A MESSAGE QUEUE BASED EVENT NOTIFICATION SYSTEM

FOOTBALL LOTTERY SYSTEM

Master of Science Thesis
University of Stavanger
The Department of Electrical and Computer Engineering

By
Xu Yunpeng
June 2010

Abstract

The event notification service enables user of getting informed about the occurrence of their events of interest. Message queue technology provides asynchronous message exchange functions between computer processes. This thesis presents a solution of building event notification system using the message queue approach. The events in the experiment are generated during the football match and used for both stateless and stateful processing.

The system includes three main applications: the football event publication system will provide sequence of events to the message broker; the football lottery client application will provide a user interface for the clients to make prediction about future matches; the football lottery server will accept user betting coupons from clients and subscribe match events from the message broker, base on the received events the server will make evaluation for each coupon and give out the result.

Acknowledgments

My supervisor, Mr Hein Meling, associate professor in Department of Electrical Engineering and Computer Science at the University of Stavanger gave me a lot of intensive guidance during the work. Mr Pål Evenson, PHD student of Department of Electrical Engineering and Computer Science at the University of Stavanger also provided many validate suggestion during the coding process. I would like to thank for their patiently assistance and kindly encourage.

Contents

Chapter 1

Introduction

Distribute system is widely used today to meet the web service application demands such as fault tolerance, load balance and sharing. A distribute system is generally a set of computers connected by internet and coordinate their actions by exchanging messages[20].

However, the computers in a distribute system may have different kinds of hardware and software. The communication protocol and internet hierarchy between the hardwire platform may also be distinct from each other. This heterogeneity brings great challenge for the design of distributed system. Message Oriented Middleware (MOM) provides a reasonable solution for this heterogeneity challenge. As it shows in Figure 1.1. This middleware lies between the platforms (computer hardware and operating system) and the applications. MOM communicates with different kind of platforms through corresponding platform interfaces to shied software developer from low lever platform details, it also provides a standard interface for all applications of upper lever to locate the applications transparency across the network[8]. the communication between the upper lever applications can be asynchronous, the applications no longer needs to keep the constant communication channel with other applications, instead messages are exchanged between the applications and middleware.

Message queue is the mechanism for MOM to store and transfer the messages exchanged between clients, it's asynchronous which make the message sender and receiver don't need to connect with the broker at the same time. The implementation of Message queue may include the enhanced resilience functionality in case of message loss due to system failure.

Java Message Service (JMS) is a standard interface for this kind of message

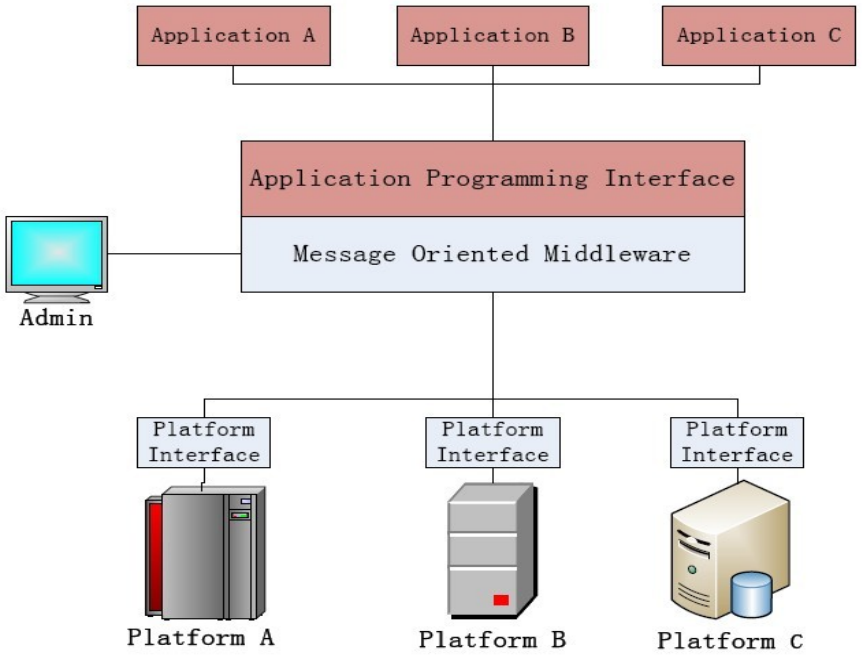

Figure 1.1: Middleware architecture

service. Many message brokers are developed base on JMS, such as *Sun MQ, BEA MQ, Apache ActiveMQ, IBM MQ* and so on[24]. ActiveMQ is used as the message broker in this solution.

1.1 The Idea

With the maturity of third generation wireless network, personal digital assistant and smart mobile phones are widely used. Many exciting smart phone applications are developed and provided to users. This thesis will focus on an implementation of such applications. Java is an Object Oriented Language which is widely used in programming. It's platform agnostic and easy to use in network programming. Java also supports multithread and has a standard API (JMS) for Message Oriented Middleware. These advantages make Java the most suitable language for the solution. Most smart phones on the market uses one of these operating systems: *Symbian, Windows Mobile, Android* and *Mac OS*. All of them support Java application except the Iphone, which uses *Mac OS*. Thus, it will be meaningful to develop both Java and C clients for different platforms.

There are a lot of football fans in Norway. They don't want to miss their favorite matches even when they are away from television, they also prefer a convenient way to make prediction about the future matches. The idea is developing a football event notification system for the clients using message queue approach. The contribution of the system should include two aspects:

(1) A live messages notification system capable of supporting a multitude of client platforms, including mobile phones like iPhone.

(2)A football lottery application that can be used to notify clients (customers) of events such as the number of correct on the lottery coupon of the user and the price need to be paid for the coupon.

1.2 This Thesis

This thesis will start with an introduction of the background which are essential for the design, including 3rd party software and technologies. Followed by a primary explanation of the application structure and implementation details. After that, the evaluation and the conclusion will be presented. At last, some plan for the future work will be suggested.

Chapter 2

Background

2.1 Java Message Service

A distribute server system is composed of a collection of heterogeneous networked computers. The internal communication which is between the inside computers and the external communication which is between the system and clients, are both based exchanging message. The message exchange can follow a lot of different paradigms such as *message passing, remote procedure call (RPC), notification, shared space, message queueing* and *publish/subscribe*. These paradigms are based on different lever of abstraction and have different performance on space, time and Synchronization decoupling. Space decoupling means communicating parties don't need to know each other, time decoupling means they don't need to "speak" or "listen" at the same time, synchronization decoupling means each side don't need to wait for other's reply [10].

RPC is a typical space, time and synchronization coupled communication paradigm which allows one computer process to cause a subroutine in another address space (commonly another computer in the shared network) to invoke services like performing native invocations, this results higher maintenance cost. Message queueing, which also referred to point to poing (PTP), is both space and time decoupled but synchronization coupled. Message producers put messages into a fixed destination and consumers pull messages from the destination, consumers need to wait if no message is available in the destination. Publish/subscribe mode is totally space, time and synchronization decoupled, the message sender puts message into a fixed topic and any subscriber of the topic and get a copy of the message. The maintenance cost of decoupled communication is much lower, this make PTP

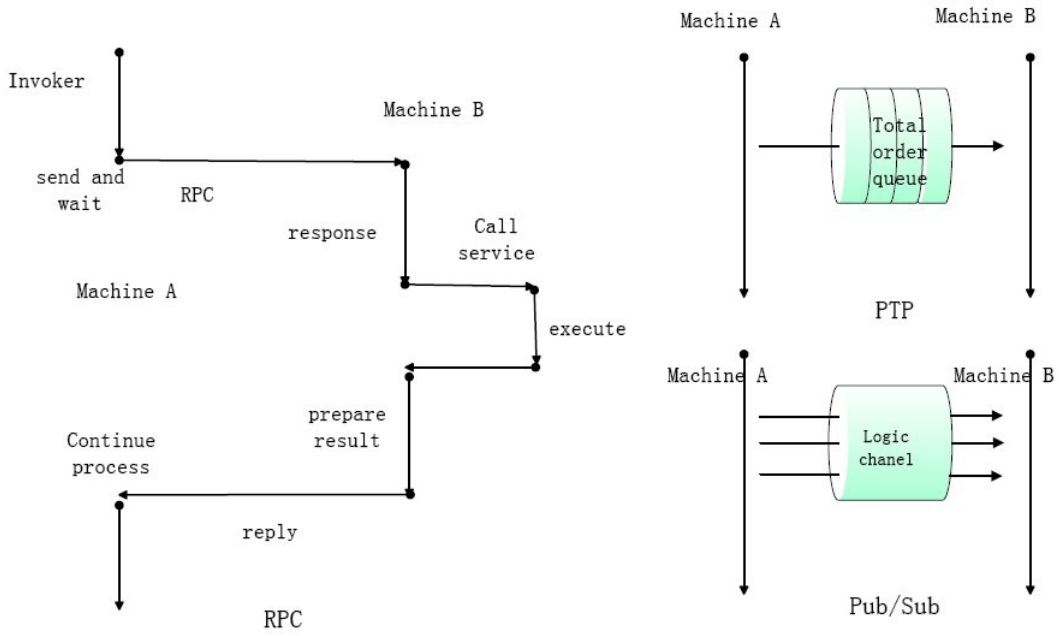

Figure 2.1: RPC and message passing

and Publish/Subscribe more suitable communication mode for event notification system. Figure 2.1 shows the compare of RPC, PTP and Publish/Subscribe.

Messaging oriented middleware (MOM) provides event based communication between programs which avoid the maintaining of direct channel between message producers and message consumers. Java Message Service (JMS) provides a standard java API for creating, sending, receiving and reading of messages[24]. JMS provides:

1, two kinds of communication model: PTP and Pubish/Subscribe

2, reliable message transport

3, transaction

4, message filtering mechanism

The structure of JMS is shown in Figure 2.2:

2.1.1 Some important terms in JMS

Below are some terms in JMS[21]:

JMS Provider: Message broker that implements JMS.

PTP: Point to Point messaging model provides durable buffering of message

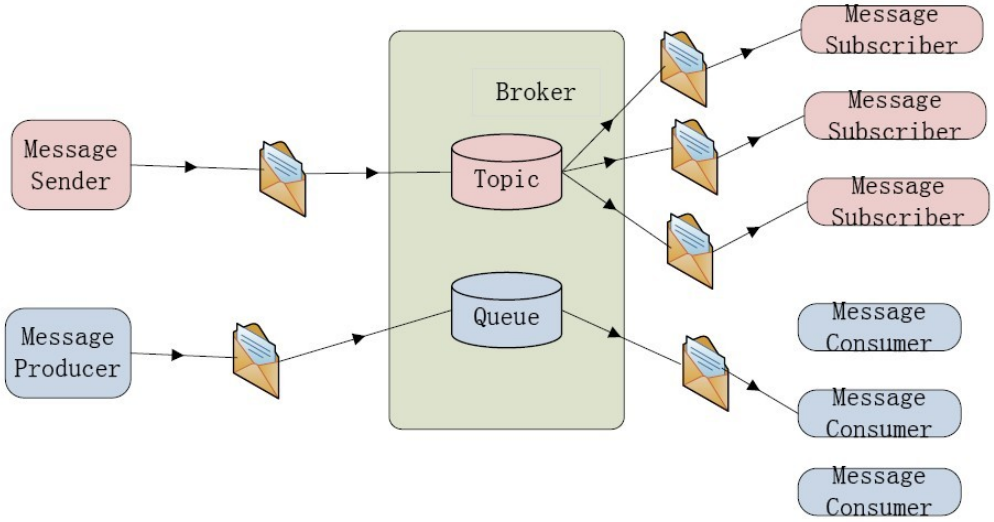

Figure 2.2: JMS structure

in queue.

Publish/Subscribe: Publish and subscribe messaging model, provides message multicast.

Queue: A message domain, contains message which can be consumed by only one consumer each time.

Topic: A message domain, contains message which can be consumed by multiple active subscribers at same time.

ConnetionFactory: Factory object used to establish connection.

Connection: The connection between client and message broker.

Destination: The message domain managed by JMS provider, which stores the message produced by clients.

