

# 國立交通大學

電機學院與資訊學院 資訊學程

## 碩士論文

FJM2 - 分散式 Java 訊息服務系統



FJM2 - A Decentralized JMS System

研究生：蘇國榮

指導教授：袁賢銘 教授

中華民國九十五年六月

FJM2 - 分散式 Java 訊息服務系統

FJM2 - A Decentralized JMS System

研究生：蘇國榮

Student：Kuo-Jung Su

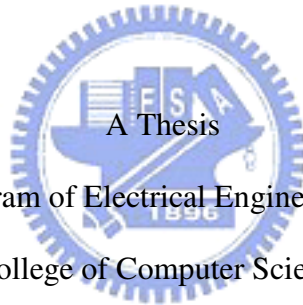
指導教授：袁賢銘

Advisor：Shyan-Ming Yuan

國立交通大學

電機學院與資訊學院專班 資訊學程

碩士論文



Submitted to Degree Program of Electrical Engineering and Computer Science

College of Computer Science

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

June 2006

Hsinchu, Taiwan, Republic of China

中華民國九十五年六月

# FJM2 -分散式 Java 訊息服務系統

學生：蘇國榮

指導教授：袁賢銘

國立交通大學 電機學院與資訊學院 資訊學程（研究所）碩士班

## 摘 要

隨著網際網路的日益發達，程式之間彼此透過網路來溝通及交換訊息的需求也漸趨頻繁。而像 Java 訊息服務系統(JMS)這類以訊息導向之中介層程式不但可以讓網路程式開發的技術門檻大幅降低，同時還有快速發展具可靠性、安全性，事件驅動等特性。傳統的用戶端/伺服器模式需要用戶端程式自行管理及維護建立連線的系統資源；同時伺服器端也必須將相同的資料複製到與使用者數量相等的份數來傳遞，不但浪費系統資源也浪費了網路資源。

本論文研製之 Fast Java Messaging 2 (FJM2) 系統為 91 年度洪傳寶學長所設計 Fast Java Messaging (FJM)系統的改良版本。FJM2 建構了一種嶄新的 JMS 系統，具備了分散、高速、可靠而且比其他 JMS 產品更容易配置與使用！FJM 2 因為使用了更有效率的 Negative-acknowledgment (NACK)-Oriented Reliable Multicast (NORM) 通訊協定，而不再採用 Topic Address Binding 的技術，所以 FJM2 的 Publisher 與 Subscriber 可執行在同一台電腦，這是 FJM 無法做到的模式；另外 FJM2 還提供了穿越 Internet 溝通的能力，大幅擴大了系統的覆蓋範圍，相對地也為擴展了可能的應用範圍。本研究的經驗，可供想利用 Java 和 Multicast 通訊協定來開發網路通訊中介軟體的讀者作為參考。

# FJM2 - A Decentralized JMS System

Student : Kuo-Jung Su

Advisor : Dr. Shyan-Ming Yuan

Degree Program of Electrical Engineering and Computer Science  
National Chiao Tung University

## Abstract

With the growth of internet, the requirement for the communication and message exchanges between programs becomes more and more popular. And the Message-Oriented Middleware (MOM), such as Java Message Service (JMS), could not only greatly reduce the technical doorsill for programmers but also has some amazing characteristics: such as reliable, secured, and event-driven. In the traditional Client / Server model, not only the client side program has to maintain the resource for connections and memory management, but also the server side has to send out several copies of duplicated messages per amount of connected clients, it not only waste the system resource but also the network environment.

A system this thesis develops is Fast Java Messaging 2 (FJM2), it's a resigned of Fast Java Messaging (FJM) which designed by Chuan-Pao Hung at 2002. FJM2 creates a whole new JMS provider which is distributed, high performance, reliable, and easy to use and deploy! While compared with FJM, FJM2 adapts a more efficient communication protocol - Negative-acknowledgment (NACK)-Oriented Reliable Multicast (NORM), and no longer adapts Topic Addressing, and thus FJM2 could have Publisher and Subscriber running at the same machine, while FJM could not, and moreover FJM2 has the ability to work across WAN environment, and thus it extends the system coverage, and could be adapted for more different application scopes. We hope that our experience would benefit those who want to create a MOM system based on Java and multicast protocol.

