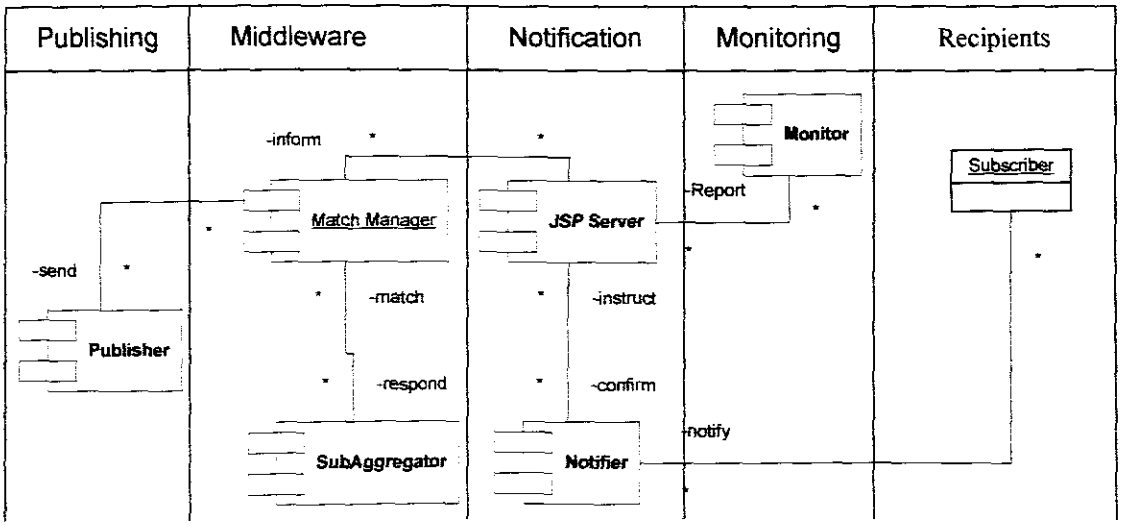



**Figure 4.2: Runtime Interaction of Message delivery components**

### 4.4.1 Publishing

The publishing site initiates the whole process by publishing events to the Middleware component. The publishing can be done manually or automatically by the system when a subscriber has disconnected and reconnect later. Subscription does not take any effect during runtime, but it is run separately by a subscriber. Subscription is discussed in details in chapter three.

### 4.4.2 Middleware

The middleware uses the published information or events to match them against subscription preferences, for every event published, the match manager communicate with the subscription aggregator (SubAggregator) which communicates back by accessing the subscriber database to check for subscription preferences. The match manager informs the JSP Server of all subscribers matching the publication.

### 4.4.3 Notification

Notification of published event/s can be sent to more than one subscriber depending on the number subscribers registered for the published preference. After matching has been done successfully, the JSP Server takes over, and makes use of the matched information, and use the subscriber id, and the published information to instruct the notifier component to deliver the publication to the registered subscriber. The Notifier component uses the subscriber id to identify the relevant subscriber to receive the notification. The

notifier confirms the notification to the JSP server as successful or unsuccessful delivery. Delivery might be unsuccessful at runtime due to a number of factors ranging from mobile device power failure, unexpected disconnections due to weak network signals, or poor connections, and this usually happens in rural areas where poor connection is experience by subscribers. We adopted the memento pattern in solving the issue of subscribers disconnecting from the system during message dissemination. By employing the memento pattern we are able to notify the subscribers of events published during their disconnection period.

When a published message is of higher priority, the JSP sever reacts to the priority factor by instructing the notifier to deliver the message first, irrespective of all messaging waiting to be delivered. All messages with a higher priority get delivered first, jumping all those in queue. Messages with a specified delivery time are also considered of high priority, depending on the delivery time specified. The message must have a shorter time to live in the queue to be considered of higher priority. For Example if a message `m1` = 5s, and `m2` = 5min, then message `m1` has a high priority than message `m2`, since `m1` has to be delivered within 5seconds, and `m2` within 5minutes. Therefore time can also be considered as the factor determining message delivery priority.

In a case whereby a subscriber has disconnected from the network due to the above factors, the JSP server communicates with the monitor component to react to reconnecting subscriptions.

### 4.4.4 Monitoring

The monitor component is only active when there are subscribers disconnected from the network, and communicates with the JSP Server which instructs the monitor to react to disconnected subscriptions. The monitor component monitors and detects reconnecting subscriptions which disconnected due to factors mentioned above. When a subscriber reconnects, the monitor component detects the reconnection and informs the JSP server which reacts by requesting all publication stored in an XML database for the reconnected subscriber by using the subscriber's id, and instruct the notifier component to deliver the event or message.

### 4.4.5 Recipients

The recipients can be subscribers using any mobile device, i.e. mobile phone, PDA, etc. These recipients who we refer to them as subscribers in our work register their preferences with the system to receive published events. From our example of sports update dissemination service a subscriber might prefer to receive soccer and rugby score notifications. A recipient changes location due to its movements. Therefore the proposed delivery model takes care of mobility of the subscribers by using the monitor service to monitor subscriber movements and notify the Event manager which will then ensure the whereabouts of a mobile subscriber. A message published at a particular time is received by the subscriber with few milliseconds as demonstrated in our reliability evaluation in section 4.5. A subscriber might lose connection from the network due to poor network signal, or intentionally by switching off his/her mobile phone or PDA and Sometimes unintentionally due to battery power.

## 4.5 m-InstantSportsUpdate (ISU) Scenario

This section presents m-SportsInstantUpdate, a sports update dissemination service for mobile users based on the publish/subscribe interaction model. This service enables users to publish and receive instant scores updates of sports of their interests, and yet stay mobile. Users receives sports news or updates of matches playing at that time, for the current scores update for every sports that the user (subscriber) showed interests on, and this will be possible on a WAP-enabled mobile phone, or in a PDA (Personal Digital Assistant), and publish the current sports update using the m-InstantSportsUpdate's Web interface.

In addition, they may subscribe to particular sports categories (e.g. soccer, rugby, etc), and supply keywords to refine their subscriptions. Subscribers will receive notifications when score updates of their sports interests matching their subscriptions are published. m-InstantSportsUpdate is a prototype system simulated using the loosely-coupled remotely accessible services. The implemented components, the Web-based publish/subscribe service, and the personal mobility component in particular, offer generic functionality and are applicable for integration into various content dissemination services.

### 4.5.1 m-InstantSportsUpdate - a Score Dissemination Service

m-InstantSportsUpdate is a content dissemination service for publishing and delivering score updates in the form of multimedia messages to mobile users. It offers flexible usage scenarios enabling personal mobility: Users can apply various devices for browsing, and

receiving the sports scores instantly. For example, users may receive a update service about a game playing on the same day, and every time the score changes, or one team scores, the subscriber is alerted of the change, and also notified of the half-time, and full time scores. The users may define and publish the different sports score updates (e.g. cricket, soccer, rugby, etc.), and define subscriptions using a WAP-enabled mobile phone, or a PDA.

## 4.6 Implementation Screenshots of the m- InstantSportsUpdate

This section presents some of the interfaces used to fulfill the goal of this research to develop a mechanism to guarantee message delivery in a publish/subscribe environment.

### 4.6.1 Subscribers registering their preferences

Figure 4.4.1 shows an interface used by subscribers to register for their preferences in order to receive published events based on their preferences. The subscriber provides his/her details such as name, surname, and cell number where the published event will be sent, and also registers for his/her preferences under topic by selecting one or more preferences and also specifies a priority for each preference. The priority factor is used to send stored messages according to their priorities during subscriber reconnection. Messages with the same priority are delivered on a first come first served basis (FCFS).

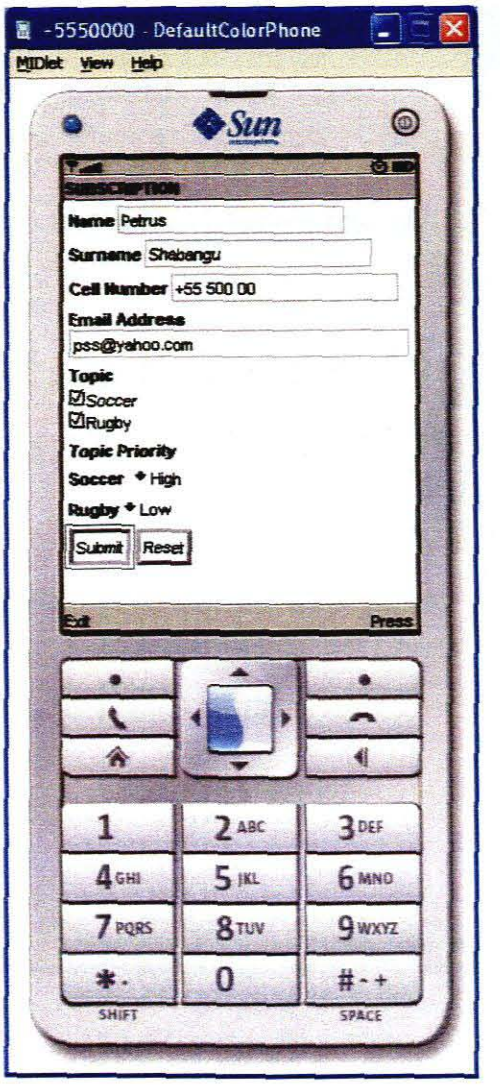



**Figure 4.3: A subscriber interface for registering preferences**

### 4.6.2 Publishing information to subscribers

Figure 4.4 is the publisher side of the implementation where a publisher specifies all information to be published to all subscribers who shown interest to a specific topic (category) during subscription. In this case soccer is being specified as the topic, and all subscribers under this category will receive the message.