Session: A thread that receives and sends messages.

Message producer: Object created by session to send message.

Message consumer: Object created by session to receive message.

2.1.2 Programming with JMS

In JMS, *ConnectionFactory* and *Destination* are administrated objects of JMS provider, which means JMS itself only provides the standard interface of these two classes, The JMS provider will implement these interfaces.

The first step of programming with JMS is to get the *ConnectionFactory* and *Destination* Object. Two methods are available for this purpose:

1 , Use Java Naming and Directory Interface (JNDI) to discover the *ConnectionFactory* and *Destination* objects, JNDI is a java API for a directory service that allows java software clients to discover and look up data and objects via a name.

2, Since we use ActiveMQ as the JMS provider, we can get *ConnectionFactory* and *Destination* by invoking the relevant constructers provided by ActiveMQ, this method is easy and adapt in the application.

The second step is to use *ConnectionFactory* to establish *Connection*. This *Connection* is the active communication channel between the JMS client and the JMS broker, The JMS broker will allocate relevant resource for handling the connection and verify the client.

Then single or multiple *Sessions* can be built by connection. *Session* is a thread context to create and process message. *Session* use the acknowledgement options and transactions to guarantee reliability.

After that *Session* and *Destination* can be used to construct *MessageProducer* and *MessageConsumer*. Client use *MessageProducer* to send message to certain physical target (*Destination*), the modes (durable or undurable), priority and expires of the messages, can also be set by the producer. Client use *MessageConsumer* to receive message from certain physical target (*Destination*), Message selector can be use to filter the messages received. *MessageConsumer* can use synchronous mode by invoke *receive()* method, or use asynchronous method by register a *MessageListener*. Finally, start the *Connection* to process messages. An example to subscribe the message.

```
import javax.jms.*;
import org.apache.activemq.*;
public class EventPublisher{
try{
        ConnectionFactory connectionfactory=new
            ActiveMQConnectionFactory();
        Connection connection=connectionfactory.createConnection();
        connection.start();
        final Session session=connection.createSession(Boolean.TRUE,
            Session.AUTO_ACKNOWLEDGE);
        Destination destination=session.createTopic("Viking");
        MessageConsumer consumer=session.createConsumer(destination);
        System.out.println(destination.toString());
```

```
    consumer.setMessageListener(new MessageListener(){
            public void onMessage(Message msg)
             {
             TextMessage message=(TextMessage)msg;
                try {
                        String detail=message.getText();
                                System.out.println("Hear Message:"+
                                    detail);
                                session.commit();
                                } catch (JMSException e) {
                                        e.printStackTrace();
                                }
             }
             });
           session.close();
           connection.close();
                 }
                 catch(Exception e)
                 {
                        e.printStackTrace();
                 }
}
```

2.2 ActiveMQ

Apache ActiveMQ is a very popular and powerful open source messaging and integrating patterns provider. It fully supports JMS 1.1 but not confined in Java.

ActiveMQ supports multiple language clients such as: *Java, C, C++, C#, Ruby, Python* and more, it shows in Figure 2.3. This opens to the door to many more opportunities where ActiveMQ can be utilized outside of just the Java world[22]. Thus a C client for Iphone can be developed.

ActiveMQ provides an Openwire protocol for high performance *Java, C, C++, C#* clients, and also provides Stomp protocol so that clients can be written easily in *C, Ruby, Python, Perl, PHP* and etc[3].

ActiveMQ provides a wide range of connectivity options including support for protocols such as *HTTP, SSL, TCP, STOMP,UDP, WS notification* and more (see Figure 2.3). This means ActiveMQ is very flexible.

ActiveMQ supports multiple kinds of message store such as *JDBC, Kaha, Journal, Caching* and etc, also the security lever of ActiveMQ can be completely cus-

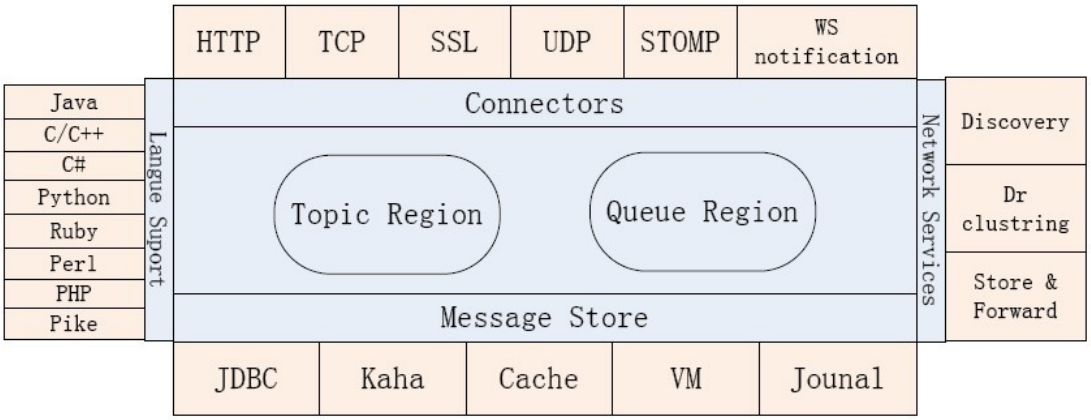

Figure 2.3: ActiveMQ architecture[3]

tomized (see Figure 2.3).

Many ActiveMQ broker can be organized into cluster for scalability purpose, this network of brokers can support various topologies. In this paper, the distributivity of Football Lottery Server is presented.

Unlike normal PC or workstation, the CPU process speed, the memory as well as the power of smart phone and personal data terminal are quite limited. Thus, it will be necessary for the smart phone applications to be simple and efficient. C langue is a procedure-oriented middle lever language that don't supports programming interface like Java. To develop a C subscriber client for Iphone, the whole communication protocol stack need to be designed. TCP provides reliable messaging in Transport layer, also both Iphone application and ActiveMQ broker totally support Tcp protocol. These make the TCP protocol an appropriate lower layer protocol for this solution. Openwire protocol is used upon TCP to provide command marshaling rules and procedures to establish/close connection, session and other entities. The ActiveMQ's Java implementation also uses TCP+Openwire as a typical protocol stack. Then an intensive study on the source code and functioning mechanism of ActiveMQ is meaningful.

2.2.1 The TCP transport mechanism of ActiveMQ

ActiveMQ supports multiple transports. Among which TCP transport is most common used. TCP means transmission control protocol, as one of the core protocols of the Internet Protocol Suite, TCP provides reliable message transmission

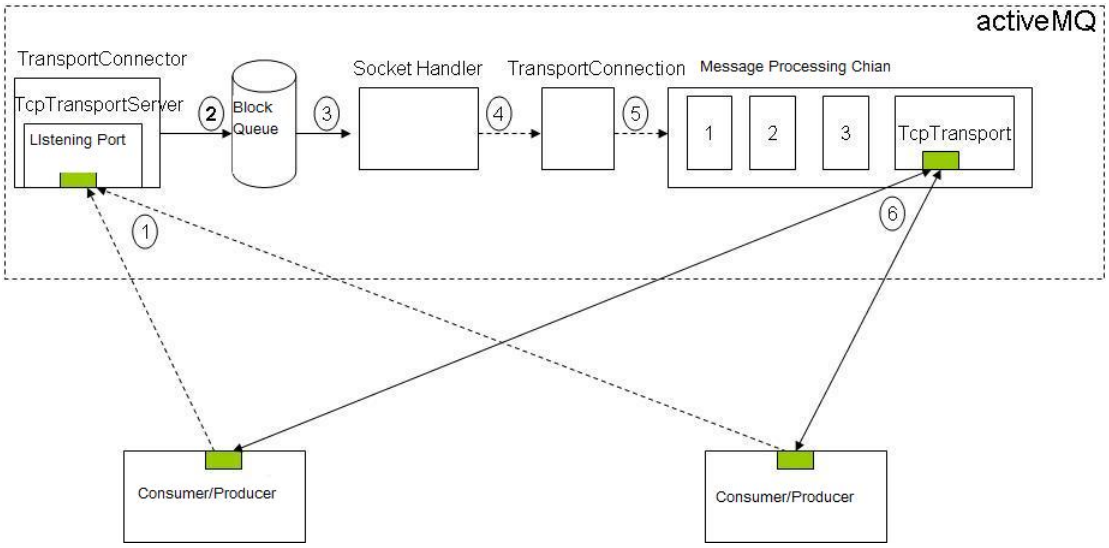

Figure 2.4: Tcp Transport Mode of ActiveMQ

between clients. The communication processes involved in TCP transport always use *Socket* to identify each other. A *Socket* is the handle of the communication channel and composed by the IP address and port number. The TCP transport mechanism of ActiveMQ can be expressed by Figure 2.4.

When ActiveMQ launched, the broker will use *TcpTransportServer* to open the port to listen to the clients' request, the detected request is then queued and handled by the Socket Handler thread. The Socket Handler thread will use *TransportConnection* to process the message. *TransportConnection* implements *CommandVisitor* interface to identify and handle the message.

2.2.2 Openwire protocol

ActiveMQ supports three wire formats: Openwire, STOMP and HTTP/REST. Openwire is the protocol used inside ActiveMQ, it can be used to develop high performance clients in *Java, C, C++, C#*. This paper will introduce an approach of constructing a C client for Iphone application using Openwire protocol.

Openwire protocol is the protocol used upon TCP protocol. The object exchanged between the clients is called *Command* Object. In the messaging approach, marshalling or serialization is the process of breaking structured data into segments so that it can be transmitted as data byte stream over network; unmar-

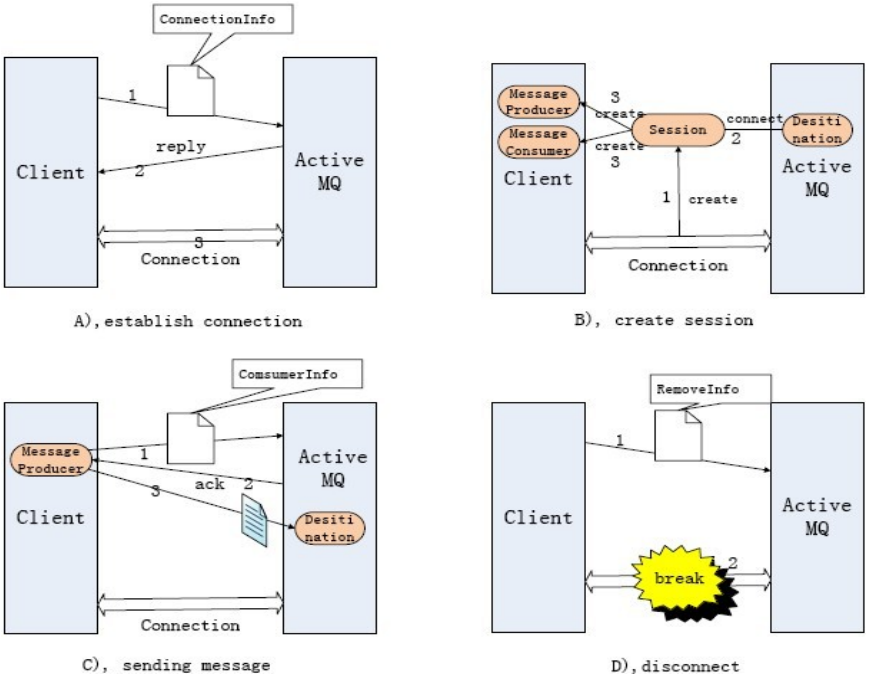

Figure 2.5: The openwire protocol

shalling or deserialization is the process of reconstructing the structured data using assembled streamed bytes arrived at the destination point. Openwire is used to marshalling the *Command* Object into streamed bytes and back.

When the client wants to establish a connection to the broker, it must send a *ConnectionInfo* command with details information of itself such as machine name, host name, user name, password and an unique *clientId*. The client then waits for a valid Response before continuing. The *clientId* is generated using a *unique string generator* provided by the ActiveMQ.