---

## ACKNOWLEDGEMENTS

---

首先感謝我的指導教授 袁賢銘教授 教導分散式系統上的基礎以及給予我論文重要的意見。在分散式系統實驗的博士班學長蕭存喻、鄭明俊、葉秉哲、高子漢，感謝他們在自己研究之餘能為我的論文給予不少的意見，也感謝已離開學校的洪傳寶學長，願意在繁忙的工作壓力下抽空給我一些寶貴的意見。最後感謝我的父母 蘇鳳霖 和 蘇郭珍珠，謝謝你們讓我能無後顧之憂並讓我自由地選擇資訊領域為我終生的志向。



---

# CONTENTS

---

<b>ABSTRACT IN CHINESE.....</b>	<b>I</b>
<b>ABSTRACT IN ENGLISH .....</b>	<b>II</b>
<b>ACKNOWLEDGEMENTS.....</b>	<b>III</b>
<b>CONTENTS.....</b>	<b>IV</b>
<b>LIST OF FIGURES .....</b>	<b>VII</b>
<b>CHAPTER 1 INTRODUCTION .....</b>	<b>1</b>
1.1 PREFACE.....	1
1.2 MOTIVATION.....	3
1.3 OVERVIEW.....	4
<b>CHAPTER 2 BACKGROUND &amp; RELATED WORK .....</b>	<b>5</b>
2.1 MESSAGE ORIENTED MIDDLEWARE (MOM).....	5
2.2 JAVA MESSAGE SERVICE (JMS) .....	6
2.3 IP MULTICAST .....	8
2.4 FORWARD ERROR CORRECTION (FEC) .....	10
2.5 NEGATIVE-ACKNOWLEDGMENT ORIENTED RELIABLE MULTICAST (NORM) .....	12
2.3 FAST JAVA MESSAGING (FJM) .....	12
<b>CHAPTER 3 FAST JAVA MESSAGING 2.....</b>	<b>17</b>
3.1 FJM2 SYSTEM ARCHITECTURE.....	18
3.1.1 <i>FJM2 Architecture Overview</i> .....	18
3.1.2 <i>FJM2 Physical Architecture Overview</i> .....	19
3.2 FJM2 MESSAGE TRANSMISSION MODEL.....	20

3.3 FJM2 MESSAGE PUBLISH PROGRAM FLOWS .....	21
3.4 FJM2 MESSAGE SUBSCRIBE/CONSUME PROGRAM FLOWS.....	23
3.5 FJM2 MANAGEMENT PROGRAM FLOWS .....	25
3.6 FJM2D SYSTEM ARCHITECTURE.....	27
3.7 FJM2-ADMIN SYSTEM ARCHITECTURE.....	28
<b>CHAPTER 4 FJM2 PROTOCOL .....</b>	<b>30</b>
4.1 FJM2 PROTOCOL LAYERS .....	30
4.2 NORM PROTOCOL – A BRIEF OVERVIEW .....	31
4.2.1 <i>NORM Delivery Service Model</i> .....	32
4.2.2 <i>NORM Scalability</i> .....	32
4.3 NORM PROTOCOL IN FJM2.....	33
4.3.1 <i>Forward Error Correction Algorithm in FJM2</i> .....	33
4.3.1 <i>NACK Algorithm in FJM2</i> .....	35
<b>CHAPTER 5 FJM2 PERFORMANCE ANALYSES.....</b>	<b>36</b>
5.1 TEST ENVIRONMENT .....	36
5.2 UDP PERFORMANCE BENCHMARK ON LOOP-BACK INTERFACE.....	37
5.3 UDP PERFORMANCE BENCHMARK ON 100Mbps ETHERNET INTERFACE.....	39
5.4 JMS PERFORMANCE BENCHMARK ON LOOP-BACK INTERFACE .....	41
5.5 JMS PERFORMANCE BENCHMARK ON 100Mbps ETHERNET INTERFACE .....	43
5.5.1 <i>One-to-One Benchmark on 100Mbps Ethernet Interface</i> .....	43
5.5.2 <i>One-to-Two Benchmark on 100Mbps Ethernet Interface</i> .....	45
5.5.3 <i>1-to-1 Benchmark on 100Mbps Ethernet Interface across WAN</i> .....	47
5.6 CONCLUSION .....	48
<b>CHAPTER 6 FJM2 IMPLEMENTATION ISSUES AND ANALYSES .....</b>	<b>49</b>

6.1 CHOOSING A RIGHT NETWORK ADAPTER .....	49
6.2 FAMILIAR WITH YOUR SWITCH AND ROUTER .....	49
6.2.1 Turn off or increase the value for Multicast flow control .....	49
6.2.2 The processor speed limits the tunnel performance.....	50
6.3 MEMORY ALLOCATION REDUCTION .....	52
<b>CHAPTER 7 CONCLUSION .....</b>	<b>53</b>
<b>BIBLIOGRAPHY .....</b>	<b>55</b>