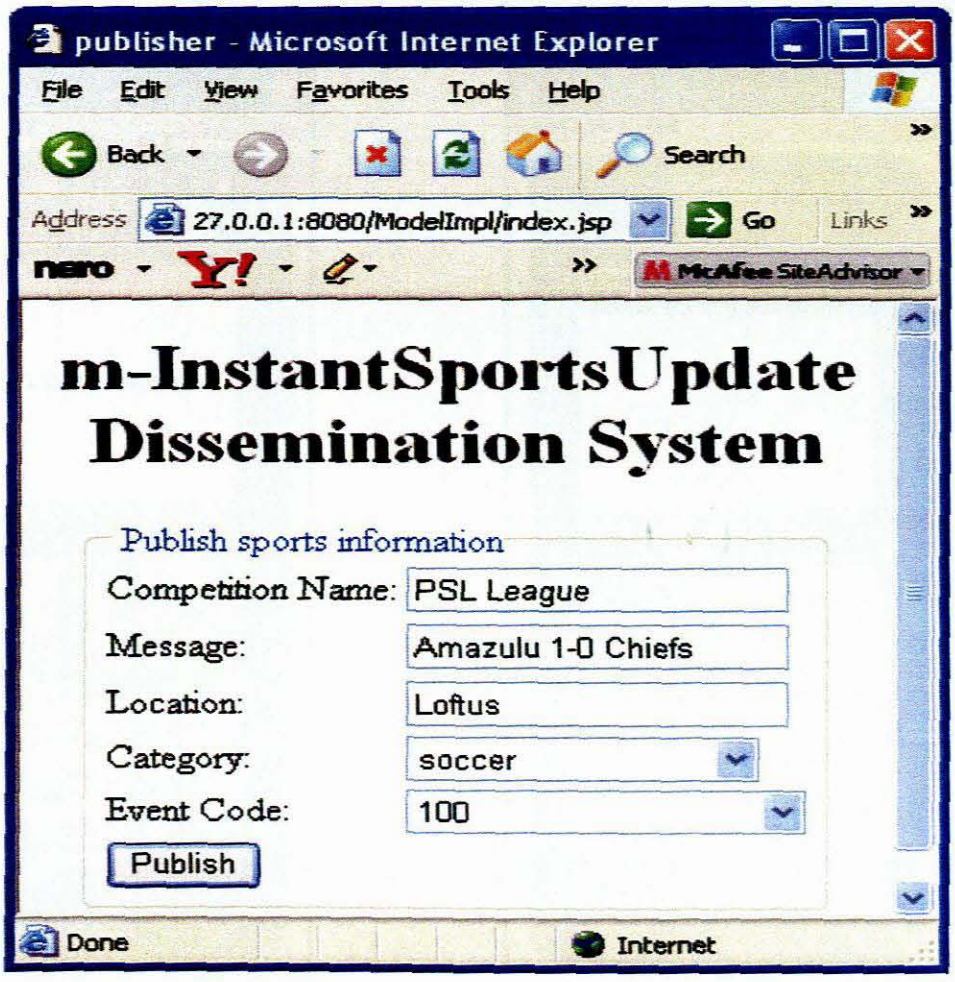



**Figure 4.4: A publisher interface for publishing score updates**

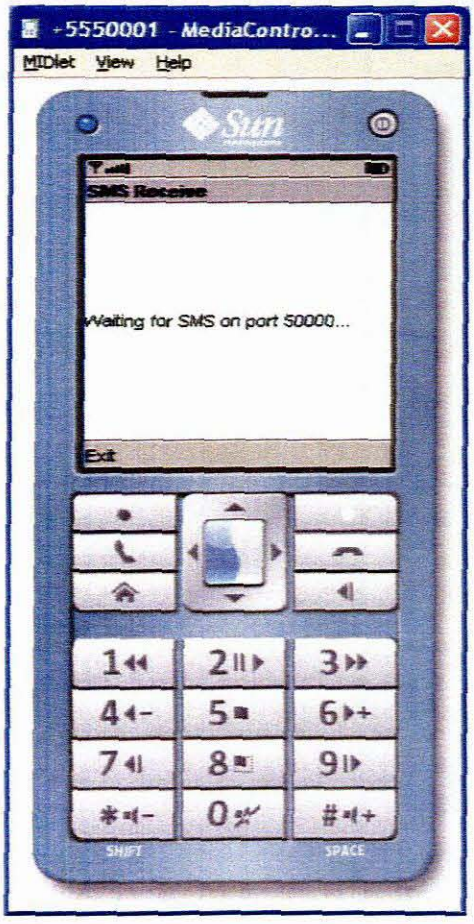



**Figure 4.5: Active Subscriber**


**Figure 4.6: Inactive Subscriber**

Figure 4.5 and figure 4.6 show an active and inactive mobile devices respectively, the active mobile device with an id "5550001" is ready to receive any published information, and the inactive mobile device with an id "5550000" cannot receive any instant published messages due to its offline state but all messages for this subscriber re stored in persistence storage and retrieved when the subscriber reconnects. The disconnection of a subscriber can be as a result of many factors such as battery failure, network connectivity or mobile device switched off. All published messages to the inactive subscriber will be stored in an XML database to be retrieved and forwarded to the subscriber after successful reconnection.

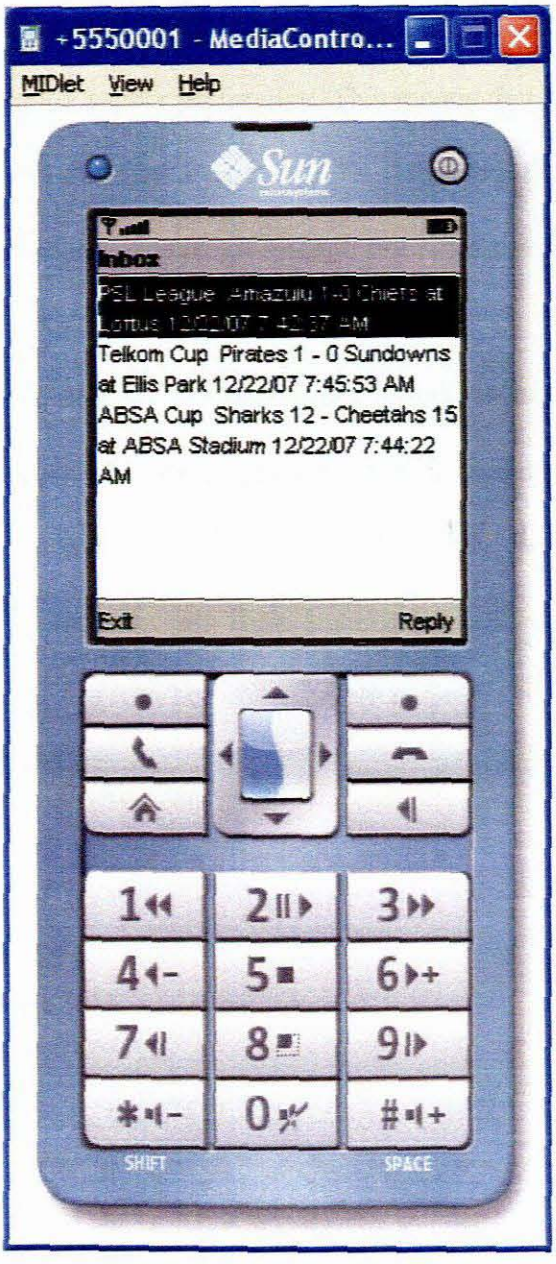



**Figure 4.7: Three Sent Notifications received by an Active Subscriber**

Figure 4.7 shows an active mobile device receiving three events being published by a soccer match publisher of figure 4.4. The two subscribers (Figure 4.5 and Figure 4.6) have both registered for soccer and rugby as their preferences. But the inactive subscriber (figure 4.6) will not receive the messages due to its inactive status. But all the three published events will be stored in an xml database (figure 4.8) to be retrieved when

subscriber "5550000" reconnects. Subscriber "5550000" preferred the rugby preference to have high priority than the soccer preference. This means that from the three published events the message for the rugby event will be delivered first even though it was published after the soccer event because of its high priority from the soccer preference.

Figure 4.8 shows the XML database for persistence storage of undelivered messages. The inactive Subscriber of figure 4.6's messages are stored in this database for later retrieval when the mobile device shows an active state. When the subscriber "5550000" reconnects or when its status changes from inactive to active the stored messages in Figure 4.8 will be forwarded to him/her immediately as he/she reconnects.

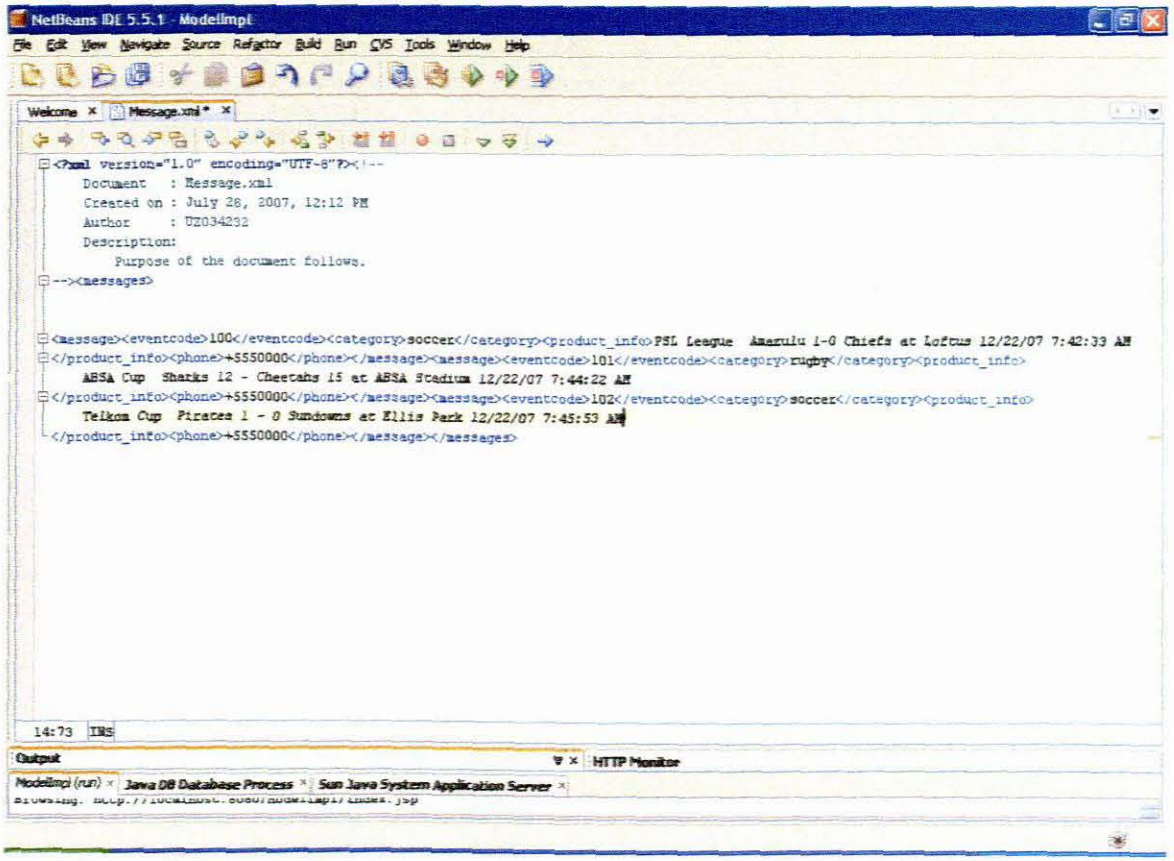



**Figure 4.8: XML database for persistence storage of undelivered events: Event stored**

Messages are not duplicated since our implementation ensures only once delivery to a mobile subscriber.

Figure 4.9 shows the XML database empty with no stored events. Figure 4.8 has three events waiting to be delivered to a subscriber who is currently disconnected from the system. After the subscriber has successfully reconnected to the system, the stored messages are retrieved and forwarded to the reconnected subscriber (subscriber "5550000"). After the messages have been successfully delivered, figure 4.8 changes to figure 4.9 with no events stored on the XML database.