After the connection is established, a session is used to handle the message exchanged between both sides. Session uses logical *MessageProducer* to send messages and *MessageConsumer* to receive messages. Both *MessageProducer* and *MessageConsumer* will be involved in an authentication approach before they can actually send or receive messages. In this authentication approach, *MessageProducer* and *MessageConsumer* will send a *ProduceInfo/ConsumerInfo* command with a unique *produceId/consumerId, sessionId* and *clientId* (connection Id) to the broker. Figure 2.5 shows each scenarios.

To close a resource, client needs to send a *RemoveInfo* command with the

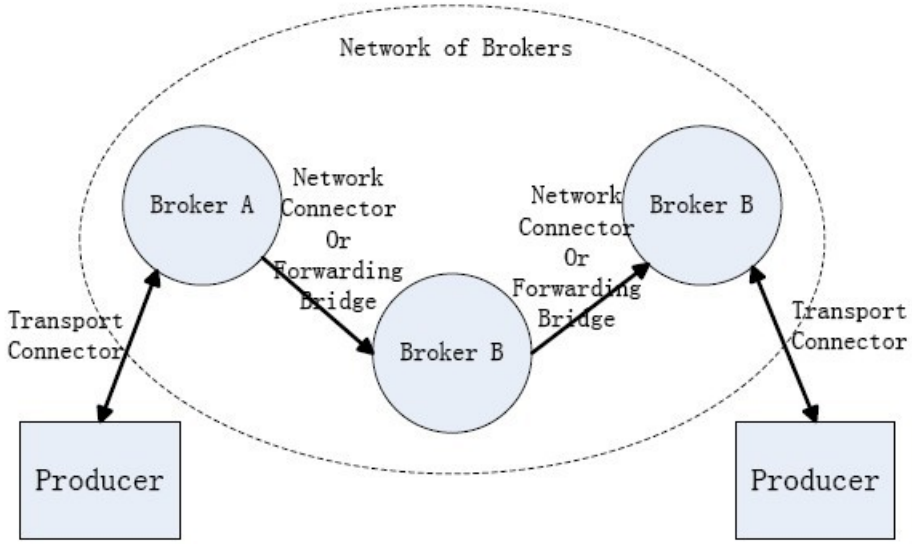

Figure 2.6: Store and forward mechanism[26]

correct *objectId* for the producer, consumer, session, connection etc to the broker.

2.2.3 ActiveMQ cluster

To support hundreds of thousands of football event subscribers, using ActiveMQ cluster is a good solution. With the auto-failover and discovery mechanism, ActiveMQ provides high scalability, reliability and high performance. The topology of the brokers can be bus, star, ring or the hybrid.

The brokers coordinate their action by a store and forward mechanism. In this mechanism, if a broker has producers, but no consumers, it may use one or more forwarding bridges to forward messages on to brokers that have appropriate consumers, if a broker has multiple forwarding bridges, with appropriate consumers at the other ends of the bridges, it will load balance messages across the bridges[26]. As it shows in Figure 2.6, the brokers are connected by *network connector* or *forwarding bridge*, broker A will forward the message to broker C to avoid message piling up without processing.

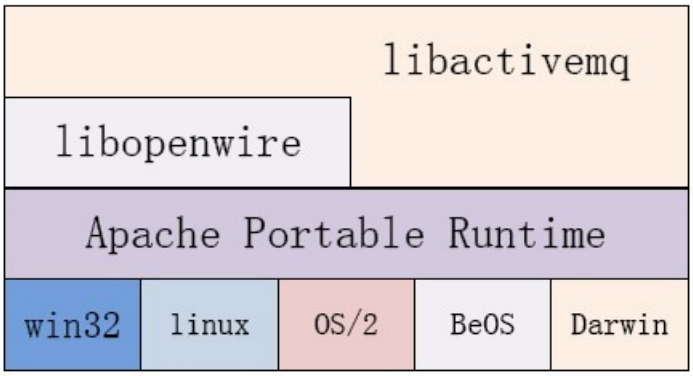

Figure 2.7: libactivemq build stack

2.2.4 C library for Openwire and APR

The C library for Openwire is used for developing Iphone client. This library includes *libactivemq* and *libopenwire* files. *libactivemq* is wrapped around *libopenwire* and provides useful functions like *connect,disconnect,send* and more. The library is based on Apache Portable Runtime (APR), which creates and maintains software libraries that provide a predictable and consistent interface to underlying platform-specific implementations[4]. The relationship of the libraries is shown in Figure 2.7.

2.3 Other Technologies Used

2.3.1 XML

Extensible Markup Language (XML) is a W3C recommended standard of encoding electronic documents. It separates the content and format of data by using tags, which can greatly simplify the processing of document contained information.

XML provides an excellent way to store and transfer information over internet. It's also the format adapt by football match broadcaster (Lyse) to store the events happened during the football match, here the XML file is named by the match and Event number. For example, a free kick happened during the match "Start vs Viking" with the name of "match136633Event4238049.xml" can be expressed like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE match SYSTEM "match.dtd">
<match id="136633" start-time="21.09.2009 19:00" status-id="31"
    status-name="1. omgang" transfer-type="single-event">
        <home goals="1" team="305">Start</home>
        <away goals="0" team="303">Viking</away>
        <details>
                <detail id="4238049" type="26" minute="19" real-time=
                    "21.09.2009 19:18:38" event-name="Frispark" mode="
                    insert">
                        <player id="62882"><![CDATA[Samuelsen]]></
                            player>
                        <team id="303"><![CDATA[Viking]]></team>
                </detail>
        </details>
</match>
```

2.3.2 JDOM

Java Document Object Model (JDOM) is an open source document object model based on java that used for processing XML files. JDOM integrates Document object model (DOM) and simple API for XML (SAX), supports XSLT and XPath.

An code script of processing "match136633Event4238049.xml"using JDOM:

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(new FileInputStream("
    match136633Event4238049.xml"));
Element root = doc.getRootElement();
Element home = root.getChild("home");
System.out.println("Home score is"+home.getContent(0).getValue());
```

2.3.3 JSON

JavaScript Object Natation (JSON) is a light weight data interchange format based on pure text. JSON is more simple and flexible compared with XML. Despite the advantages such as platform and language independent, XML is not easy to generate and process in server side, neither easy to parse in the client side. JSON, on the other hand, has less content compared with XML which means lower transmission cost. It provides very simple API that easier to be processed

by Java. JSON is also supported by Iphone and can be processed by C language. These reasons make JSON the appropriate data structure to store the events.

JSON is composed of two structures: A dictionary like collection of key-value pairs and an ordered list of values. An example of JSON event is:

```
{    "match":"Start VS Viking",
     "result":"1-1",
     "minute":83,
     "player":"Ijeh",
     "team":"Viking",
     "eventname":"Frispark"}
```

In this solution, the event processed by all parts is expressed by JSON string. This paper uses a "Json_simple.jar" package to process JSON strings.

Chapter 3

Detailed Structure

The solution presented here is a Message Queue based Event Notification system (MQEN) which is implemented in Java language. C Event Subscriber (CES) as a C langue client is also introduced in this thesis.

MQEN is composed of three parts (shown in Figure 3.1):

1, The Football Event Publication application (FEP) provides sequence of events to the message broker.

2, The Football Lottery Client application (FLC) provides a user interface for the clients to subscribe their favorite ongoing matches and make prediction about future matches.

3, The Football Lottery Server (FLS) will accept user betting from clients and subscribe match events from the message broker, base on which to make evaluation for each coupon and give out the result.

The ActiveMQ can be running on a single machine or hosted by multiple computers to form a network of brokers. The data source can be live XML stream, database, remote file system or local file system and more. The connection channel between FLC and FLS is TCP but can also be designed by alternative protocols. This solution processes football events using Publish/Subscribe mode and handle result events by PTP mode.

3.1 Football Event Publication (FEP)

A football match can be break up into single events. A event can be a goal, corner, free kick and etc. Each event can be expressed by a XML file. The XML files are named by the match and event number. For example: match136633Event4237920.xml.

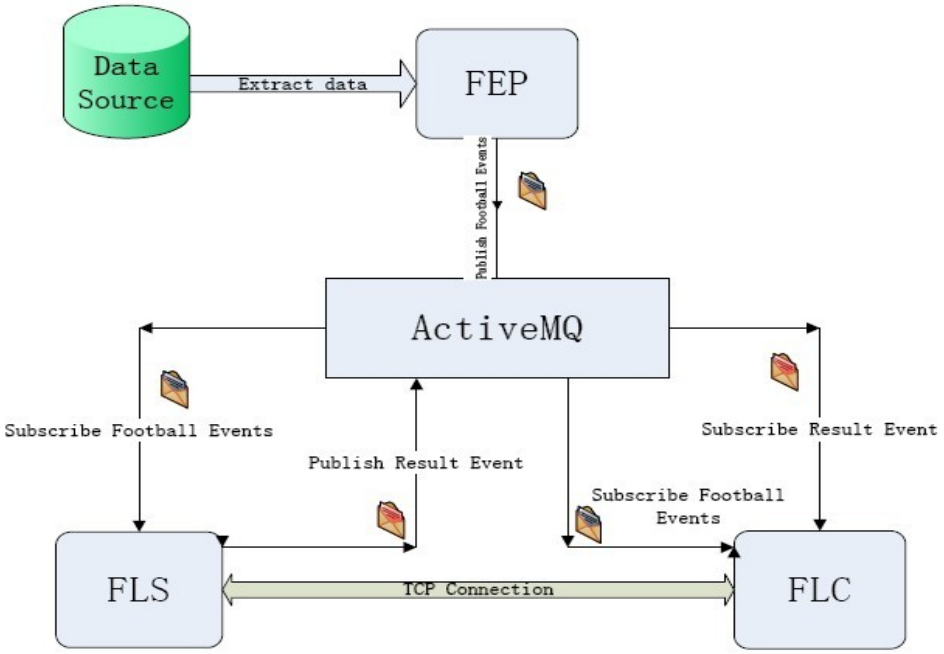

Figure 3.1: Overview Scheme OF MQEN

The XML files should be provided by the television signal provider (Lyse) and accessible for the application through multiple ways. In this solution, The XML files is stored in the hard disk and grouped into different folders by the match name. Thus, this thesis make the assumption that the data source is a file system. The design can be transplant to remote file system if necessary. However, other specific adapters need to be developed for piped XML stream or database.

The system will parse the XML files in the hard disk and generate related events instance, then send them to both the home team and away team topic in the broker. The events use a light weight data format with the name of JSON.

Figure 3.2 shows the structure of FEP. First of all, the *EventPublihser* object will establish the connection to ActiveMQ and read the root path of the XML files directory. It will then create one *MessageSender* object for each XML files within that directory. The *MessageSender* object will use *MatcheParser* object to parse the XML files to JSON strings. At last the JSON string is sent to both the Home team topic and Away team topic of the broker.