---

## LIST OF FIGURES

---

<i>Figure 2-1. JMS Programming Model</i> .....	6
<i>Figure 2-2. JMS PTP Messaging Model</i> .....	7
<i>Figure 2-3. JMS Publisher/Subscriber Messaging Model</i> .....	8
<i>Figure 2-4. Systematic FEC Encoding</i> .....	11
<i>Figure 2-5. Systematic FEC Decoding</i> .....	11
<i>Figure 2-6. FJM Membership Topology</i> .....	14
<i>Figure 2-7. FJM Publisher Registration Flow</i> .....	15
<i>Figure 2-8. FJM Subscriber Registration Flow</i> .....	15
<i>Figure 3-1. FJM2 Architecture Overview</i> .....	18
<i>Figure 3-2 FJM2 Physical Architecture</i> .....	19
<i>Figure 3-3. FJM2 Message Transmission Model</i> .....	20
<i>Figure 3-4. FJM2 Message Publish Program Flows</i> .....	21
<i>Figure 3-5. FJM2 Message Subscribe/Consume Program Flows</i> .....	23
<i>Figure 3-6. FJM2 Management Model and Flows</i> .....	25
<i>Figure 3-7. FJM2D Architecture and Operation Flows</i> .....	27
<i>Figure 3-8. FJM2-Admin Architecture and operation flows</i> .....	28
<i>Figure 4-1. FJM2 Protocol Layers</i> .....	30
<i>Figure 4-2 Compact No-Code FEC ("fec_id" = 0) "fec_payload_id" Format</i> .....	33
<i>Figure 5-1. UDP UNICAST Throughput on Loop-back Interface</i> .....	37
<i>Figure 5-2. UDP MULTICAST Throughput on Loop-back Interface</i> .....	38
<i>Figure 5-3. 1-to1 performance test connection topology</i> .....	39
<i>Figure 5-4. UDP UNICAST Throughput on 100Mbps Ethernet Interface</i> .....	40
<i>Figure 5-5. UDP MULTICAST Throughput on 100Mbps Ethernet Interface</i> .....	40

<i>Figure 5-6. JMS throughput on Loop-back Interface</i> .....	42
<i>Figure 5-7. JMS 1-to1 performance test connection topology</i> .....	43
<i>Figure 5-8. JMS 1-to-1 Throughput on 100Mbps Ethernet</i> .....	44
<i>Figure 5-9. JMS 1-to-2 performance test connection topology</i> .....	45
<i>Figure 5-10. JMS 1-to-2 Throughput on 100Mbps Ethernet</i> .....	46
<i>Figure 5-11. FJM2 1-to-1 WAN performance test connection topology</i> .....	47
<i>Figure 5-12. FJM2 WAN Throughput on 100Mbps Ethernet</i> .....	48
<i>Figure 6-1. Simplified Ethernet Block Diagram</i> .....	50
<i>Figure 6-2. Router with only 1 Ethernet MAC</i> .....	51
<i>Figure 6-3. Router with 2 Ethernet MAC</i> .....	51



---

# CHAPTER 1 INTRODUCTION

---

## 1.1 Preface

The Networking Technology is making a progress at a tremendous step in recent years. In wired environment, the bandwidth today has already been improved to 1Gbps, and in wireless network, the 802.11n [1] proposes some new physical transmission technologies, and has improved the transmission rate up-to 300Mbps. Therefore applications such as Digital Home, Video on Daemon would no longer be a utopian idea, but a reasonable application to the real world.

With the advancements of the physical networking technology, more and more applications are trying to take the advantage of the network to establish a large-scale system, however such system would have a higher technical doorsill, and difficult to debug and thus violate the Time-to-Market issue, so that some kind of Message-Oriented Middleware (MOM) architecture has been proposed, MOM is a technology which could hide the technical complexity from programmers to make it become much easier and quickly for programmer to create a large-scale, reliable, secured application. There are many MOM systems today, such as CORBA [11], DCOM [10], Java Message Service (JMS) [8][9][12]...etc. Today JMS is the most popular and widely deployed among all of them.