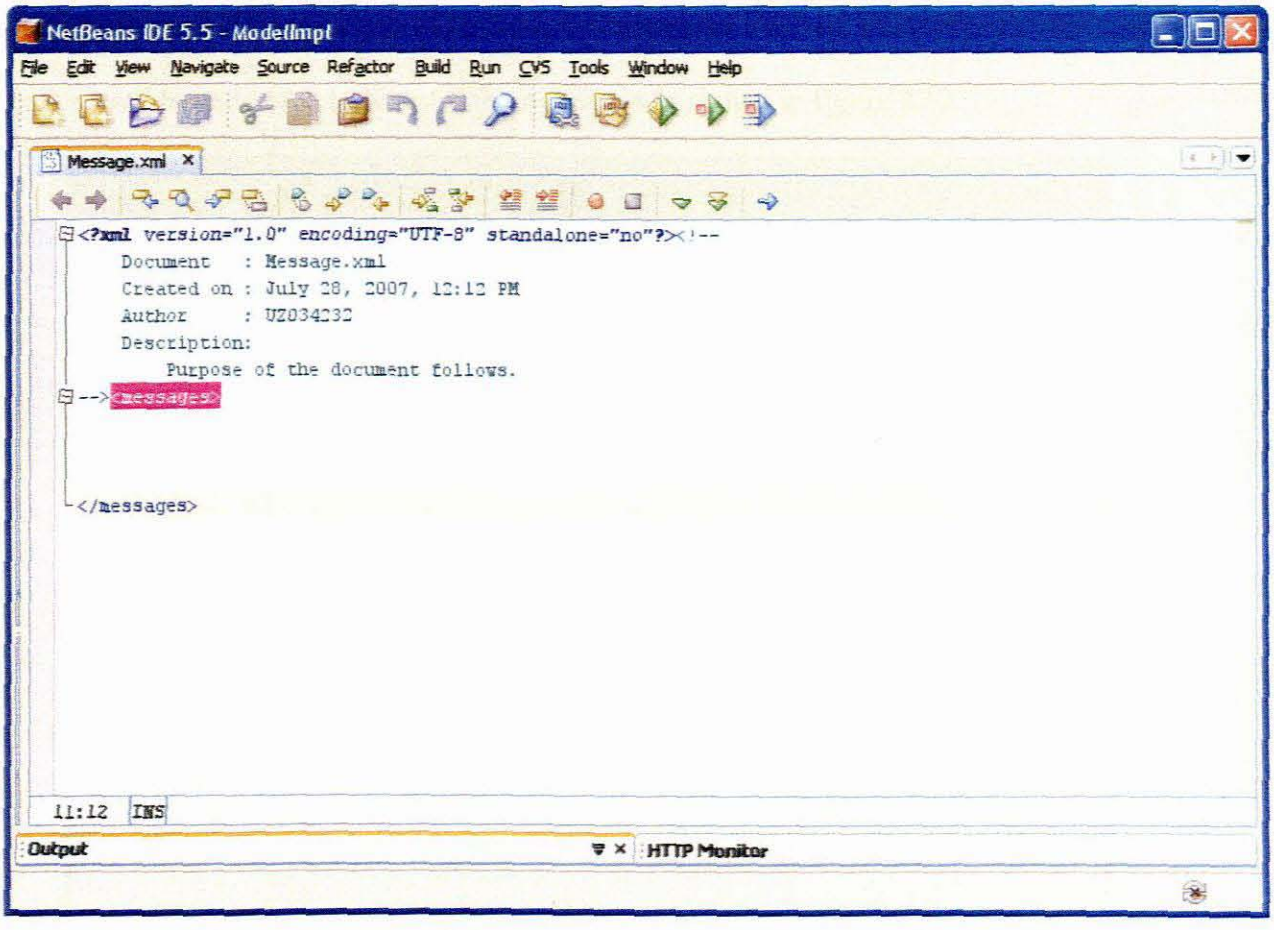



**Figure 4.9: XML database for persistence storage of undelivered events: After event retrieved**

Figure 4.10 shows the subscriber database for all registered subscribers containing a subscriber's details and a subscriber's connection status (sub status) to determine whether

a subscriber is online or offline (online = active, and offline = inactive). In figure 4.10 we observe that the subscriber with an id "5550002" is in an inactive state, and all events directed to him/her will be kept in figure 4.8. When the status of this subscriber changes from inactive to active, or when the subscriber regain network connectivity, all published events for this subscriber will be delivered. This means that a subscriber's status determines if a message or event should be delivered or not. Figure 10.11 shows the database for priority preferences used to determine which message is delivered first after a subscriber has reconnected. This priority factor is used only for disconnected subscribers and for stored messages and for connected (active) subscribers a first come first served basis is used to forward the messages as shown in figure 4.12.

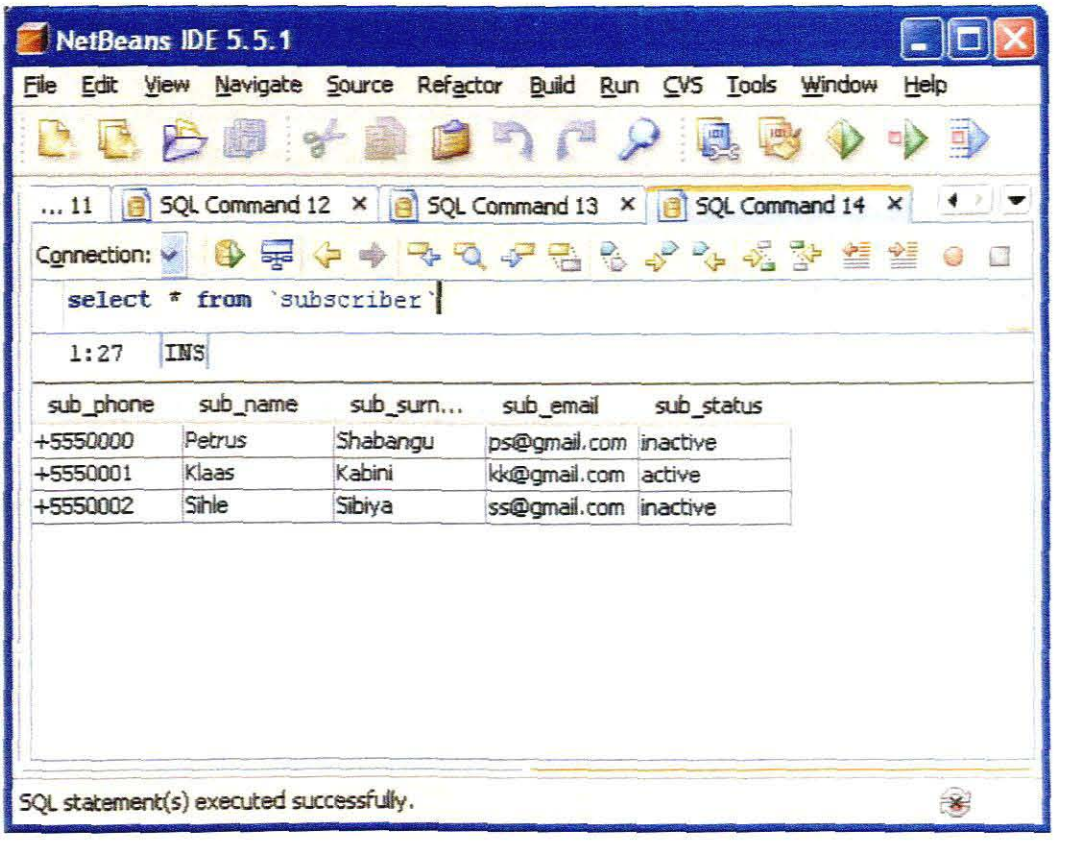



**Figure 4.10: Subscriber's database showing subscriber's status connection**

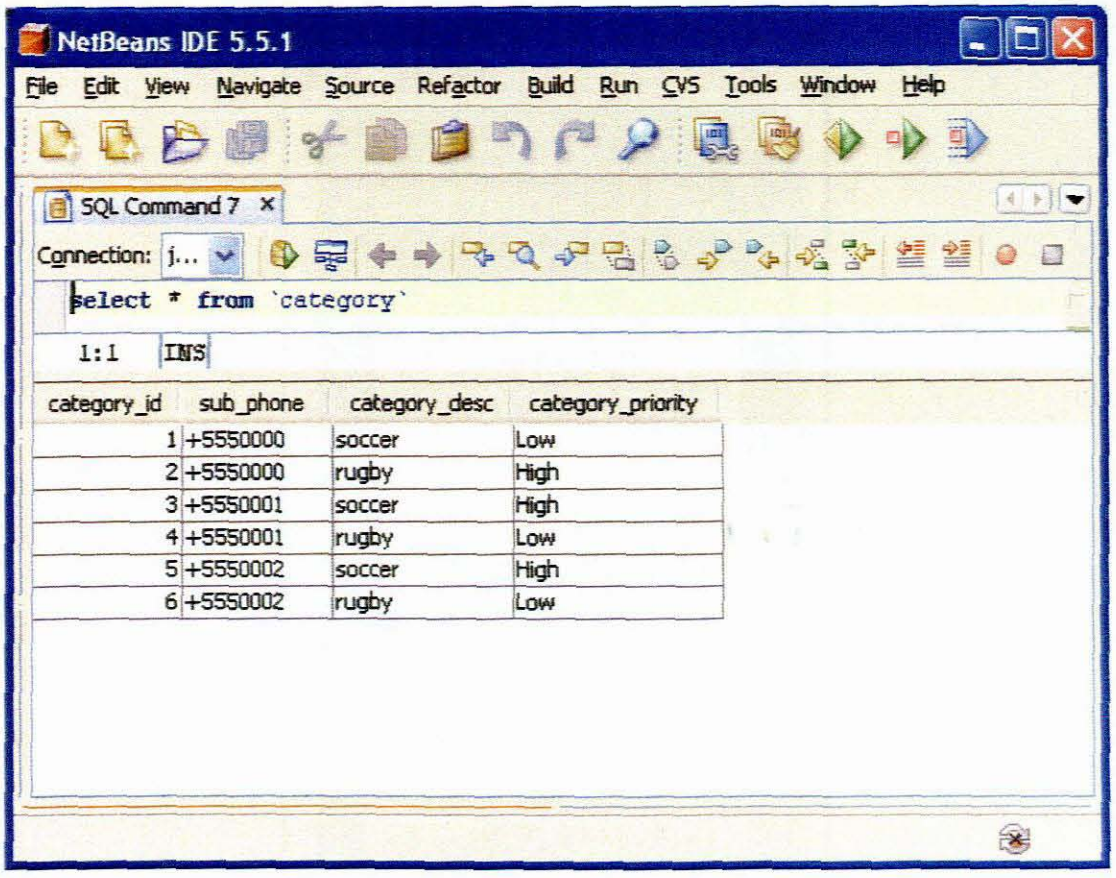



**Figure 4.11: Subscriber's database for priority preferences**

Figure 4.12 shows event notification to subscriber "5550000" who was previously disconnected when the messages were published. The messages were retrieved from the XMLdatabase (Figure 4.8) after the Monitoring component (Chapter 3) noticed that the subscriber has reconnected. The messages are being delivered according to their priority, from the high to a low priority. The rugby event was published second, but is being received by subscriber "5550000" first since it has a high priority than the others.

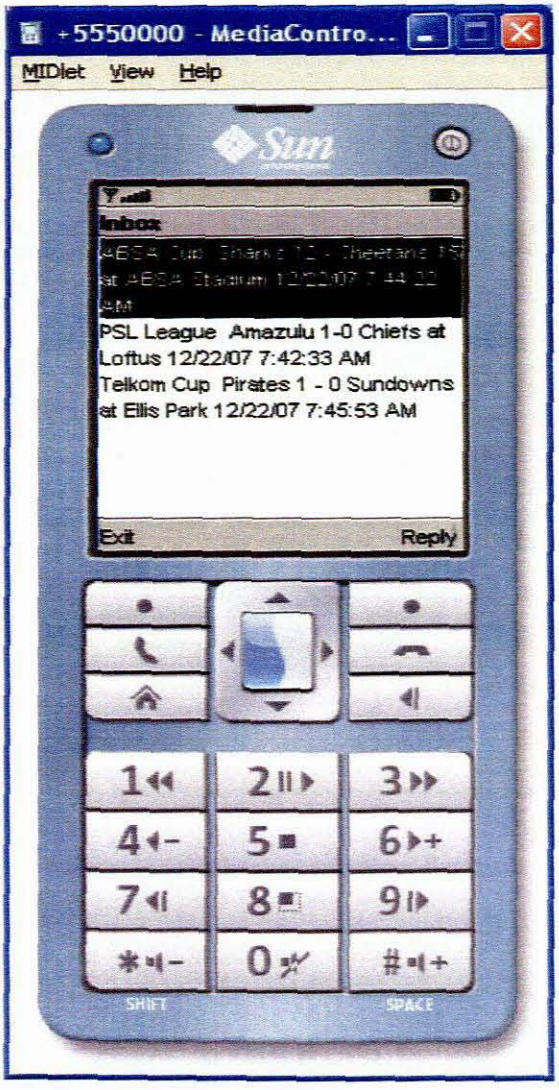



**Figure 4.12: Subscriber receives stored messages after reconnection based on priority**

Figure 4.12 allows subscribers to disconnect at any time and also connect when they need to without asking for messages lost during disconnection period, but rather receives all dated messages published to him/her after reconnection.