When extracting the files from the root directory, a recursive method is adapt, this is shown in Figure 3.3. the function only processes files with the type of

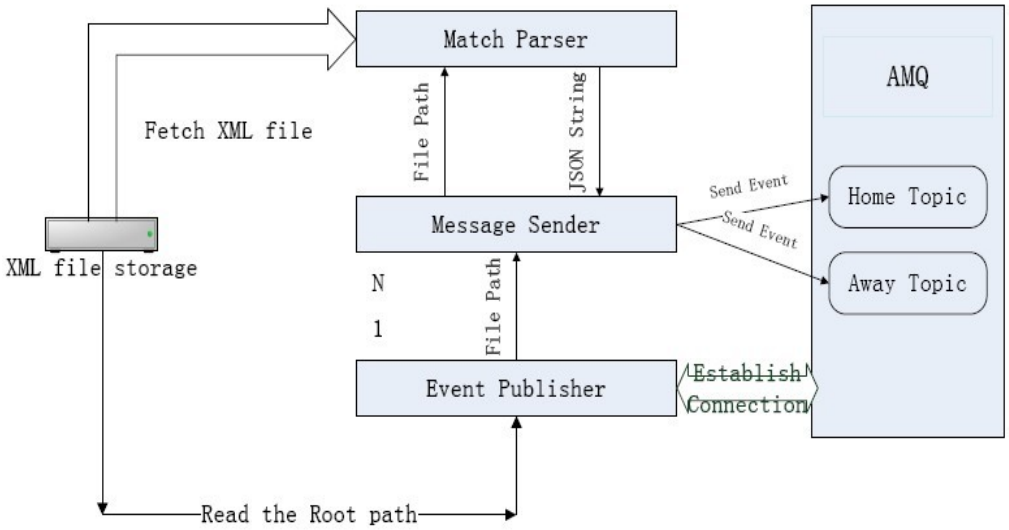

Figure 3.2: Football Event Publication

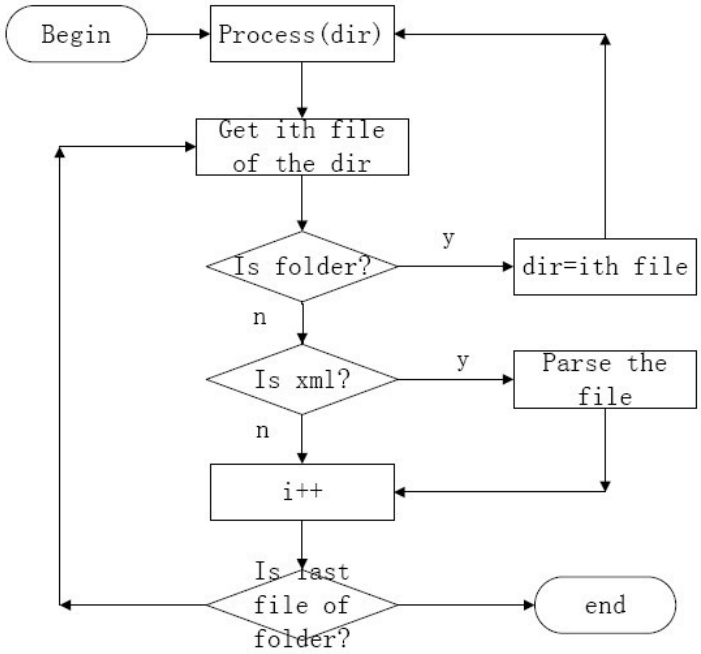

Figure 3.3: Xml file extraction

XML. In the parsing procedure, only important elements of XML is extracted and delivered to JSON event constructer. These elements include:

For matches summery: match name, result, home team and away team.

For single events: match name, result,events name, events committer, committer name, committer team and events time (happening minuter).

3.2 Football Lottery Server (FLS)

The system will accept the user's betting coupons, base on which to calculate the results (number of correct) of each coupon and put the results into the relevant result queues in the broker.

Figure 3.4 showed detailed structure of FLS, The *GoalServer* will provide coupon service at port 52070, when there is request for service, it will open a *ServerThread* object to process the client request. The information provided by the Football Lottery Client (FLC) includes the *prediction form*, a unique *formId* and the *amount* need to be paid. The *ServerThread* will open a *GoalSubscriber* object to subscribe events from the 12 matches predicted by the user. *GoalSubscriber* will use a *ProcessElement* object as a state machine to response the goal event. *ProcessElement* will use a *Coupon* logic to model the user prediction and calculate results. When a *GoalEvent* comes, it will fire on the *ProcessElement* to change the state. After all 12 matches result are received, *ProcessElement* will calculated the final result and notify the *GoalSubscriber*, who will write the result into the relevant Queue in the broker.

3.2.1 Coupon logic

The *Coupon* logic is developed by Mr Hein Meling and proved very efficient. It keeps the match state in a list and user's betting in a immutable list. the match state is changed when goals come. The *Coupon* logic also provides method of evaluating how many matches the user make the right prediction. The primary methods of *Coupon* logic:

```
//inner enumerate class to define 3 outcomes: Win, Draw, Lose
public enum Outcome{};
//inner class to build the coupon
public static final class Builder{};
//method of class Builder
    public Builder set(Outcome o);
```

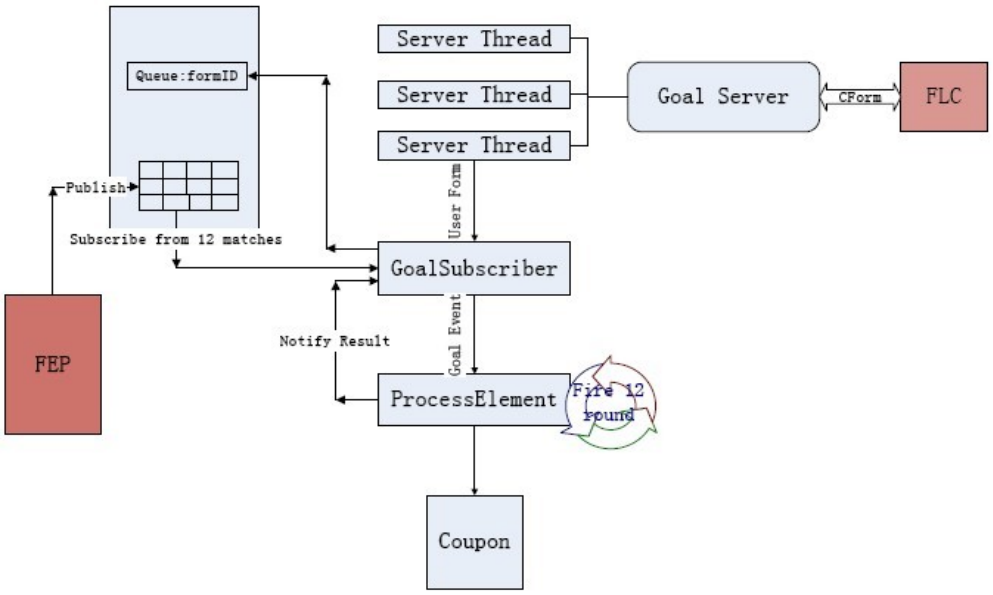

Figure 3.4: Football Lottery Server

```
    public Builder set(Outcome o1, Outcome o2);
    public Builder set(Outcome o1, Outcome o2, Outcome o3);};
    public Coupon build(String owner);
//public methods
public void goal(int matchId, int homeScore, int awayScore);
public int evaluate();
public static String matchState();
public String getPredict();
public String toString();
```

3.2.2 Process element

The *ProcessElement* logic is the state machine that fires on the *GoalEvent*. *GoalSubscriber* objects subscribe football events from ActiveMQ and produce *GoalEvent* for *ProcessElement*, *ProcessElement* use a boolean type array to record which match result is not received yet, thus duplicate message is avoid. After all match result received, the result can be calculated and sent to *GoalSubscriber*. The logic is show in Figure 3.5.

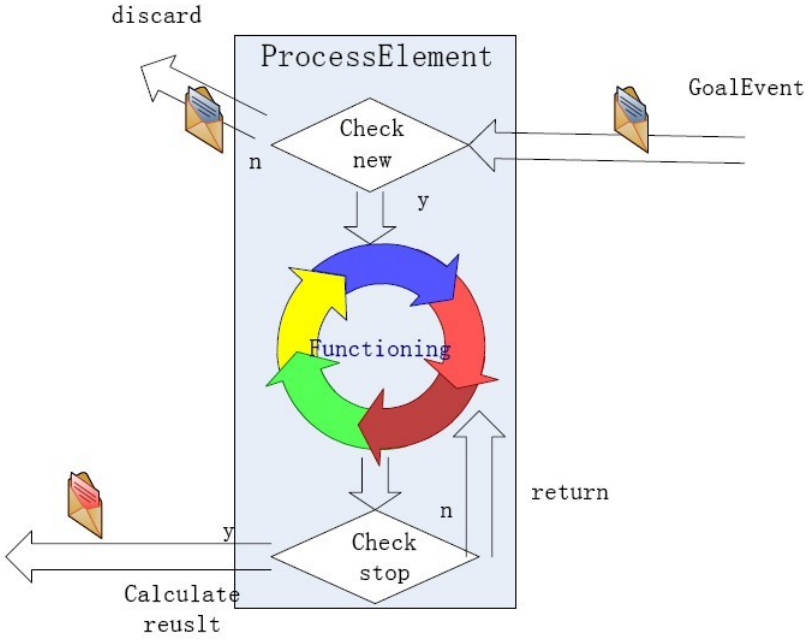

Figure 3.5: The ProcessElement

3.3 Football Lottery Client (FLC)

This application is installed on the user's computer or mobile phone. It provides a graphical user interface for the users. FLC uses TCP as the connection protocol. It will send the user's betting form(*CFrom*), unique *formId* as well as the amount to pay (*price*) to the FLS. FLC gets the betting result and the events of team of interest from the message broker. The *CForm* is a 3×12 array of boolean type, as it shows in Figure 3.7, the 3 columns means *Win,Draw,Lose*, the 12 rows means 12 matches of prediction. At least one element of the same row should be *Ture*, this means user have to make prediction on all of the 12 matches. Assume the price for one prediction is 5 NOK, the total price of the coupon is calculated by the follow formula:

$$y = 5 \times 2^{(x-12)} \tag{3.1}$$

y is the price need to be paid,
x is the number of *Ture* in the *CForm* array. For example, a betting coupon make like it shows in figure 3.7, the price need to pay is $5 \times 2^{15-12} = 40nok$. The *formId* is used as the identifier of result destination, the client will subscribe result from the result queue in the broker.

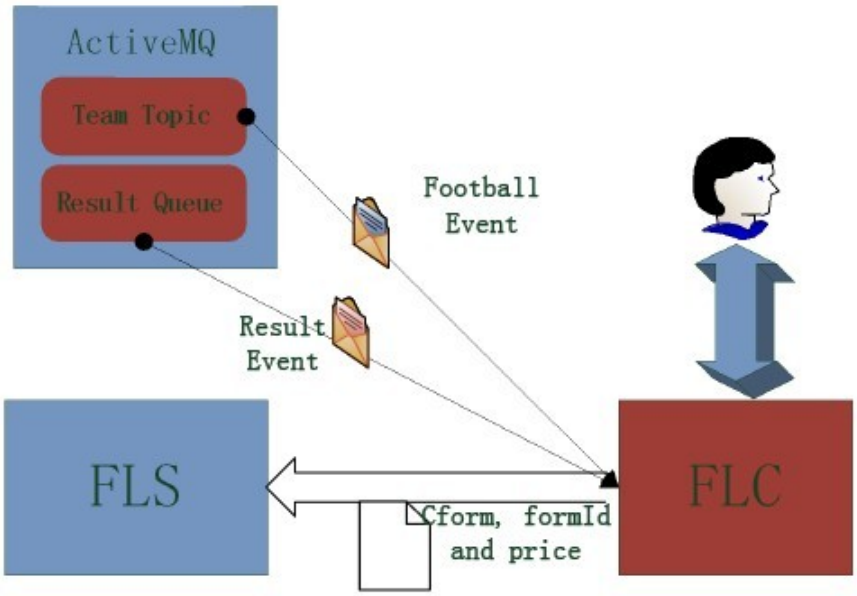

Figure 3.6: Football Lottery Client

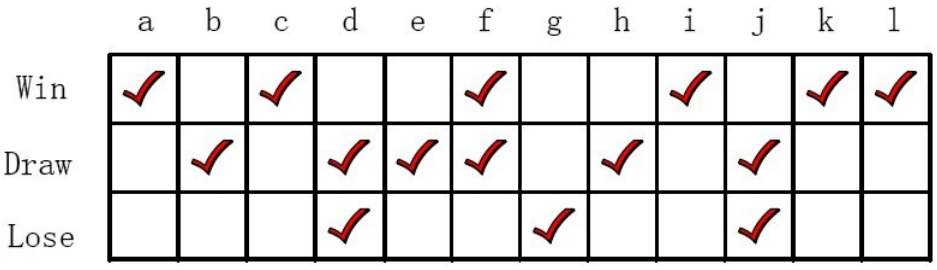

Figure 3.7: The Coupon Form

3.4 C Event Subscriber (CES)

This application is designed using the *libactivemq* and Apache Portable Runtime (APR). APR provides memory pool management and error code enumeration to handle the session. *libactivemq* provides functions and data structure for connecting and sending messages. below is the listing of the primary functions and data structures.

```
//APR
apr_status_t;
apr_pool_t;
//libactivemq
amqcs_connect_options;
apr_status_t amqcs_send(amqcs_connection *connection,
    ow_ActiveMQDestination *dest, ow_ActiveMQMessage *message, ow_int
     deliveryMode, ow_int priority, ow_long timeToLive, apr_pool_t *
    pool);
apr_status_t amqcs_disconnect(amqcs_connection **connection);
apr_status_t amqcs_connect(amqcs_connection **conn,
    amqcs_connect_options *options, apr_pool_t *pool);
```

The design of CES follows these steps:

1. Initialize the APR and allocate the memory pool for the session.
2. Set the connection option and make connection to the ActiveMQ broker.
3. Allocate memory for the destination and the message from the memory pool, initialize the destination.
3. Receive message from the destination.
4. Close the connection.

Chapter 4

Experiments and Evaluation

For the Message queue based event notification system, the performance of both message broker and the football lottery server need to be evaluated. The important measurements should include follow aspects:

Throughput of the message broker, which means number of input events and output events by the broker per second.

Latency of messages, which means the time delay between the sent of a message by message publisher and the receive of the message by the subscriber.

Cpu/memory usage of the servers, include both the message broker and lottery server.

Response time of the lottery server under different lever of load.

message loss of the broker when handling large quantity of events.

4.1 Experiment Setup

The experiment uses computers in the Linux laboratory at the University of Stavanger. All computers in the test approach are running in the same local network, thus the network latency is minimal and negligible, the computers also use network time protocol (NTP) to synchronize their clocks, so they can be regard as time synchronous in the experiment. The configuration of the experiment is shown in Figure 4.1: Computer A runs the lottery server, computer B hosts the ActiveMQ message broker, computer C runs the lottery client and mock client, computer D runs the event publisher and mock message publisher. By putting applications of different functions on different machines, the interference is avoid, thus it will be easier to evaluate each application's performance.

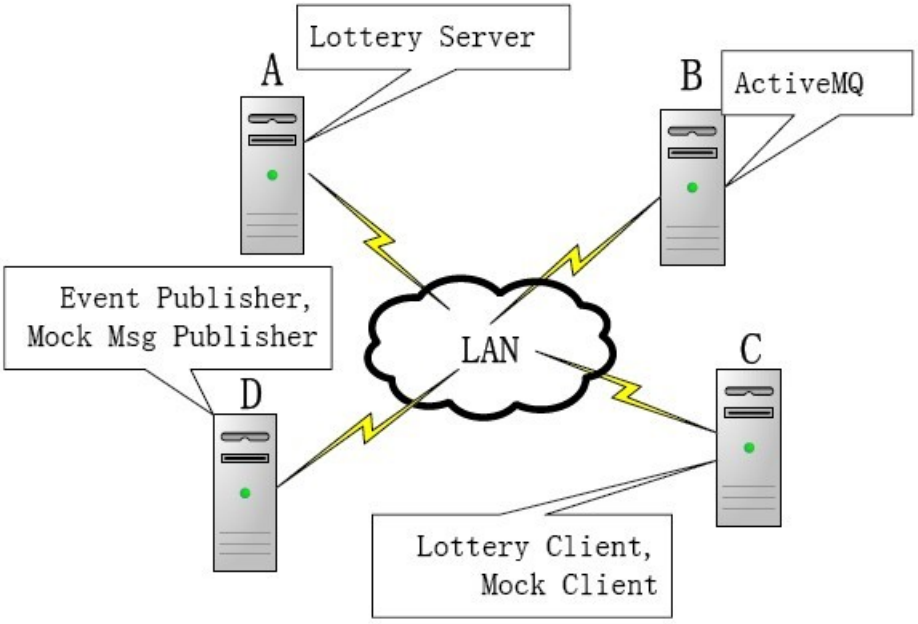

Figure 4.1: Test configuration

For testing the latency of events, each message processed is stamped with the current system time by the sending method; For testing the throughput of broker, the execution time is recorded by both the publication and subscription application; For testing the hardware usage of broker and server, a independent thread which displays the cpu and memory usage periodically is used at machine A and machine B; For testing the server react time, a *MockClient* application is developed to automatically generate client requests; For testing the message loss rate, a *MockMsgGenerator* application is developed to generate large number of mock messages.

4.2 Experiment Results

The test of opening 3 real lottery clients, sending coupons and subscribe is shown in Figure 4.2, the user interface will show a panel to clients and receive user's betting in the check boxes, after pressing send and subscribe button, the result will be returned when it's calculated.

The follow pages shows the statistical figures of message handling speed (*Through-*

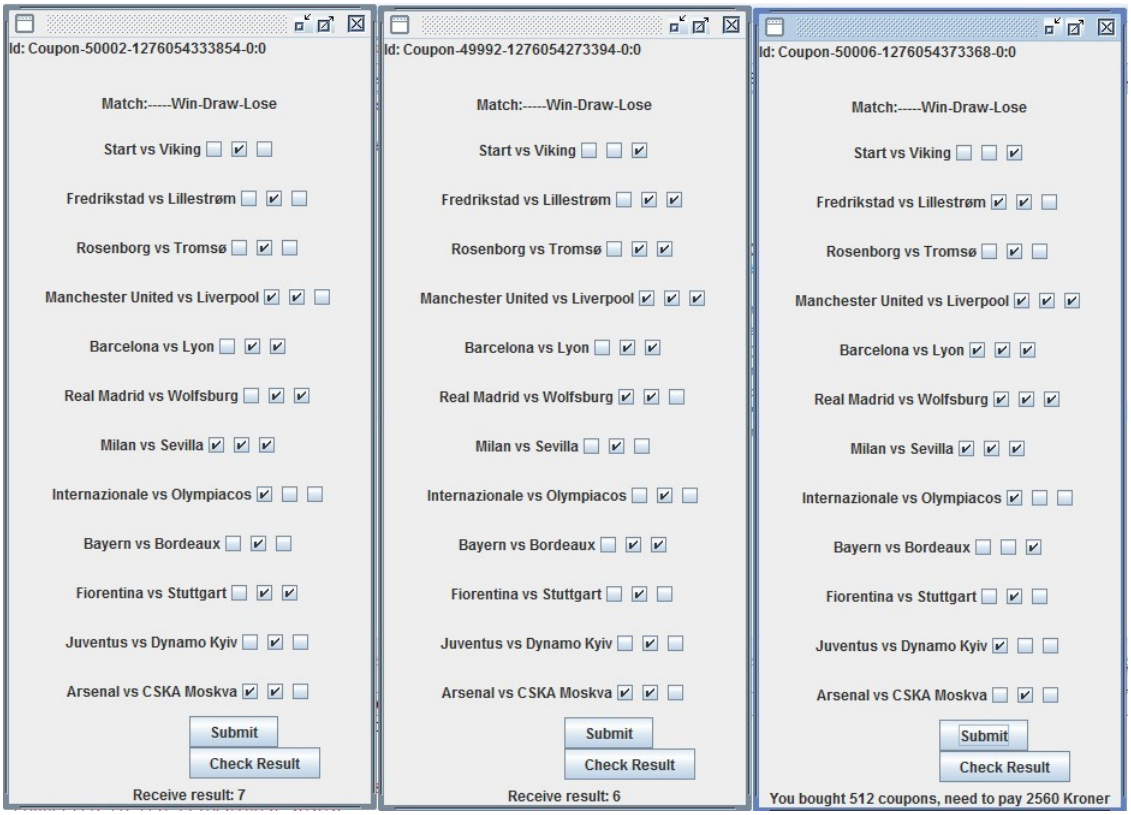

Figure 4.2: Lottery client interface

put), message *latency*, server *response time* and the *cpu/memory usage* of the system. During the test of throughput and latency, different numbers of subscribers (1, 2, 3, 6) are used; During the test of server response time, different number of mock lottery clients (10,20,50,100,200,300) are connected to the server to send coupon forms and subscribe for the result, when trying to open more than 300 clients, the server will crash as failed to create more threads.

The message loss is also test by publishing and subscribing large number of messages, when publishing 10000 messages, no message is lost; When publishing 100000 messages, also no message is lost.

The monitor thread will observe the cpu and memory usage of the message broker and lottery server for 60 seconds, the execution time of lottery server and message broker for handling 200 clients last for approximately 40 seconds, from 10th to 50th second.