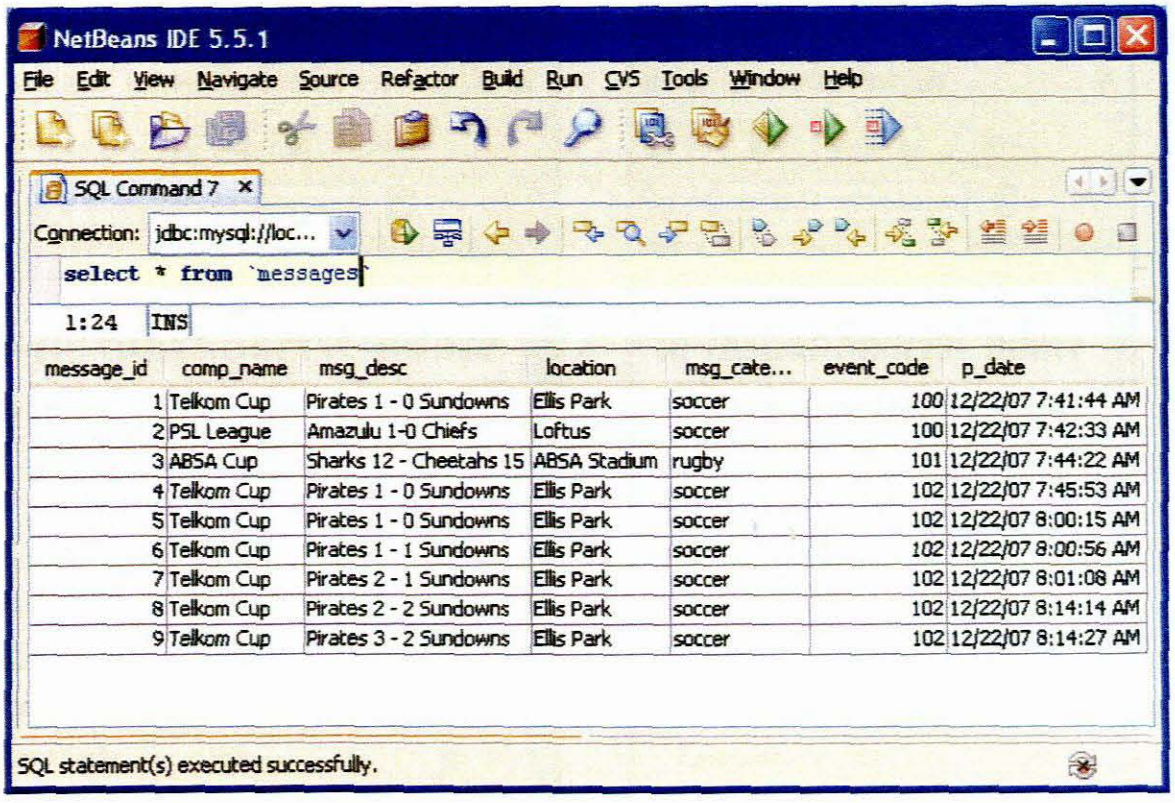
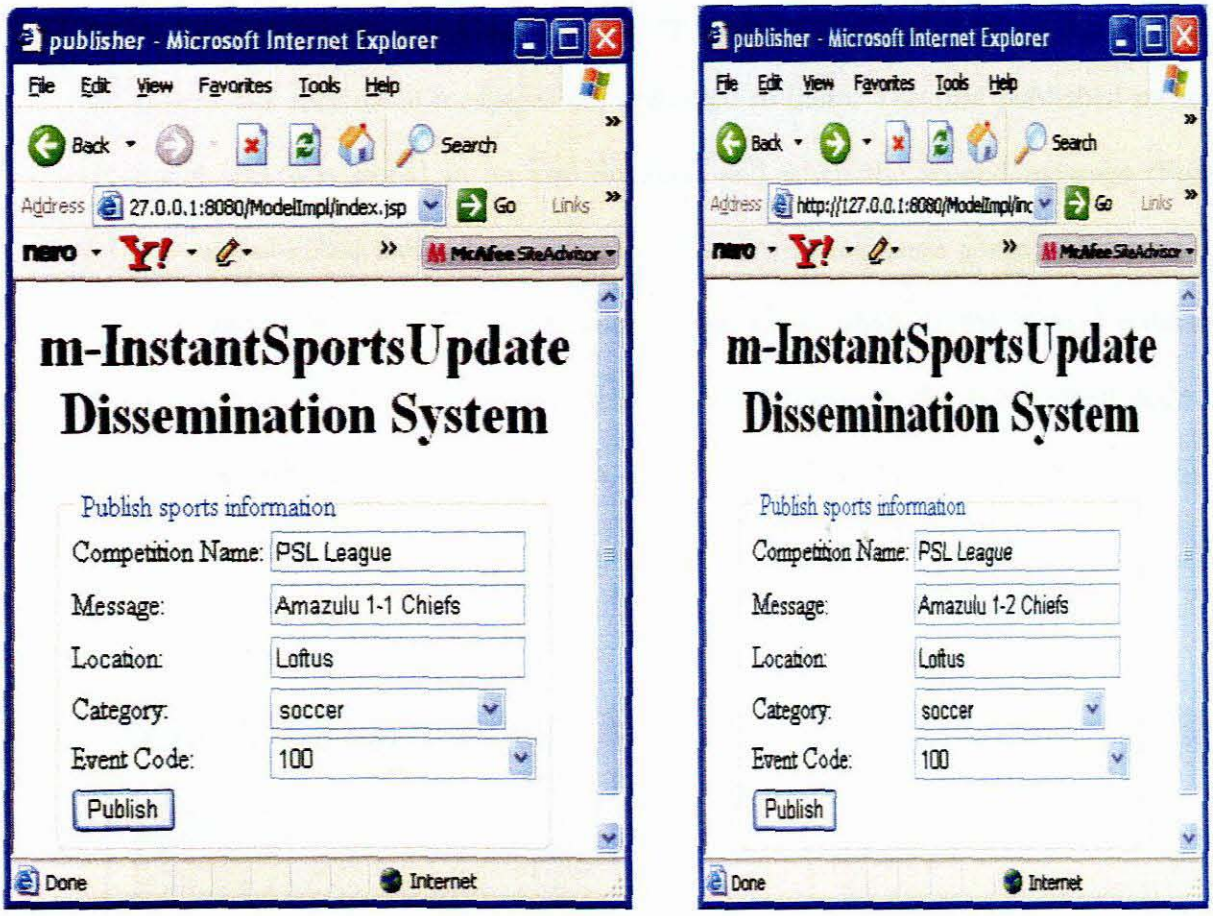



**Figure 4.13: Published messages database**

Figure 4.13 shows the database where the published messages are stored after publishing for the record of the publisher. All published messages are stored in a database (figure 4.13), for each and every published event, an event code is assigned to each message by the publisher to avoid delivery confusion to subscribers. This event code is used in the xml database to eliminate duplication of messages delivered to a subscriber. In figure 4.13 two soccer and one rugby message updates were published with event codes 100 for soccer's first game message , 102 for soccer's second game message and 101 for the rugby message. This implies that only subscribers whose subscriptions match the category or preference will receive the message. In our case figure 4.7 and figure 4.12 receives the messages which matched their preferences during registration (Figure 4.3).




**a:** First Published Message

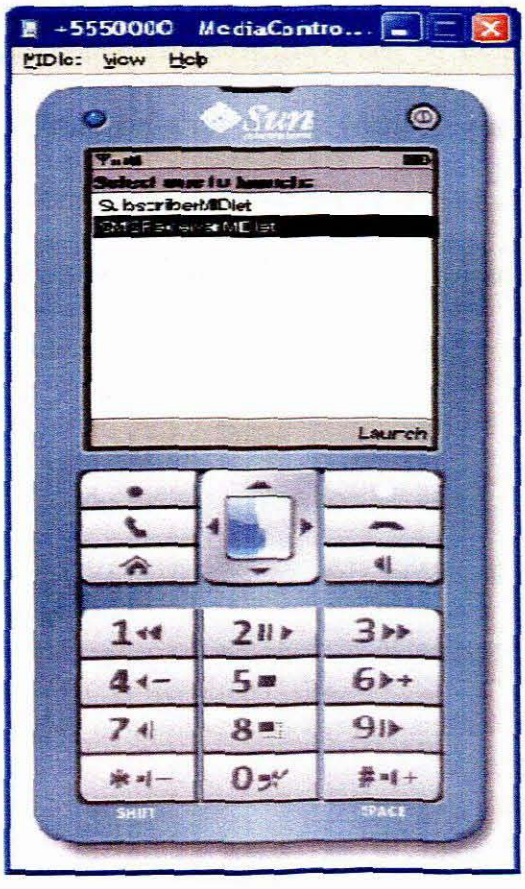
**b:** Second Published Message

**Figure 4.14: Publishing Events of the same event ode to a disconnected subscriber**

Figure 4.14 shows the publishing of events of the same code to an offline subscriber and the event code means that the published messages are of the same game. Here the messages are for the game between Amazulu and Chiefs and the messages (Figure 4.14a and Figure 4.14b) only have one difference which is the score update. These messages were supposed to be received by subscriber "5550000" (Figure 4.15a) who lost connection and was unable to receive the messages. Figure 4.15b shows the subscriber "5550000" reconnecting and receiving the dated message since the first published message (Figure 4.14a) was overwritten by the second message (Figure 4.14b). This

means that subscribers will not receive stale or outdated messages when they reconnect to the system but only dated messages are delivered to them. The first published message (Figure 4.14a) was stored in an xml database and when the second message (Figure 4.14b) was published the system recognized that it's of the same game it discarded the first one and only stored the latest score of the game which is the second published message. All published messages irrespective that they have the same event code are stored in the published messages database (Figure 4.13) for the publisher's record of all published messages for each game.

a: Disconnected subscriber "5550000" Reconnects




b: Subscriber "5550000"

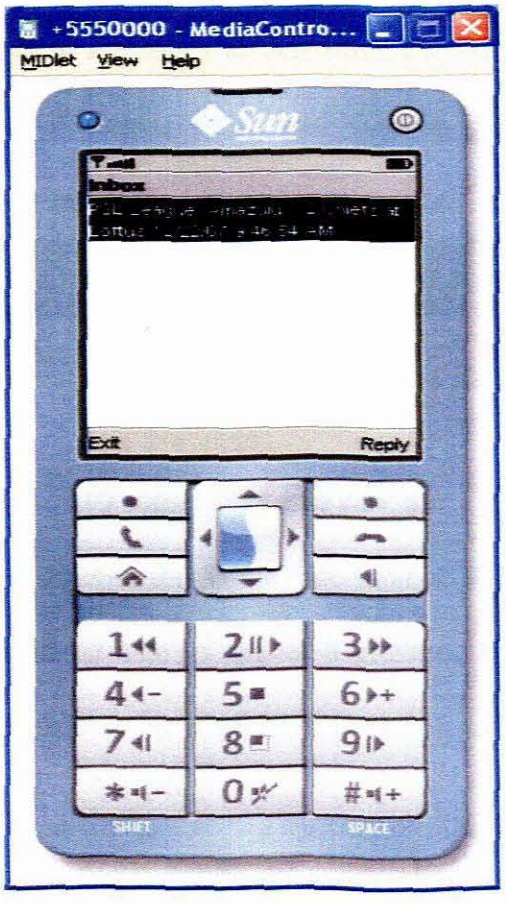



**Figure 4.15: Reconnected Subscriber receives the latest score update**

## 4.7 Performance Evaluation

In this section a description is presented of how performance evaluation for the Guaranteed Message Delivery model was carried out using the Instant-Sports Update (ISU) as our testbed. The results obtained are also presented. Message reliability and scalability were used to test the proposed model.