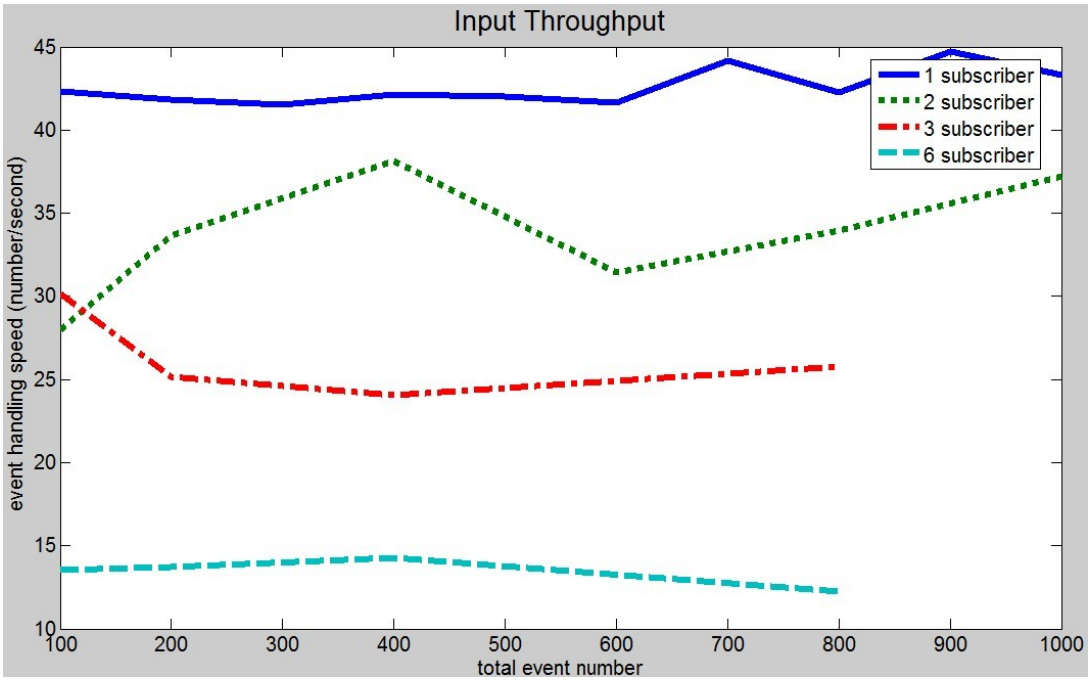

Figure 4.3: Message broker input event handling speed

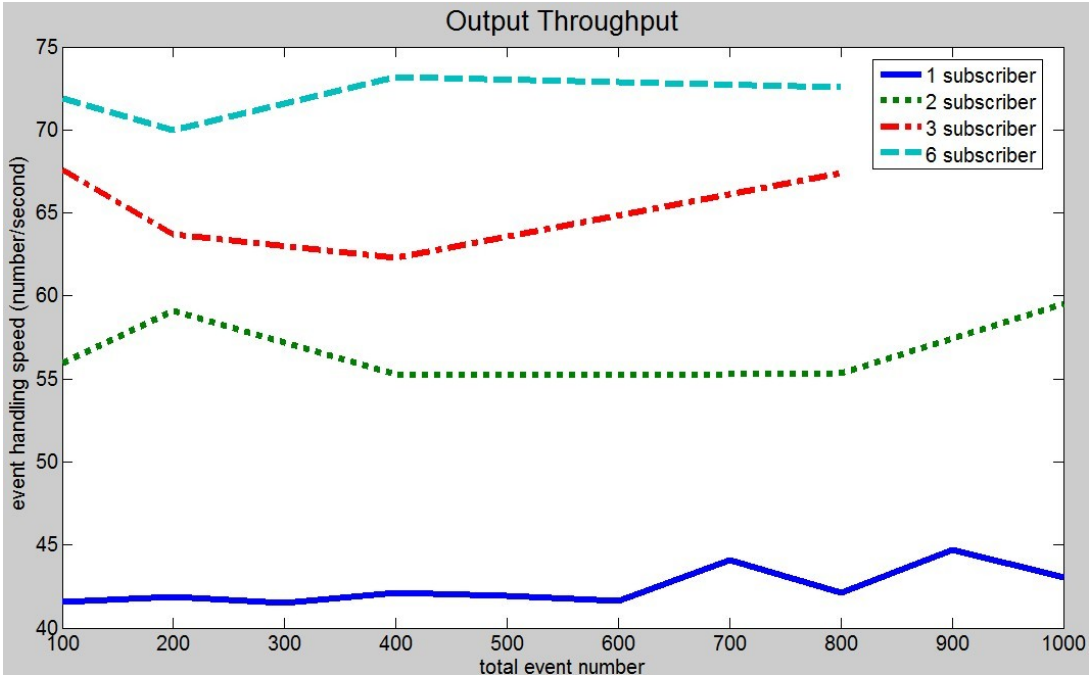

Figure 4.4: Message broker output event handling speed

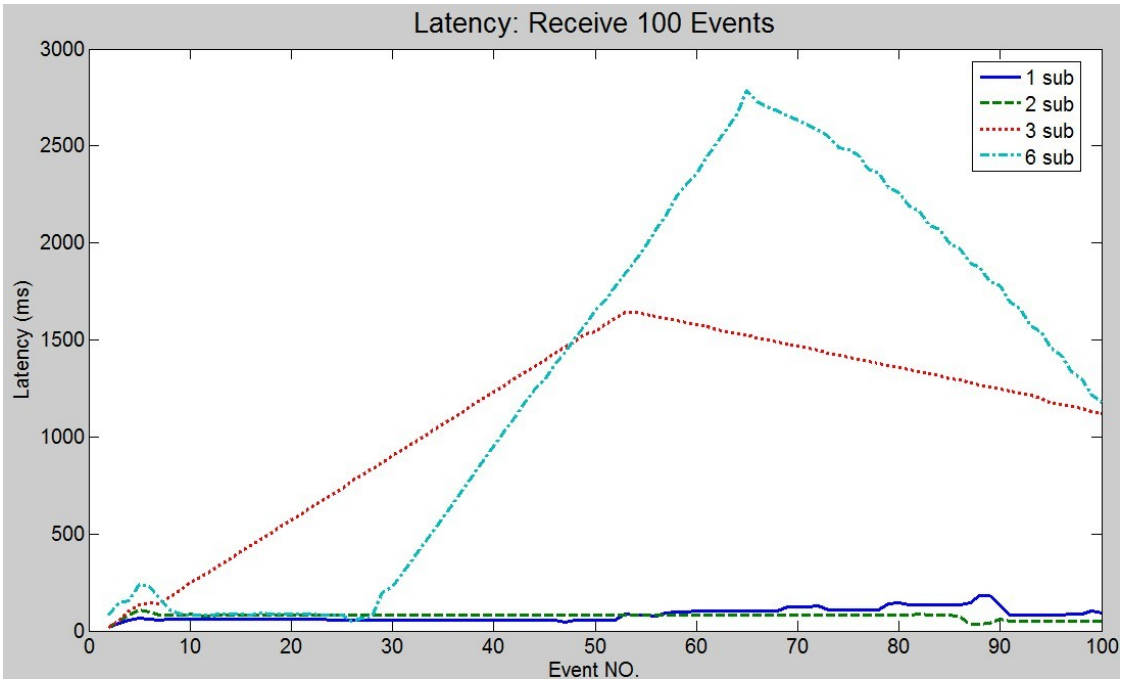

Figure 4.5: Latency of receiving 100 messages

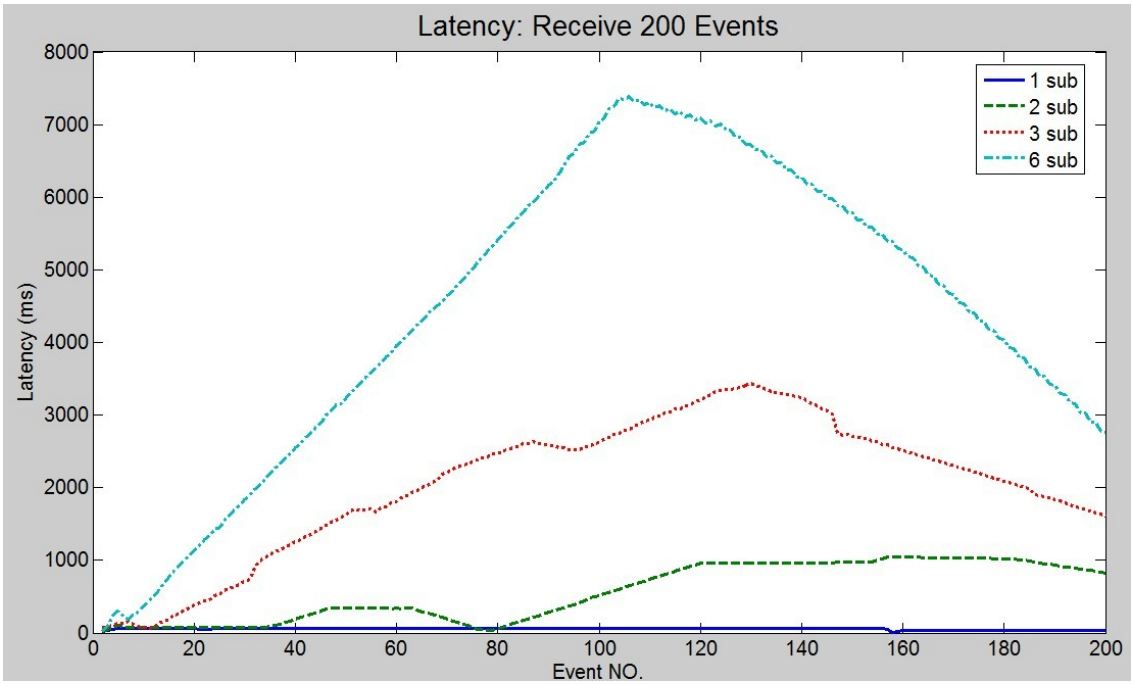

Figure 4.6: Latency of receiving 200 messages

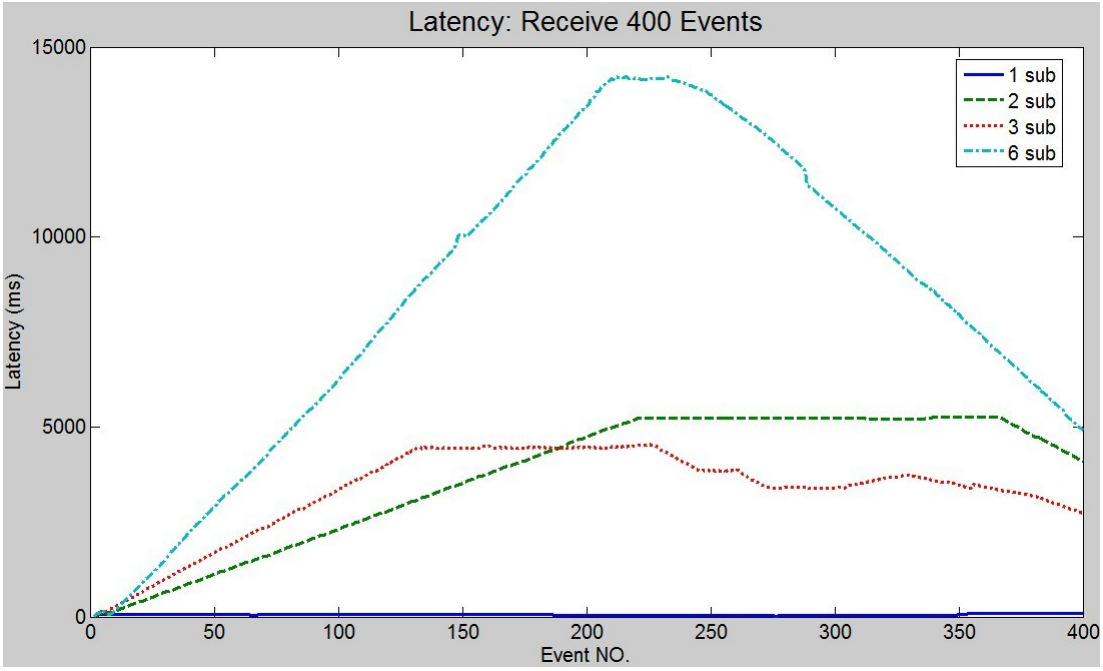

Figure 4.7: Latency of receiving 400 messages

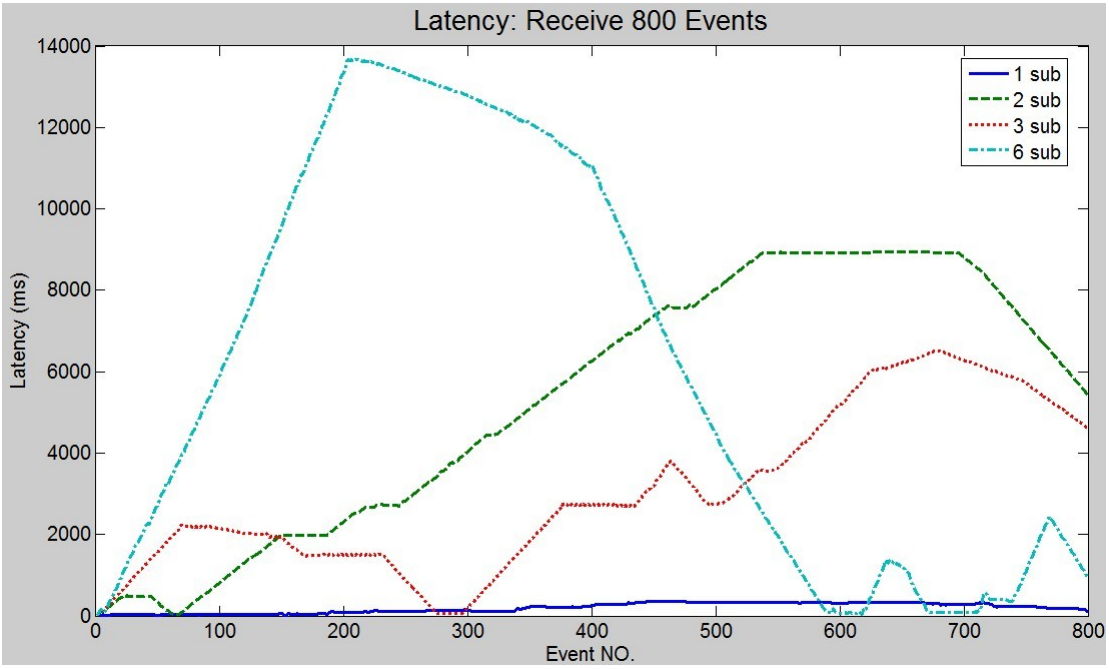

Figure 4.8: Latency of receiving 800 messages

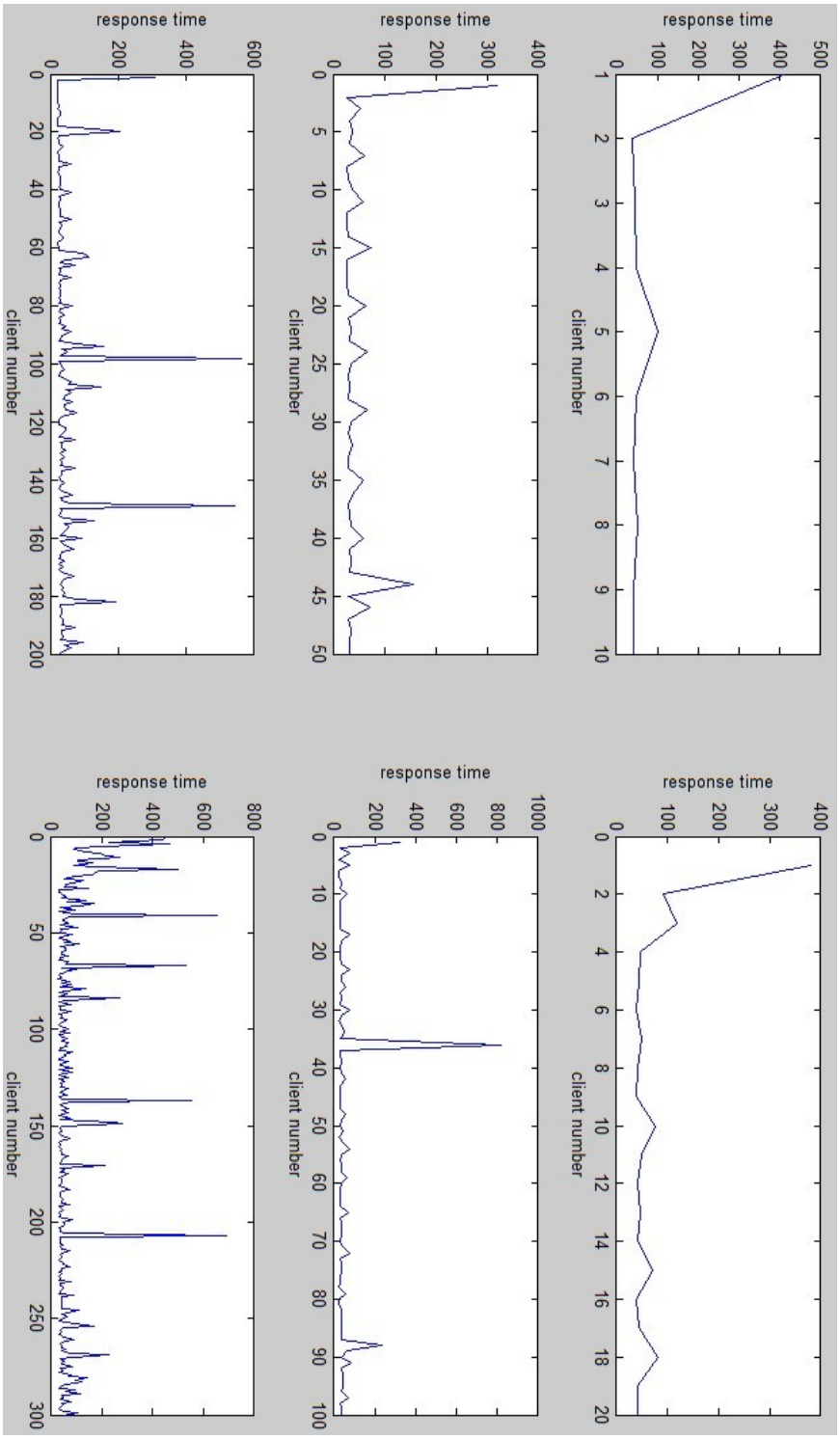

Figure 4.9: Response time of lottery server for different number of clients

```
Send Message: 9997 To topic: mock_topic
Tread: ActiveMQ Session Task Hear Message: uhocqjh2nj6ytn9yy2cn
Number: 9998---Lantency: 78
Send Message: 9998 To topic: mock_topic
Tread: ActiveMQ Session Task Hear Message: z3und4p8t2amm3o9kao6
Number: 9999---Lantency: 72
Send Message: 9999 To topic: mock_topic
totall message send : 10000
 Time consumed: 627764ms
Tread: ActiveMQ Session Task Hear Message: 4idel5ol4yw800zc2lrn
Number: 10000---Lantency: 78
Hear 10000 messages, time consumed 627754ms
```

Figure 4.10: Execution result of handling 10 000 events

```
Send Message: 99997 To topic: mock_topic
Tread: ActiveMQ Session Task Hear Message: ewjovb67xld7oziwd9o4
Number: 99996---Lantency: 157
Send Message: 99998 To topic: mock_topic
Tread: ActiveMQ Session Task Hear Message: drb497jw22m0i2ag0d3h
Number: 99997---Lantency: 156
Send Message: 99999 To topic: mock_topic
totall message send : 100000
 Time consumed: 6015411ms
Tread: ActiveMQ Session Task Hear Message: w1fd72uj3z69u6lcpwju
Number: 99998---Lantency: 221
Tread: ActiveMQ Session Task Hear Message: 6of0w5rdx1zn5hooecgy
Number: 99999---Lantency: 225
Tread: ActiveMQ Session Task Hear Message: u8av5qsvc8d2mdijfa5b
Number: 100000---Lantency: 202
Hear 100000 messages, time consumed 6015506ms
```

Figure 4.11: Execution result of handling 10 000 events

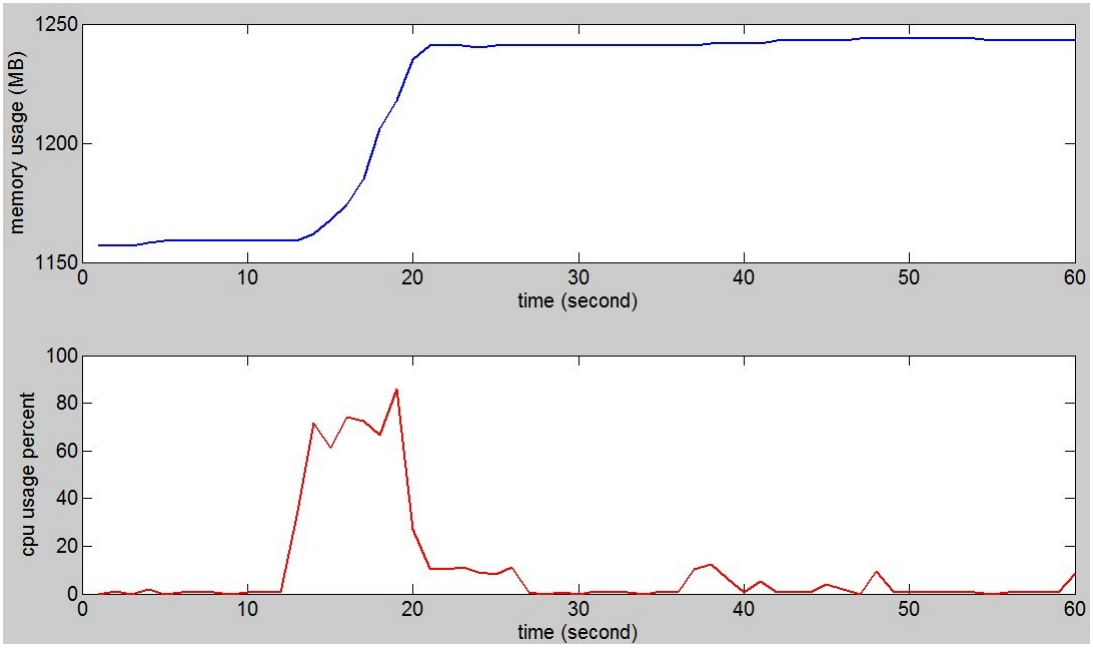

Figure 4.12: CPU and memory usage of the ActiveMQ

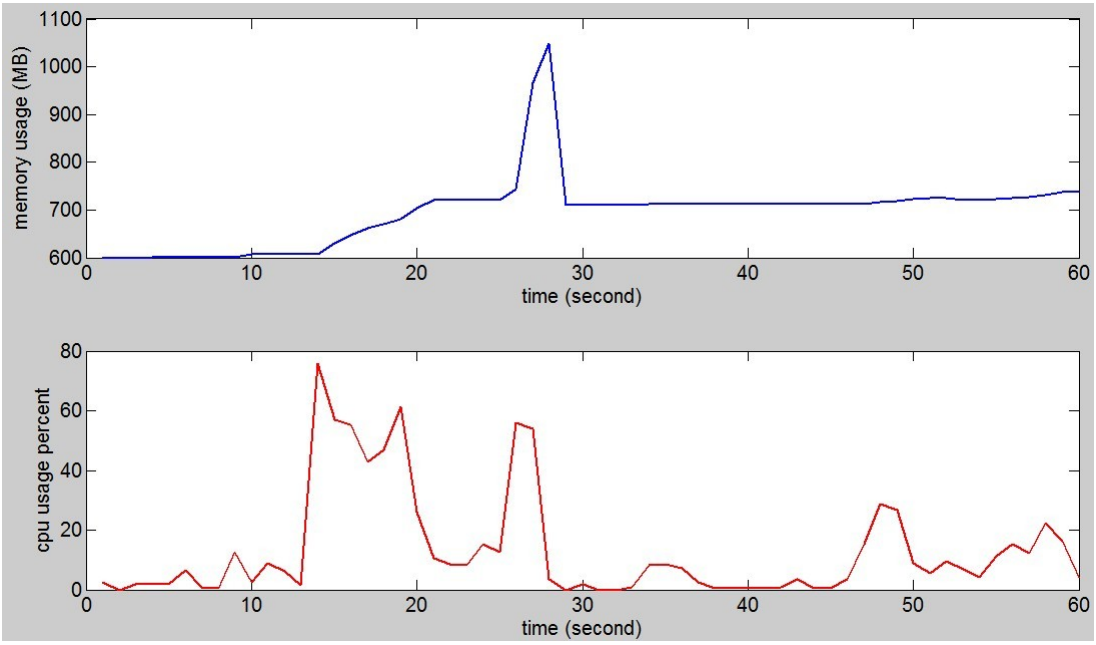

Figure 4.13: CPU and memory usage of the lottery server

4.3 Comments

The performances of lottery server and lottery client are satisfactory, the result calculated for the 3 real clients are right. The server do supports multi threads. But the client will need to keep a connection with the broker all the time before it gets the result. This feature is not fit for the smart phone applications, an improvement approach is proposed at chapter 5.3.

As it shows in the result, the input handling speed of broker is faster when there are less subscribers, the total output handling speed is faster when there are more subscribers. The output handling speed for each single subscriber is close to the input handling speed. The total handling speed of the broker is approximately const which is about 90 messages per second for the broker.

The latency of messages increases rapidly as the number of subscribers increases, scale from dozens of milliseconds to several seconds. During the starting and ending stage of subscribing, the messages have smaller latency. These indicates a single broker is not suitable for providing service to large number of subscribers.

The lottery server's response time for each clients is roughly fair except that the response for first client is always slower. As the number of clients increases, the lottery server became unstable. This means it's not wise to put more than 200 clients on the server. The result indicates that the current design of lottery server won't satisfy the commercial need. Some improvement approach is introduced at chapter 5.1 and 5.4.

When using auto acknowledge mode in the subscriber, ActiveMQ don't drop any messages in the test, even when the total message number reaches 100 000. The message loss test shows that ActiveMQ is quite stable and is more than enough to fulfill enterprise level applications needs.

The cpu and memory test shows that the message broker applies about 80MB memory for handling the events, the memory usage doesn't drop immediately after the finish of execution. The cpu usage increases rapidly at the starting stage of the execution, but drops to normal lever afterwards.

The memory usage of lottery server increases when the execution starts, the peak value at 28th second is due to the starting of receiving events from message broker, the memory usage also does't drop immediately after the end of execution. The cpu usage of lottery server is also very high during the starting of execution, it reaches another high lever at 26th and 27th second due to subscription for football events, at the ending stage, the cpu usage is higher because of sending the result.

Chapter 5

Future Work

5.1 Distributed Football Lottery Server

The Football Lottery Server (FLS) now runs on one machine and incapable of supporting too many clients. A distributed server cluster can be developed to meet the commercial needs.

One basic thought is about using the Spread Toolkit and the Java Server group of Spread (JaSoS) for the solution. Spread is an open source toolkit that provides a high performance messaging service which is resilient to faults across local and wide area networks[6]. JaSoS integrates object group technology and distributed objects using Spread Toolkit[17], it provides *group method invocations* to simplify the designing of server group. Client can interact with the server group using *external group method invocation* (EGMI), servers within the group coordinate their action with *internal group method invocation* (IGMI). JaSoS also provides *group membership service* to make the server group failure awareness.