### 4.7.1 Message Delivery Reliability Results Evaluation

We used Microsoft Visual Studio 2005 as the simulation environment for our results. Any number of subscribers can be used for any number of messages to determine the performance of the system when the parameters (Number of Messages sent, and Number of Connected or online subscribers) increases. When the simulation is running, results evaluation is done in Microsoft Excel noting the number of messages sent and time taken to deliver these messages to intended subscribers.

We conducted the evaluation by using a constant number of 10 subscribers and vary the number of messages, we only varying the number of messages as this is the core subject in this research, we need to determine if the system will be able to deliver a greater number of messages in a short space of time. We therefore use a constant number of subscribers for our test of ensuring that subscribers will receive the events, someone might decide to use only two subscribers to experiment this. We therefore only differ the number of messages to test system scalability and reliability.

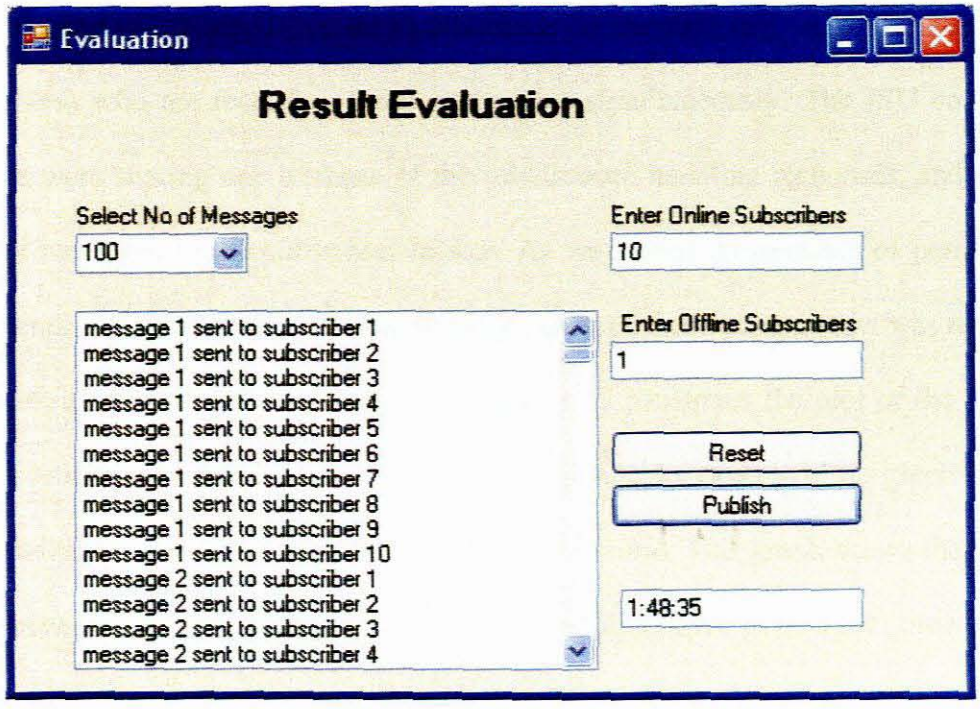



**Figure 4.16: Message Delivery Reliability Results Evaluation Interface**

Figure 4.16 shows the interface in a running mode which was used to evaluate our performance analysis. Online subscribers are subscribers which are in an active state, and ready to receive any published information or event. While disconnected subscribers are subscribers whose status are inactive and cannot receive any published information or event. The number of message shows the message to be delivered to online subscribers.

### 4.7.2 Message Throughput

The aim of this experiment was to measure the number of real-time events the system was able to handle in a given unit of time. In conducting this experiment two parameters were varied namely, the number of participating components and the number of real-time events or messages each component was able to send in a fixed time period. This parameter will also demonstrate how monitoring component affects the time of delivery. This was achieved by constantly increasing the number of ISU component instances that

were run simultaneously. This setup was done to emulate the idea of many clients (subscribers) who are receiving the same service simultaneously. The ISU component instances were sharing one instance of the middleware handling responses, and sending real-time messages to the subscriber broker. As we varied the number of participating components and the number of event messages each of these components was receiving, we observed how the system performed. Figure 4.17 Illustrates the plot of the message delivery reliability graph based on system throughput. As expected, the graph is linear and shows that our simulation guarantees delivery in time. This graph shows that there is no gap between the sending and receiving time and this prove to be a real time message delivery simulation. There is only minimal delivery delay which does not affect the subscriber. The smaller the number of components engaged in communication, the higher the throughput peak point value. This is a normal behavior of any distributed system.

Figure 4.17 gives the results of message delivery reliability and scalability. The number of messages was varied from 10 to 100, and the number of subscribers was kept at a constant value of 10 noting the time taken by the system to forward the messages to all the subscribers. In this case we define reliability in terms of delivery time based on the number of messages sent, and we define scalability in terms of number of messages sent per unit time. Now in this graph we observe that the system is not affected by the number of messages which convince us that scalability is being achieved in terms of messages delivered without any major delays for subscriber satisfaction.

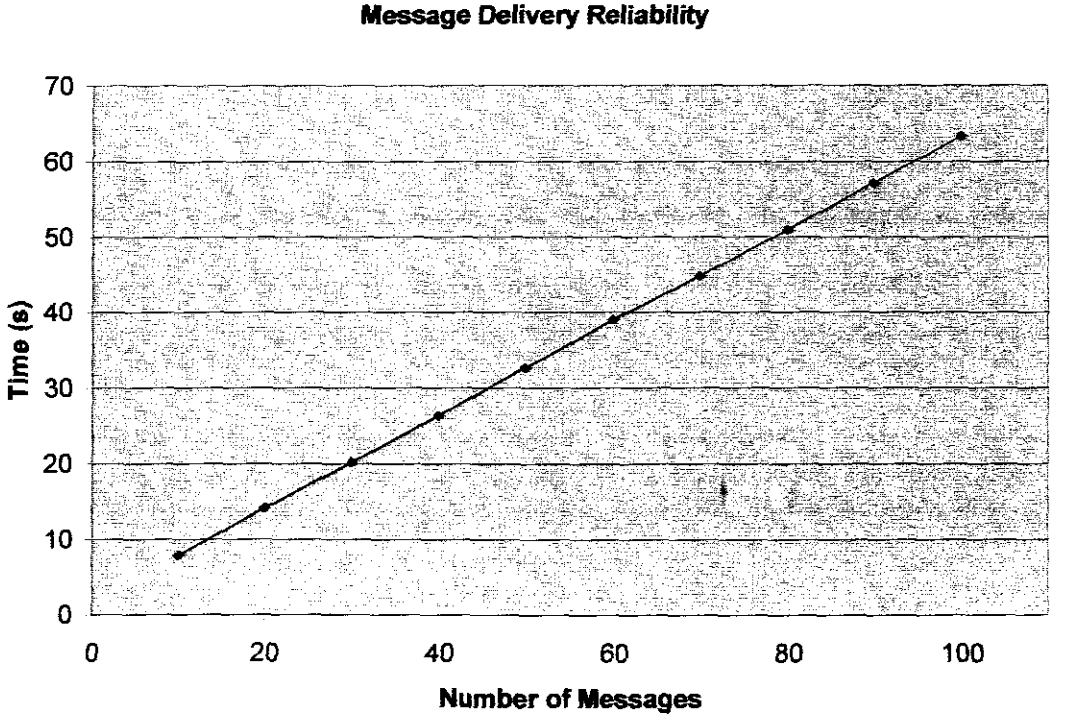



**Figure 4.17: Message delivery reliability based on the system throughput**

The graph shows that the system is not affected by the increase in the number of messages sent to registered subscribers. From the graph, if 20 messages are sent to 10 subscribers, the subscriber receives the messages in approximately 15s, and if 40 are sent, the subscriber receives them in approximately 25s. The difference of time in receiving 20 and 40 messages is 10s.

To conduct this experiment, two parameters were varied:

- Number of Messages and the
- Time taken by the system to send the message (Time in second).

From this experiment, we observed that the increase of number of subscribers on the system to receive notifications does not affect the performance or effectiveness of the system.

Based on the results obtained, it shows that the system is reliably based on the messages published and received by participating subscribers.

This chapter was based on the implementation of a prototype as a proof of concept. We proposed a scenario and also developed an implementation model to show the interaction of the components and modules discussed in chapter three, and we simulated and implemented the proposed scenario to show that the solution works. In the next chapter we conclude the whole work.

## 4.8 System and Connectivity Requirements

Windows 98 or NT or superior, Pentium processor at 90 MHz or higher, System memory (Hard drive): 40GB, RAM: 512MB or more, a Java wireless toolkit 2.5.1 or higher is needed to run the mobile devices or middlets to simulate the running of the system. Network scale and connectivity requirements: 2 personal computers, 1 to act as client, and the other as server and atleast 2 mobile devices to test if messages are received, and also if the device is able to receive messages after reconnection. Network cables to connect the PCs for communication.

UNIX and Windows are the platforms on which the Java effort has been the most focused. Consequently, Java seems to run more smoothly here. If you're using

Windows95, we recommend using Netscape's Navigator 4.5 as your browser. You basically have the choice of two browsers: Netscape Navigator (3.0 or 4.x) or Microsoft's Internet Explorer (also 3.0 or 4.x). Both are Java enabled. There are some other choices, Apple's Cyberdog, NeoPlanet, or Sun's HotJava browser (which happens to be written entirely in Java) the recommended browsers will help/used to run the server.

The requirements are not limited to the one listed above, a java expert might feel to use a different approach to achieve the same goal. Therefore, all requirements leading to the same goal are acceptable.

# CHAPTER FIVE

# CONCLUSION

## 5.1 Conclusion

The objective of this research was primarily to develop mechanisms to guarantee real-time message delivery for Publish/Subscribe Systems in a mobile environment that supports mobile users which will have the following features: a publisher portal that will be used by information producers to notify subscribers of different events, a mobile interface achieved by personalization for subscribers to register their preferences and also receive notifications. This objective of the research has been achieved by developing the Publish/Subscribe Message delivery model/mechanism that guarantees real-time delivery of messages to registered subscribers. The model achieved this by introducing a service called a Monitor which monitors disconnected subscribers, and monitors subscriber movements to ensure that the message is delivered to the registered subscriber/s. The monitor service is only active to locate a subscriber and also when there are disconnected subscribers. The other service is the 'Event manager', which ensures that messages being published are not lost for disconnected subscribers, this service communicates with the monitor service and the notifier service for successful delivery of a message. In our model we also included a matching manager which will match any published information with the subscriber's preference. The matching manager filters unwanted messages and ensures that a subscriber will not receive any message that he/she had not registered for,

but only messages that the subscriber registered for will be delivered. The features that were achieved in order to fulfill the objective of this research are the publisher portal interface for publishing events to subscribers and a mobile interface for subscribers to register and receive any published events in the form of messages (sms) using their mobile devices.

The current and most publish/subscribe systems discussed in chapter two do not give any assurance that a message will be delivered to all subscribers. Some employ a send and forget approach, where a message or event is sent without really guaranteeing that the message will be delivered to message consumers (Subscribers). In our work we proposed and proved the use of a monitoring service that will monitor subscriber migration from broker to broker and also monitor the status of each subscriber during message notification to all registered subscribers. This service ensures that only subscribers who are online (active) at time of delivery receive a published message, and offline (inactive) subscribers will only receive the published message stored in persistence storage for retrieval when the subscriber regains its active status.

Our proposed architecture in chapter three addresses most of the problems identified in chapter one, all issues are implicit in the architecture except the issue of reliable message delivery which we took a further step by developing a simulation for evaluating results of message delivery to determine reliable and on time message delivery. From the results obtained, we observed that our system is robust in terms of reliability, and the system is also scalable, in a sense that it is not affected by the number of subscribers joining, and

leaving the network. In terms of messages sent, we observe a linear graph, which shows that in terms of number of messages sent, the system remains scalable.

Persistence of information is very crucial in this study in order to ensure mobile users do not loose information when temporarily disconnected from the network [Linda, 2006], which we have achieved in this work. The scope of this work is broad since it addresses most issues that are concerned with customer satisfaction by ensuring message delivery, and also eliminating the issue of duplicating messages to one subscriber and also this work addresses the issue of disconnected subscriptions during message dissemination (delivery) by ensuring that subscribers do not lose messages pushed to them during their disconnection period from the network and we also addressed the issue of stale or outdated messages which can result in a subscriber receiving outdated messages, We therefore included elimination of stale messages in our implementation so that subscribers would always receive updated and latest messages.

As in our Instant Sports update where subscribers are updated of current scores of games in progress and when a subscriber unintentionally or intentionally disconnect from the network, All latest published scores are delivered to subscriber. Let us presume that during a soccer match which is in progress between team A and team B and one or more subscribers who selected to be notified of soccer scores are disconnected. From the match team A scores a goal (i.e. team A 1- 0 team B), A message is then published to update all registered subscribers and when team B equalises (i.e. team A 1-1 team B) the updated score overwrites the first published score message. This implies that disconnected subscribers would only receive the most updated message when reconnecting back to the

network. The implementation and the simulation serves only as a proof of the concept for ensuring guaranteed and reliable message delivery for the publish/subscribe paradigm.

## 5.2 Contribution

This dissertation attempts to realize the issues of reliable message delivery for publish/subscribe systems in a mobile environment, where mobile users migrate from broker to broker and during this period subscribers or information consumers might miss crucial events published during their offline or inactive period. Subscribers cannot afford to lose events published when they were disconnected from the network due to a number of factors, ranging from power off, network connectivity due to weak signals or mobile device switched off. The proposed guaranteed delivery model for events to subscribers ensures that subscribers will not lose any information during disconnection periods, but will rather receive any published event after a successful reconnection and also without receiving any outdated events which is discarded before is sent. In this case subscribers can always move anywhere, anytime without the worry of loosing events when no connection is maintained during their movement.

The implemented Instant Sports Update system can be used during world cups (soccer, rugby, cricket, etc) to update registered supporters of the most up to date scores for every game playing. This implies that even sports supporters who do not have TVs and Radios will be able to receive instant score updates of their favorite teams.

## 5.3 Future Work

This study is based on software but it can also be extended to the networking environment by adopting the same proposed message delivery model for publish/subscribe communication paradigm for content dissemination. This would enable the researcher to identify some algorithm that would be tested with the following metrics, minimal processing load, minimal bandwidth consumption, network congestion and notification delay to solve the performance of the publish/subscribe systems in a dynamic environment especially with mobile clients by addressing the issues concerning networking (network congestion and overloading).

To move the research from software to networking, there is a need to consider issues like node mobility, by this we refer to the movement of subscribers from one network location to another, usually referred to as handoff management, which occurs when a mobile subscriber changes cells (brokers), and also by investigating on issues based on network traffic which are the main cause of event delay during event dissemination. This work can also be extended by conducting research on issues of network delays and setup an experiment varying the number of subscribers to a sufficiently large number and record the time taken by the system to send the message. And also compare with varying the number of messages sent then also records the effect of increasing the number of users of messages sent.

# REFERENCES


Aguilera, M. K. Strom, R. E. Sturman, D. C. Astley, M. and Chandra, T. D. Matching events in a content-based subscription system. In Proceedings of the Principles of Distributed Computing, 1999, pages 53–61, May 1999.

Bacon, J. Moody, K. Bates, J. Hayton, R. Ma, C. McNeil, A. Seidel, O. and Spiteri, M. Generic support for distributed applications. IEEE Computer, 33(3):68–76, March 2000.

Banavar, G. Chandra, T. Mukherjee, B. Nagarajarao, J. Strom, R. E. and Sturman, D. C. An efficient multicast protocol for content-based publish-subscribe systems. In Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, 1999.

Behnel, S. Fiege, L. Muhl, G. On Quality-of-Service and Publish-Subscribe, ICDCS Workshops 2006.

Bhola, S. Strom, R. Bagchi, S. Zhao, Y. and Auerbach, J. Exactly-once Delivery in a Content-based Publish-Subscribe System, Proceedings of The International Conference on Dependable Systems and Networks, 2002.

Bygdås, S.S. Malm, P.S. and Urnes T. A Simple Architecture for Delivering Context Information to
Mobile Users. Telenor Research and Development, Norway, 2001

Cao, J. Feng, X. Lu, J. Chan, H. Das, S.K. "Reliable Message Delivery for Mobile Agents: Push or Pull," *icpads*, Ninth International Conference on Parallel and Distributed Systems (ICPADS'02), 2002

Caporuscio, M. Carzaniga, A. and Wolf, A. L. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. IEEE Transactions on Software Engineering, December 2003.

Carzaniga, A. Rosenblum, D. S. and Wolf, A. L. Achieving scalability and expressiveness in an internet scale event notification service. In Proceedings of the 19th ACMSymposium on Principles of Distributed Computing, ACM Press, July 2000.

Castro, M. Druschel, P. Kermarrec, A. and Rowston, A. Scribe: A large-scale and decentralized application-level multicast infrastructure, IEEE Journal on Selected Areas in Communications 20 (October 2002).

Chand, R. Felber, P. XNET: a Reliable Content Based Publish Subscribe System. SRDS 2004, 23rd Symposium on Reliable Distributed Systems, October 18-20 2004 – Florianopolis, Brazil

Cugola, G. and Munoz de Cote, J.E. On introducing location awareness in publish subscribe middleware. In Proc. of the 4th Int. Workshop on Distributed Event-Based Systems (DEBS), June 2005.
Cugola, G, Nitto, E. D. and Fuggetta, A. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Transactions on Software Engineering, September 2001.

Cugola, G. and Nitto, E. D. Using a publish/subscribe middleware to support mobile computing. In Proceedings of the Advanced Topic Workshop on Middleware for Mobile Computing, in association with IFIP/ACM Middleware 2001, November 2001.

Enterprise Integration Patterns, 2005, http://www.eaipatterns.com, last accessed: *15/10/2007*

Eugster, P. Pascal, TH. Felber, Rachid Guerraoui A. Anne-Marie Kermarrec The Many Faces of Publish/Subscribe, ACM Computing Surveys, June 2003.

Fiege, L. G¨artner, F. C. Kasten, C. and Zeidler, A. Supporting mobility in content-based publish/subscribe middleware. In Proceedings of the ACM/IFIP/USENIX International Middleware Conference (Middleware 2003), Springer- Verlag, June 2003.

Gryphon Web Site, http://www.research.ibm.com/gryphon, last accessed: 10/10/2007

Gudgin, M. H. Mendelsohn, N. Moreau J-J. Nielsen H. *SOAP Version 1.2 Part 1: Messaging Framework*, W3C Working Draft M. 26 June 2002 (See http://www.w3.org/TR/2002/WD-soap12-part1-20020626, last accessed: 05/06/2007)

Huang, Y and Garcia-Molina, H. Publish/subscribe in a mobile environment. In Proceedings of the 2nd ACM InternationalWorkshop on Data Engineering forWireless and Mobile Access (MobiDE'01), May 2001.

IBM, Websphere mq, http://www.ibm.com/websphere, 2003, last accessed: 01/11/2007

Jerzak, Z. Fetzer, C. Handling Overload in Publish/Subscribe Systems, Dresden University of Technology D-01062, Proceedings of ICDCS'2006 DEBS workshop 2005.

Julien, C. Roman, G-C. EgoSpaces: Facilitating Rapid Development of Context-Aware Mobile Applications. IEEE Trans. Software Eng. 2006.

Merriam-Webster. http://www.m-w.com/, last accessed: 23/11/2007

Muhl, G. Large-Scale Content-Based Publish/Subscribe Systems, Phd thesis, Technical University of Darmstadt, 2002.

ObjectWeb Open Source Middleware. JORAM - Java Open Reliable Asynchronous Messaging (release 3.6.0), August 2003. http://www.objectweb.org/joram, last accessed: 17/11/2007

Oki, B. Pfluegel, M. Siegel, A. and Skeen, D. "The Information Bus – An Architecture for Extensive Distributed Systems." Proceedings of the 1993 ACM Symposium on Operating Systems Principles, December 1993.

Opyrchal, L. Astley, M. Auerbach, J. S. Banavar, G. Strom, R. E. and Sturman, D. C. Exploiting IP multicast in content-based publish-subscribe systems, Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000), 2000,

PardoCastellote, G. Schneider, Hamilton, S.M. "NDDS: The real-time Publish-Subscribe Middleware", OMG document Innovations, 1999

Pietzuch, P and Bacon, J. Hermes: a distributed event-based middleware architecture, Proceedings of the International Workshop on Distributed Event-Based Systems (DEBS'02), 2003.

Platt, R. D. Understanding COM+, Microsoft Press, 2001.

Podnar, I. Hauswirth, M. and Jazayeri, M. Mobile Push: Delivering content to mobile users. In Proceedings of the 22nd International Conference on Distributed Computing Systems - Workshops (ICDCS 2002 Workshops), IEEE Computer Society, July 2002.

Podnar, I and Pripuzic, K. m-NewsBoard: A news dissemination service for mobile users. In Proceedings of the 7th International Conference on Telecommunications (ConTEL 2003), FER, University of Zagreb, June 2003.

Riabov, A. Zhen, L. Wolf, J.L. Yu, P.S. and Zhang, L. Clustering algorithms for content-based publication-subscription systems, Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, 2002.

Riabov, A. Zhen, L. Wolf, J.L, Yu, P.S, Zhang, Li. New algorithms for content-based publication-subscription systems, proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS'03), 2003.

Rowstron, A and Druschel, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems, Proceedings of International Conference on Distributed Systems Platforms (Middleware), 2001.

SIENA Web Site, http://www.cs.colorado.edu/users/carzanig/siena, last accessed: 10/12/2006

Softwired. iBus//Mobile developer's manual release 3.1., August 2002. http://www.softwired-inc.com, last accessed: 13/07/2007

Sutton, P. Arkins, R. and Segall, B. Supporting disconnectedness—Transparent information delivery for mobile and invisible computing. In Proc. of the IEEE Int. Symp. on Cluster Computing and the Grid, May 2001.

Wang, J. Cao, J. Li, J. Supporting Mobile Clients in Publish/Subscribe Systems. ICDCS Workshops 2005.

WS-Oxygen Tank. http://wso2.org/library/335, 2006, last accessed: 15/10/2007

Yoneki, E and Bacon, J. Pronto: MobileGateway with publish-subscribe paradigm over wireless network. Technical Report UCAM-CL-TR-559, Computer Laboratory, University of Cambridge, 2003.

Zeidler, A and Fiege, L. Mobility support with REBECA. In Proceedings of the 23rd International Conference on Distributed Computing Systems - Workshops (ICDCS 2003 Workshops), May 2003.