Anther idea is about using ActiveMQ as the middleware to cache the client requests. This structure is simple but needs a server to monitor the server group. Human intervene is required in case of failure. As it shows in Figure 5.1. The requests from clients are stored in the *Lottery Service Queue*, and consumed by the servers of the group. All servers will send one heart beat message to the *Heart Beat Queue* every 20 seconds, the monitor server will detect failure by subscribing events from the *Hear Beat Queue*.

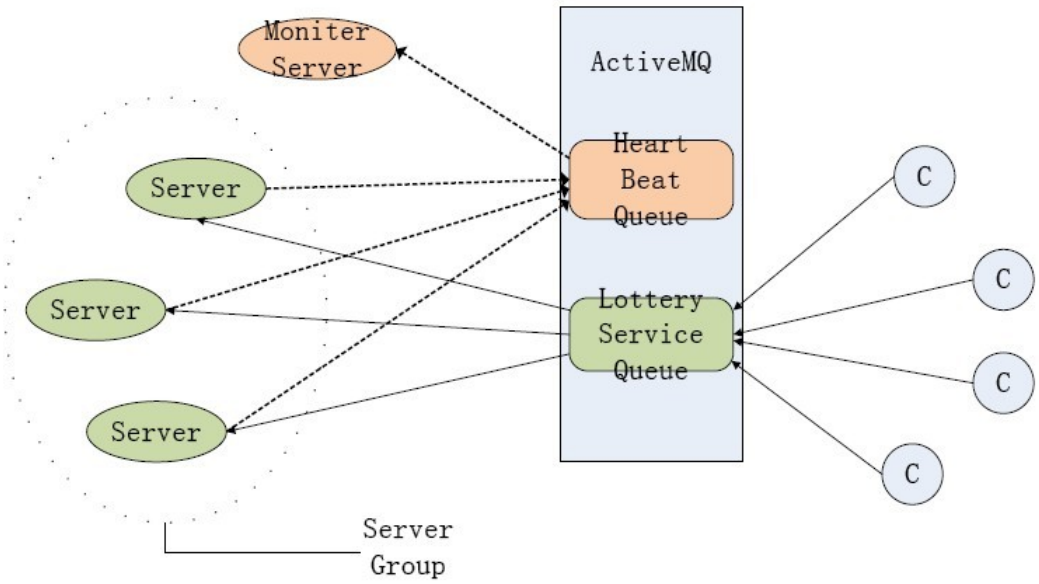

Figure 5.1: Distribute solution using ActiveMQ

5.2 Memory pool management on C Event Subscriber

The Apache Portable Runtime (APR),which provides memory pool management for the C Event Subscriber (CES), is a quite big library. APR library includes a lot of functions that is redundant to CES. Both the storage and memory of smart phone devices are limited, this brings both simple and efficient requirement for smart phone applications. Thus, it's necessary to adapt a light weight memory management library rather than APR for CES. The *Memory Pool C Library*[2] is a good option, it manages multiple heaps that can be allocated and destroyed without fragmenting memory. User can have multiple heaps and reset them easily to completely reclaiming the memory. Below lists the important functions provided by the library[2].

```
mpool_t *mpool_open(const unsigned int flags, const unsigned int
    page_size, void *start_addr, int *error_p);
   // Open/allocate a new memory pool.
int mpool_close(mpool_t *mp_p);
  //  Close/free a memory allocation pool.
int mpool_clear(mpool_t *mp_p);
```

```
    //Wipe a memory pool clean so we can start again.
void *mpool_alloc(mpool_t *mp_p, const unsigned long byte_size, int *
    error_p);
   // Allocate space for bytes inside of an already open memory pool.
void *mpool_calloc(mpool_t *mp_p, const unsigned long ele_n, const
    unsigned long ele_size, int *error_p);
   // Allocate space for elements of bytes in the memory pool and
       zero the space afterwards.
int mpool_free(mpool_t *mp_p, void *addr, const unsigned long size);
  /* Free an address from a memory pool. This is different from
      normal free because it needs the addresses size. Future
      versions of the library will not have that restriction.*/
void *mpool_resize(mpool_t *mp_p, void *old_addr, const unsigned long
     old_byte_size, const unsigned long new_byte_size, int *error_p);
   /* Reallocate an address in a memory pool to a new size. This is
       different from realloc in that it needs the old  address' size.
        If you don't have it then you need to allocate new space, copy
        the data, and free the old pointer yourself.*/
int mpool_stats(const mpool_t *mp_p, unsigned int *page_size_p,
    unsigned long *num_alloced_p, unsigned long *user_alloced_p,
    unsigned long *max_alloced_p, unsigned long *tot_alloced_p);
   // Return stats from the memory pool.
int mpool_set_log_func(mpool_t *mp_p, mpool_log_func_t log_func);
   // Set the Return stats from the memory pool.
int mpool_set_max_pages(mpool_t *mp_p, const unsigned int max_pages);
   // Set the maximum number of pages that the library will use. Once
       it hits the limit it will return  MPOOL_ERROR_NO_PAGES.*/
const char *mpool_strerror(const int error);
    //Return the corresponding string for the error number.
```

5.3 Automatical Result Notification

The current design requires the Football Lottery Client (FLC) to subscribe the result event from ActiveMQ, this brings inconvenience to the users. Automatical result notification enables the user be notified for the coupon results. The FLC needs to send the *deviceId* which can be cell phone number, International Mobile Equipment Identity (IMEI) or IP address together with *CForm, formId* and *price*. After the result is calculated, the server will send the result to the right client using SMS service or TCP transport.

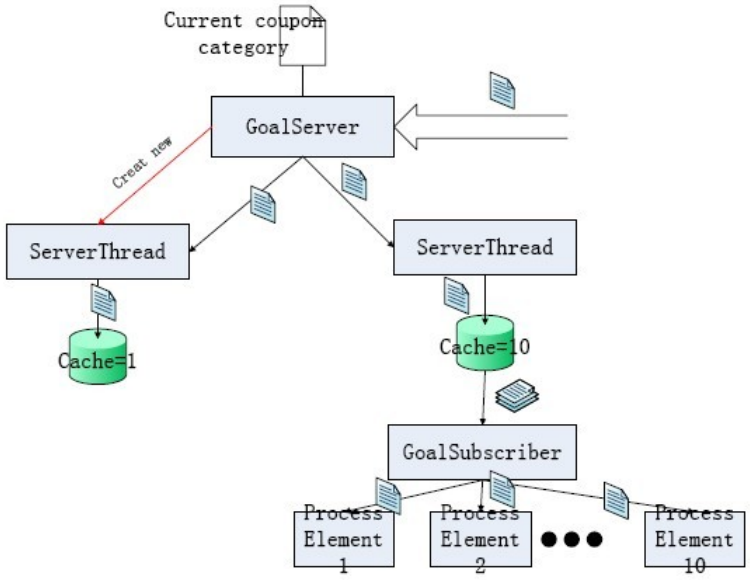

Figure 5.2: High performance server tread

5.4 High Performance Server Thread

In the current design, the FLS opens a *ServerThread* for each coupon from the client, the *ServerThread* will use *GoalSubscriber* to subscribe events from 12 matches based on the coupon and open a *ProcessElement*. However, many users may make prediction on the same match group (12 matches), these coupons are called homogeneous coupons. It will be not necessary to open multiple *GoalSubscriber* for homogeneous coupons. The idea is using a caching mechanism for processing coupons: The server will open new thread for new kind coupon and cache coupons of old kind, it use a *current coupon category* table to record the coupon kinds in processing. The *ServerThread* will wait until 10 homogenous coupons received before opening a *GoalSubscriber*. The *GoalSubscriber* will open 10 *ProcessElement* for processing. The diagram is shown in Figure 5.2.

Appendix A

Listed Java Classes

This section lists the primary classes of the MQEN system.

JsonEvent is the message published to the broker and consumed by clients.

MatchParser is used to parse the XML file and produce JsonEvent.

MessageSender is used to send JsonEvent to the broker.

EventPublisher parse all the XML files from data source and publish them.

EventSubscriber is used to subscribe events from broker.

GoalServer is used to response the clients' request.

ServerThread is used to process clients' request.

GoalSubscriber is used to subscribe matches for clients' coupons.

GoalEvent is used to express the outcome of a match and be passed to ProcessElement.

ProcessElement is used to fire on GoalEvent and produce result.

Coupon can accept user prediction and calculate the result.

Cform stores user prediction and can be passed to the server through internet by implementing *serializable* interface.

CouponForm is used to transform the Cform to provide immutable list to initialize the Coupon.

ClientPanel is used to provider user interface and collect user prediction, it also send the prediction as Cform and calculate the price for the coupon.

MockEventGenerator is used to generate mock GoalEvent for testing.

TcpConnector provides the connection from client to server.

MockClient is used to generate large number of clients for testing.

MockMsgGenerator is used to generate large number of mock messages for testing.

Bibliography

[1] *JMS 1.1 API.* Website. `http://java.sun.com/products/jms/javadoc-102a/index.html`.

[2] *A Memory Pool C Library.* Website. `http://256.com/sources/mpool/`.

[3] *Apache ActiveMQ.* Website. `http://activemq.apache.org`.

[4] *Apache Portable Runtime.* Website. `http://apr.apache.org/`.

[5] *JDOM API.* Website. `http://www.jdom.org/docs/apidocs/`.

[6] *Spread Toolkit.* Website. `http://www.spread.org`.

[7] K. Arabshian and H. Schulzrinne. A generic event notification system using XML and SIP. In *New York Metro Area Networking Workshop 2003*, 2003.

[8] Edward Curry. *Middleware for Communications*, chapter 1,Message-Oriented Middleware. Ireland.

[9] Opher Etzion and Peter Niblett. *Event Processing in Action.* Manning Publications, 2009.

[10] P.T. Eugster, P.A. Felber, R. Guerraoui, and A.M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):131, 2003.

[11] Pål Evenson. *Applying Message-Oriented Middleware and Autonomic Computing Principles in Heterogeneous Sensor Environments.* University of Stavanger, May,2010.

[12] Pål Evenson and Hein Meling. *SenseWrap: A Service Oriented Middleware with Sensor Virtualization and Self-Configuration.* University of Stavanger.

[13] A. Hinze. A-mediAS: an adaptive event notification system. In *Proceedings of the 2nd international workshop on Distributed event-based systems*, page 8. ACM, 2003.

[14] Binjia Jiao, Sang H. Son, and John A. Stankovic. *GEM: Generic Event Service Middleware for Wireless Sensor Networks*. University of Virginia.

[15] Martin Kuehnhausen and Victor S. Frost. *Application of the Java Message Service in Mobile Monitoring Environments*. Technical report, The University of Kansas, 2010.

[16] H. Meling, A. Montresor, B.E. Helvik, and O. Babaoglu. Jgroup/ARM: a distributed object group platform with autonomous replication management. *Software-Practice and Experience*, 38(9):885–924, 2008.

[17] Hein Melling and Alberto Montresor. *JaSoS Tutorial*, September 2008.

[18] Gero Mühl, Ludger Fiege, and Peter Pietzuch. *Distributed Event-Based Systems*. Springer, July 2006.

[19] Peter Robert Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, Queens' College, University of Cambridge, February 2004.

[20] Guerraoui Rachid and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, April 2006.

[21] Mark Richards, Richard Monson-Haefel, and David A Chappell. *Java Message Service, Second Edition*. O'Reilly Media, May 2009.

[22] Bruce Snyder, Dejan Bosanac, and Rob Davies. *ActiveMQ in Action*. Manning Publications, November 2009.

[23] Jonathan R. Stanton. *A Users Guide to Spread Version 0.11*. Johns Hopkins University, October 2002.

[24] Sun Microsystems, Inc. *JavaTM Message Service Specification*, April 2002.

[25] Sun Microsystems, Inc. *Sun GlassFish Message Queue 4.4 Technical Overview*, October 2009.

[26] Total Transaction Management, San Marcos,USA. *ActiveMQ 5.2 Reference Guide v1.8*, 2008.