# APPENDIX A

## SOURCE CODE

### Business Layer classes

#### PublishedMessage.java: This class represents the structure of a published message

```
package za.ac.uzulu.cs.coe.businessobjects;


public class PublishedMessage
{
        private int messageID;
        private String competitionName;
        private String description;
        private String location;
        private String category;
        private int eventcode;
        private int priority;

        public PublishedMessage()
        {
        }

        public int getMessageID()
        {
                return messageID;
        }

        public void setMessageID(int messageID)
        {
                this.messageID = messageID;
        }

        public String getCompetitionName()
        {
                return competitionName;
        }

        public void setCompetitionName(String competitionName)
        {
                this.competitionName = competitionName;
        }

        public String getDescription()
        {
                return description;
        }

        public void setDescription(String description)
        {
                this.description = description;
        }

        public String getLocation()
        {
                return location;
        }

        public void setLocation(String location)
        {
                this.location = location;
        }
        public String getCategory()
        {
                return category;
```

```
        }

        public void setCategory(String category)
        {
                this.category = category;
        }

        public int getEventcode()
        {
                return eventcode;
        }

        public void setEventcode(int eventcode)
        {
                this.eventcode = eventcode;
        }

        public int getPriority()
        {
                return priority;
        }

        public void setPriority(int priority)
        {
                this.priority = priority;
        }

}
```

## MessageServer.java: This class represents the messaging server responsible for message dissemination to potential subscribers.

```
package za.ac.uzulu.cs.coe.placing;
import com.sun.kvem.midp.io.j2se.wma.*;
import com.sun.kvem.midp.io.j2se.wma.client.WMAClient;
import com.sun.kvem.midp.io.j2se.wma.client.WMAClient.MessageListener;
import com.sun.kvem.midp.io.j2se.wma.client.WMAClientFactory;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Properties;
import za.ac.uzulu.cs.coe.businessobjects.PublishedMessage;
import za.ac.uzulu.cs.coe.dataaccesslogic.SubscriberDB;
import za.ac.uzulu.cs.coe.dataaccesslogic.XMLReader;

public class MessagingServer implements  MessageListener
{

    private final int SMS_PORT  = 50000;
    private  WMAClient messageHandler;
    String title =null;

    ArrayList conected = new ArrayList();
    ArrayList msgQueue = new ArrayList();

//    private static final int HIGH = 0;
//    private static final int MEDIUM = 1;
//    private static final int LOW = 2;

    String priority;

    public MessagingServer()
    {

        try
        {

            Properties properties = System.getProperties();
            properties.put("kvem.home","D:\\WTK2.5.1");
            messageHandler = WMAClientFactory.newWMAClient("+1111000",WMAClient.SEND_AND_RECEIVE);
            messageHandler.connect();
```

```
            messageHandler.setMessageListener(this);

        }
        catch(Exception ex)
        {
            ex.printStackTrace();
        }

    }
    public void sendMessage(String topicMessage,String destAddress)
    {

        Message message = new Message(topicMessage);
        message.setToAddress("sms://" + destAddress + ":" + SMS_PORT);
        message.setFromAddress("sms://" + messageHandler.getPhoneNumber() + ":" +SMS_PORT);

        try
        {

            messageHandler.send(message);
            System.out.println("Message sent....");
            //JOptionPane.showMessageDialog(null,"message sent"+message.toString());
        }
        catch (Exception e)
        {

            System.err.println("Messsage not sent");
            e.printStackTrace();
        }

    }

    public String getPhone(String address)
    {
        return address.substring(6);
    }
    public void notifyIncomingMessage(WMAClient client)
    {

        try
        {
            WMAMessage mess = messageHandler.receive();

            if (mess instanceof Message)
            {
                Message message = (Message)mess;


                if (message.isSMS())
                {
                    title = getPhone( message.getFromAddress()) ;
                    System.out.println(title);
                    conected.add(title);
                    UpdateDB(title,message.toString());
                    System.out.println(message.toString() + "hello");
                }

            }
        }
        catch (IOException ioe)
        {
            System.err.println("Caught executing:");
            ioe.printStackTrace();
        }
    }
    public void UpdateDB(String phone, String message)
    {
        SubscriberDB db = new SubscriberDB();
        if(message.equals("close"))
        {
            db.updateStatus(phone,"inactive");
        }
```

```
        else if(message.equals("active"))
        {
            XMLReader read = new XMLReader();
            db.updateStatus(phone,"active");

            ArrayList messages = read.fromXML(phone);

            read.DeletefromXML(phone);

            if(messages.size() > 0)
            {

                for(int i = 0;i < messages.size();i++)
                {
                    PublishedMessage msg = (PublishedMessage)messages.get(i);

                    String category = msg.getCategory();
                    int priority = db.getPriority(category,phone);
                    msg.setPriority(priority );
                    msgQueue.add(msg);


                }

               PublishedMessage [] msgs = new PublishedMessage[msgQueue.size()];

                //Loading the
                for(int x = 0;x < msgs.length;x++)
                {
                    msgs[x] = (PublishedMessage)msgQueue.get(x);
                }
                //sorts
                for(int y = 0; y < msgs.length; y++)
                {
                    for(int z = y + 1; z < msgs.length; z++)
                    {
                        if(msgs[y].getPriority() < msgs[z].getPriority())
                        {
                            PublishedMessage mBean = msgs[y];

                            msgs[y] = msgs[z];

                            msgs[z] = mBean;
                        }
                    }
                }//
                for(int k = 0; k < msgs.length;k++)
                {
                    this.sendMessage(msgs[k].getDescription(),phone);
                    System.out.println(msgs[k].getDescription());
                }

            }
        }
    }

}
```

## MessageSender.java: This class harness the capabilities exposed by the messaging server in sending messages to potential subscribers.

```
package za.ac.uzulu.cs.coe.placing;

import java.util.ArrayList;
```

```
import za.ac.uzulu.cs.coe.dataaccesslogic.XMLReader;

/**
*
* @author UZ034232
*/
public class MessageSender
{
    private XMLReader xmlReader =null;
    private MessagingServer messagingServer = null;

   // boolean pass = false;
    public MessageSender()
    {
        xmlReader = new XMLReader();
        messagingServer = new MessagingServer();
    }
    public void saveMessage(String comp_name,String msg_desc,String location,String category,int event_code, String phone,
String dateStamp)
    {

       // xmlReader.deleteMessageFromXML(event_code);

        String message = comp_name +"  " + msg_desc + " at " + location + " " + dateStamp;

        xmlReader.toXML(event_code,message,category,phone);

    }
    public void sendMessage(String comp_name, String msg_desc,String location,String phone,String dateStamp)
    {

        String message = comp_name + "  " + msg_desc + " at " + location +" " + dateStamp;

        messagingServer.sendMessage(message,phone);

    }

}
```

## Register. java: This class handles of new registrations from subscribers.

```
package za.ac.uzulu.cs.coe.subscriptions;

import za.ac.uzulu.cs.coe.dataaccesslogic.SubscriberDB;
import java.io.*;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.swing.JOptionPane;


public class Register extends HttpServlet
{

    SubscriberDB subDB = new SubscriberDB();
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        response.setContentType("text/plain");
        PrintWriter out = response.getWriter();
        if(request.getHeader("userAgent") != null)
        {
            if(request.getHeader("UserAgent").indexOf("MIDP") != -1)
            {
                String name = request.getParameter("name");

                String surname = request.getParameter("surname");
                String phone = request.getParameter("cellNumber");
                String email = request.getParameter("emailAddress");
                String [] topics = request.getParameterValues("topic");
```

```
                String cell = "+"+phone.trim();

                System.out.println(cell);



                if( !name.equals("") && !surname.equals("") && !cell.equals("") && !email.equals("")  && topics != null)
                    subDB.insertSubscriber(cell,name,surname,email,topics);

                out.println("success");
                out.flush();
            }
        }
        out.close();
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the code.">
    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        processRequest(request, response);
    }

    /** Handles the HTTP <code>POST</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        processRequest(request, response);
    }

    /** Returns a short description of the servlet.
     */
    public String getServletInfo()
    {
        return "Short description";
    }
    // </editor-fold>
}
```

## MessagePublisher.java: This class handles publications from publishers.

```
package za.ac.uzulu.cs.coe.matchmaker;

import za.ac.uzulu.cs.coe.placing.MessageSender;
import java.io.*;
import java.util.ArrayList;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.swing.JOptionPane;
import za.ac.uzulu.cs.coe.dataaccesslogic.MessageDB;
import za.ac.uzulu.cs.coe.dataaccesslogic.SubscriberDB;
import za.ac.uzulu.cs.coe.dataaccesslogic.XMLReader;

/**
 *
 * @author UZ034232
 * @version
 */
public class MessagePublisher extends HttpServlet
{
```

```
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
    }
    protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();

        SubscriberDB subDB = new SubscriberDB();
        MessageDB msgDB = new MessageDB();
        MessageSender msg_sender = new MessageSender();
        DateTime dateInstance = new DateTime();
        XMLReader r = new XMLReader();

        if(request.getParameter("comp_name") != null && request.getParameter("msg_desc") != null &&
request.getParameter("location") != null && request.getParameter("category") != null && request.getParameter("eventcode") != null)
        {
            String comp_name = request.getParameter("comp_name");
            String msg_desc = request.getParameter("msg_desc");
            String location = request.getParameter("location");
            String category = request.getParameter("category");
            String event_code = request.getParameter("eventcode");



            ArrayList inactiveSubscribers = subDB.retrieveInactive(category);

            ArrayList activeSubscribers = subDB.retrieveActive(category);



            if(inactiveSubscribers.size() > 0)
            {
                for(int i =0; i < inactiveSubscribers.size(); i++)
                {
                    String phone = ""+inactiveSubscribers.get(i);

                    int code = Integer.parseInt(event_code);

                    r.deleteMessageFromXML(code);


msg_sender.saveMessage(comp_name,msg_desc,location,category,code,phone,dateInstance.getDateTimeStamp());

                }
            }
            // get phone subscribe and currently connected and send published info to them
            if(activeSubscribers.size() > 0)
            {
                for(int i =0; i < activeSubscribers.size(); i++)
                {

                    String phone = ""+activeSubscribers.get(i);
                    msg_sender.sendMessage(comp_name, msg_desc , location , phone,dateInstance.getDateTimeStamp());

                }
            }
            // add published info into database
            msgDB.persist(comp_name , msg_desc , location , category , Integer.parseInt(event_code)
,dateInstance.getDateTimeStamp());
            // call a jsp page correct
            RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/success.jsp");
            dispatcher.forward(request, response);

        }
        else
        {
            RequestDispatcher dispatcher = getServletContext().getRequestDispatcher("/error.jsp");
```

```
            dispatcher.forward(request, response);
        }

        out.close();
    }

    // <editor-fold defaultstate="collapsed" desc="HttpServlet methods. Click on the + sign on the left to edit the code.">
    /** Handles the HTTP <code>GET</code> method.
     * @param request servlet request
     * @param response servlet response
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        processRequest(request, response);
    }

    /** Handles the HTTP <code>POST</code> method
     * @param request servlet request
     * @param response servlet response
     */
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
        processRequest(request, response);
    }

    /** Returns a short description of the servlet.
     */
    public String getServletInfo()
    {
        return "Short description";
    }
    // </editor-fold>
}
```

## Data Access Layer classes

### MessageDB.java: This class handles the persistence of published messages

```
package za.ac.uzulu.cs.coe.dataaccesslogic;
import java.sql.*;

/**
 *
 * @author UZ034232
 */
public class MessageDB
{

    DataSourceConnector connector  = new DataSourceConnector();
    Connection connection  = null;

    public MessageDB()
    {
        connection = connector.getConnection();
    }

    //Persisting the messages
    public void persist(String comp_name,String msg_desc,String location, String category,int eventCode , String dateTime)
    {

        try
        {

            connection.setAutoCommit(false);

            String query = "insert into messages ( comp_name , msg_desc , location , msg_category , event_code , p_date)
values(?,?,?,?,?,?)";
```

```
                PreparedStatement pstmt = connection.prepareStatement(query);

                pstmt.setString(1, comp_name);
                pstmt.setString(2, msg_desc);
                pstmt.setString(3, location);
                pstmt.setString(4, category);
                pstmt.setInt(5,eventCode);
                pstmt.setString(6,dateTime);

                int status = pstmt.executeUpdate();

                if(status == 0)
                {
                    connection.rollback();
                }
                else
                    connection.commit();
                connection.close();
            }
            catch (SQLException sqlex)
            {
                sqlex.printStackTrace();
            }

        }
}
```

## XMLReader.java: This class manages the persistence and retrieval of messages published during client disconnections.

```
package za.ac.uzulu.cs.coe.dataaccesslogic;


import java.io.File;
import java.util.ArrayList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Result;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.w3c.dom.Text;
import za.ac.uzulu.cs.coe.businessobjects.PublishedMessage;


public class XMLReader
{
                    //Parser Factory
                    DocumentBuilderFactory parserFactory;

                    //DOM Parser
                    DocumentBuilder parser;

                    //Writer factory
                    TransformerFactory writerFactory;

                    //XML Writer
                    Transformer writer;


                    public XMLReader()
                    {
```

```
                //Creating new parser factory instance
                parserFactory = DocumentBuilderFactory.newInstance();

                //parserFactory.setValidating(true);
                //parserFactory.setNamespaceAware(true);

                //Creating new writer factory instance
                writerFactory = TransformerFactory.newInstance();
                try
                {
                        //Getting the appropriate parser from a factory
                        parser =  parserFactory.newDocumentBuilder();

                        //Getting the writer from a factory
                        writer =  writerFactory.newTransformer();
                }
                catch(Exception ex)
                {
                        ex.printStackTrace();
                }
        }
        public void toXML(int event_code,String message,String category,String phone)
        {
File file = new File("D:/ModelImpl/messages/Message.xml");
if(file.exists())
{
    try
    {
        Document xmlDoc = parser.parse(file);

        Element root =  xmlDoc.getDocumentElement();



         Element messageElement = xmlDoc.createElement("message");

         Element evtCodeElement = xmlDoc.createElement("eventcode");
         evtCodeElement.appendChild(xmlDoc.createTextNode(String.valueOf(event_code)));
         messageElement.appendChild(evtCodeElement);

         Element categoryElement = xmlDoc.createElement("category");
         categoryElement.appendChild(xmlDoc.createTextNode(category));
         messageElement.appendChild(categoryElement);

         Element description = xmlDoc.createElement("product_info");
         description.appendChild(xmlDoc.createTextNode(message));
         messageElement.appendChild(description);

         Element phoneElement = xmlDoc.createElement("phone");
         phoneElement.appendChild(xmlDoc.createTextNode(phone));
         messageElement.appendChild(phoneElement);
         root.appendChild(messageElement);
        // xmlDoc.normalize();
        Source source = new DOMSource(xmlDoc);
         Result result = new StreamResult(file);

         writer.transform(source,result);
    }
    catch(Exception ex)
    {
        ex.printStackTrace();
    }
}
else
    System.out.println("File does not exists");
      }

      public ArrayList fromXML(String aPhone)
      {
ArrayList messages = new ArrayList();
File file = new File("D:/ModelImpl/messages/Message.xml");
```

```
            if(file.exists())
            {
                try
                {
                    Document xmlDoc =  parser.parse(file);

                    Element root = xmlDoc.getDocumentElement();

                    NodeList empList =  root.getElementsByTagName("message");

                    for(int i = 0;i < empList.getLength();i++)
                    {


                        Element msgElement = (Element)empList.item(i);


                        Element phoneElement = (Element)msgElement.getElementsByTagName("phone").item(0);
                        Text txtPhone = (Text)phoneElement.getFirstChild();
                        String phone = txtPhone.getNodeValue();

                        if(phone.equals(aPhone))
                        {
                            PublishedMessage m = new PublishedMessage();

                            Element catElement = (Element)msgElement.getElementsByTagName("category").item(0);
                            Text txtCat = (Text)catElement.getFirstChild();
                            String category = txtCat.getNodeValue();
                            m.setCategory(category);


                            Element descElement = (Element)msgElement.getElementsByTagName("product_info").item(0);
                            Text txtDesc = (Text)descElement.getFirstChild();
                            String desc = txtDesc.getNodeValue();
                            m.setDescription(desc);

                            messages.add(m);
                        }

                    }
                }
                catch(Exception ex)
                {
                    ex.printStackTrace();
                }
            }
            else
                System.out.println("File does not exists");
            return messages;
                }

        public void deleteMessageFromXML(int eventCode)
        {
            File file = new File("D:/ModelImpl/messages/Message.xml");
            if(file.exists())
            {
                try
                {
                    Document xmlDoc = parser.parse(file);

                    Element root = xmlDoc.getDocumentElement();
                    if(root.hasChildNodes())
                    {
                        NodeList messageList = root.getElementsByTagName("message");
                        for(int i = 0;i < messageList.getLength();i++)
                        {
                            Element messageElement = (Element)messageList.item(i);

                            Element evtCodeElement =
(Element)messageElement.getElementsByTagName("eventcode").item(0);

                            Text txtEvtCode = (Text)evtCodeElement.getFirstChild();
```

```
                    String evtcode = txtEvtCode.getNodeValue();

                    String parevtCode = String.valueOf(eventCode);

                    if(evtcode.equals(parevtCode))
                    {
                        messageElement.getParentNode().removeChild(messageElement);
                        xmlDoc.normalize();

                    }
                }
                Source source = new DOMSource(xmlDoc);
                Result result = new StreamResult(file);
                writer.transform(source,result);
            }
        }
        catch(Exception sqlex)
        {
            sqlex.printStackTrace();
        }

    }
    else
        System.out.println("File does not exists");
}
public void DeletefromXML(String phone)
        {
    File file = new File("D:/ModelImpl/messages/Message.xml");
    int len = 0;
    if(file.exists())
    {
        try
        {
            Document xmlDoc =  parser.parse(file);


            Element root = xmlDoc.getDocumentElement();

            NodeList msgList =  root.getElementsByTagName("message");

            len = msgList.getLength();

           // System.out.println(msgList.getLength());

            for(int i = 0;i < msgList.getLength();i++)
            {
                    Element messageElement = (Element)msgList.item(i);



                Element phoneElement = (Element)messageElement.getElementsByTagName("phone").item(0);
                Text txtPhoneElement = (Text)phoneElement.getFirstChild();
                String aPhone = txtPhoneElement.getNodeValue();
                if(aPhone.equals(phone))
                {

                  messageElement.getParentNode().removeChild(messageElement);

                }
            }
            xmlDoc.normalizeDocument();

            Source source = new DOMSource(xmlDoc);
            Result result = new StreamResult(file);
            writer.transform(source,result);


        }
        catch(Exception ex)
        {
            ex.printStackTrace();
```

```
            }
        }
        else
            System.out.println("File does not exists");
        if(len > 0)
          DeletefromXML(phone);
              }


}
```

## SubscriberDB.java: This class handles persistence and management of subscriber information.

```
package za.ac.uzulu.cs.coe.dataaccesslogic;
import java.sql.*;
import java.util.ArrayList;
/**
 *
 * @author UZ034232
 */
public class SubscriberDB
{
    private Statement stmt = null;
    private ResultSet rs = null;
    private Connection connection =null;
    DataSourceConnector connector = new DataSourceConnector();

    public SubscriberDB()
    {
        connection = connector.getConnection();
    }


    //Retrieves active subscriber
    public ArrayList retrieveActive(String category)
    {
        ArrayList activeSubscribersList = new ArrayList();
        try
        {

            String query = "select subscriber.sub_phone from subscriber inner join category on subscriber.sub_phone =
category.sub_phone where category.category_desc ='"+category+"' and subscriber.sub_status = 'active'";
            stmt = connection.createStatement();

            rs = stmt.executeQuery(query);

            while(rs.next())
            {

                activeSubscribersList.add(rs.getString("sub_phone"));

            }
            rs.close();
            stmt.close();

        }
        catch (SQLException sqlex)
        {
            sqlex.printStackTrace();
        }
        return activeSubscribersList ;
    }
    public void insertSubscriber(String phone,String name,String surname,String email,String [] topics)
    {
        String topic = null;
        String priority = null;
        try
```

```
        {
            connection.setAutoCommit(false);


            //Inserts into a subscriber table
            String insertSubQuery = "insert into subscriber values(?,?,?,?,?)";

            PreparedStatement pstmt_1 = connection.prepareStatement(insertSubQuery);

            pstmt_1.setString(1,phone);
            pstmt_1.setString(2,name);
            pstmt_1.setString(3,surname);
            pstmt_1.setString(4,email);
            pstmt_1.setString(5,"inactive");

            int result_1 =  pstmt_1.executeUpdate();

            if(result_1 == 0)
            {
                connection.rollback();
            }

            //Inserts into a category table
            String insertCatQuery = "insert into category ( sub_phone , category_desc , category_priority ) values(?,?,?)";

            PreparedStatement pstmt_2 = null;
            for(int i = 0 ; i < topics.length;i++)
            {
                pstmt_2 = connection.prepareStatement(insertCatQuery);

                int index = topics[i].indexOf(':');
                topic = topics[i].substring(0,index);
                priority = topics[i].substring(index+1,topics[i].length());

                pstmt_2.setString(1,phone);
                pstmt_2.setString(2,topic);
                pstmt_2.setString(3,priority);

                int result_2 = pstmt_2.executeUpdate();

                if(result_2 == 0)
                {
                    connection.rollback();
                }
            }
            connection.commit();
            pstmt_1.close();
            pstmt_2.close();

           }
        catch(SQLException sqlex)
        {
            sqlex.printStackTrace();
        }
    }
    public ArrayList retrieveInactive(String category)
    {
        ArrayList inActiveSubscribersList = new ArrayList();
        try
        {


            String query = "select subscriber.sub_phone from subscriber inner join category on subscriber.sub_phone =
category.sub_phone where category.category_desc = '"+category+"' and subscriber.sub_status = 'inactive'";

            stmt = connection.createStatement();

            rs = stmt.executeQuery(query);

            while(rs.next())
            {
```

```
                    inActiveSubscribersList.add(rs.getString("sub_phone"));

                }
                rs.close();
                stmt.close();

            }
            catch (SQLException sqlex)
            {
                sqlex.printStackTrace();
            }
            return inActiveSubscribersList;
        }
        public int getPriority(String category,String phone)
        {
            int pValue = 0;
            String priority = null;

            try
            {
                String query = "select category.category_priority from subscriber inner join category on subscriber.sub_phone =
category.sub_phone where category.sub_phone = " + phone + " and category.category_desc = '"+ category + "'";

                stmt = connection.createStatement();

                rs = stmt.executeQuery(query);

                while(rs.next())
                {
                        priority = rs.getString("category_priority");

                        if(priority.equals("High"))
                        {
                            pValue = 2;
                        }
                        else if(priority.equals("Medium"))
                        {
                            pValue = 1;
                        }
                        else if(priority.equals("Low"))
                        {
                            pValue = 0;
                        }
                }
                rs.close();
                stmt.close();
            }
            catch(SQLException sqlex)
            {
                sqlex.printStackTrace();
            }
            return pValue;
        }
        public void updateStatus(String phone,String status)
        {
            try
            {
                connection.setAutoCommit(false);

                String update = "update subscriber set sub_status ='"+status+"' where sub_phone ="+phone;

                stmt = connection.createStatement();

                int db_status = stmt.executeUpdate(update);

                if(db_status == 0)
                {
                    connection.rollback();
                }
                connection.commit();

                stmt.close();
```

```
            }
            catch(Exception sqlex)
            {
                sqlex.printStackTrace();
            }
        }

    }
```