In the traditional Client/Server model, we usually adapt UNICAST transmission, because it fits human's instincts much well, and it's also easier for debugging and management. However in UNICAST transmission mode, it could only have one

single destination at a time, so that if we want to broadcast a message to everyone interesting about it, we have to send out several copies of the duplicated messages, and with the number of receiver grows, it would become an obvious system overhead for both software application and network environment itself. In contrast to the UNICAST, there is another technology named MULTICAST, which could have multiple destinations for a message at a time, and thus reduce the overhead for such application. However it's also cause of the multiple destinations capability of MULTICAST, the management and debugging process would be much more complicated than UNICAST and it requires much more software efforts, too. Moreover the routing of MULTICAST protocol is a difficult course to solve, it's not only a technical issue but also a policy issue to Internet Services Provider (ISP), and most of the existing routers and gateways do not support these MULTICAST routing features, so that the MULTICAST application usually constrained into LAN environment only, and thus the possible application would be limited, too.



Speaking of the reliable service, the traditional way to provide message reliability is through positive Acknowledge (ACK), it sends out a positive acknowledge to sender per received message, however in such architecture, the acknowledgement itself is the performance bottleneck, it's the root cause of the relative longer message latency which constrains the overall system performance. In contrast, there is a technology called - Negative Acknowledge (NACK) has been proposed, it sends out only the information about the lost messages to sender, not positive acknowledges to every received messages to reduce both the number of control message and the transmission latency. In a network environment with a lower packet loss rate, the lower the lost rate is, the better performance we could gain through Negative Acknowledge (NACK). Finally let's talk about the message management and

dispatch, the traditional design uses Centralized Architecture, its topology is much like an aster, and before every message delivered from one to another, the message must be sent to the centralized management node for later destination dispatch and management, and thus the centralized node is the bottleneck of the overall performance in such system. And we usually applies a fail-over / fault-tolerance mechanism to provide reliability for such architecture, and in such design, there is nothing we could do for further network traffic optimization, because the centralized architecture usually uses UNICAST communication, and several copy of duplicated message is necessary for transmission, and it's the one who wastes network bandwidth, the only way to reduce communication overhead for group transmission is through MULTICAST, and thus in this paper, I would like to introduce a decentralized, multicast and negative acknowledge based messaging system – Fast Java Messaging 2 (FJM2) to you.



## **1.2 Motivation**

Java Message Service (JMS) proposed by Sun Microsystems is a set of Java API that allows applications to easily create, read, write, and deliver messages. JMS is a design created by Sun Microsystems and several partner companies since 1998. They define only a common set of application programming interfaces rather than constrain the way for implementation to create a Java Message Service (JMS) based Message-Oriented Middleware (MOM) systems. And thus JMS system could be a pure Java program or a combination of Java and some native languages, which usually have better performance than Java, but this way would also lose the ability to cross-platform.

Undoubtedly Java now is an importance software revolution during the recent years, it

plays a more and more essential role in the recently systems, and so does its subset application – JMS, however there neither no much thesis that discusses the detail about how to implement a decentralized, multicast and negative acknowledge based JMS systems, nor such official mature JMS products exists in the market. And thus we hope that our experience would benefit those who interests on such topic to establish an efficient, decentralized JMS system.

### **1.3 Overview**

This paper is organized as follow:

**Chapter 2 - Background & Related Works**, we'll take a short review of the background information and related works. We would briefly introduce the concept of Message-Oriented Middleware (MOM), Java Message Service (JMS), Negative-acknowledgment Oriented Reliable Multicast (NORM) [22][23] Protocol, Forward Error Correction (FEC) [24][25][26][28] and the previous multicast based JMS implementation: Fast Java Message (FJM) [13].

**Chapter 3 - Fast Java Messaging 2**, it would briefly introduce the architecture of the FJM2, and a little bit detail about its sub-components and program flows.

**Chapter 4 - FJM2 Protocol**, it abstracts the protocols in FJM2, and some minor modifications in the official NORM protocol.

**Chapter 5 – FJM2 Performance Analysis and Discussions**, we'll take some benchmarks on the existing JMS products and FJM2, and also have a short analysis on these results.

**Chapter 6 – FJM2 Implementation Issues and Discussions**, we'll abstract some implementation issues and have few analysis of FJM2 at this chapter..

**Chapter 7 – Conclusion**, in this chapter, we would have a conclusion and propose some possible future works for FJM2.

---

## CHAPTER 2 BACKGROUND & RELATED WORK

---

In this chapter, I would like to introduce the background information and related works to you. We would in turn take a short review on the following technologies: Message Oriented Middleware (MOM), Java Message Service (JMS), IP Multicast, Forward Error Correction (FEC), Negative-Acknowledgment Oriented Reliable Multicast (NORM) Protocol and finally the previous design – Fast Java Messaging (FJM).

### 2.1 Message Oriented Middleware (MOM)

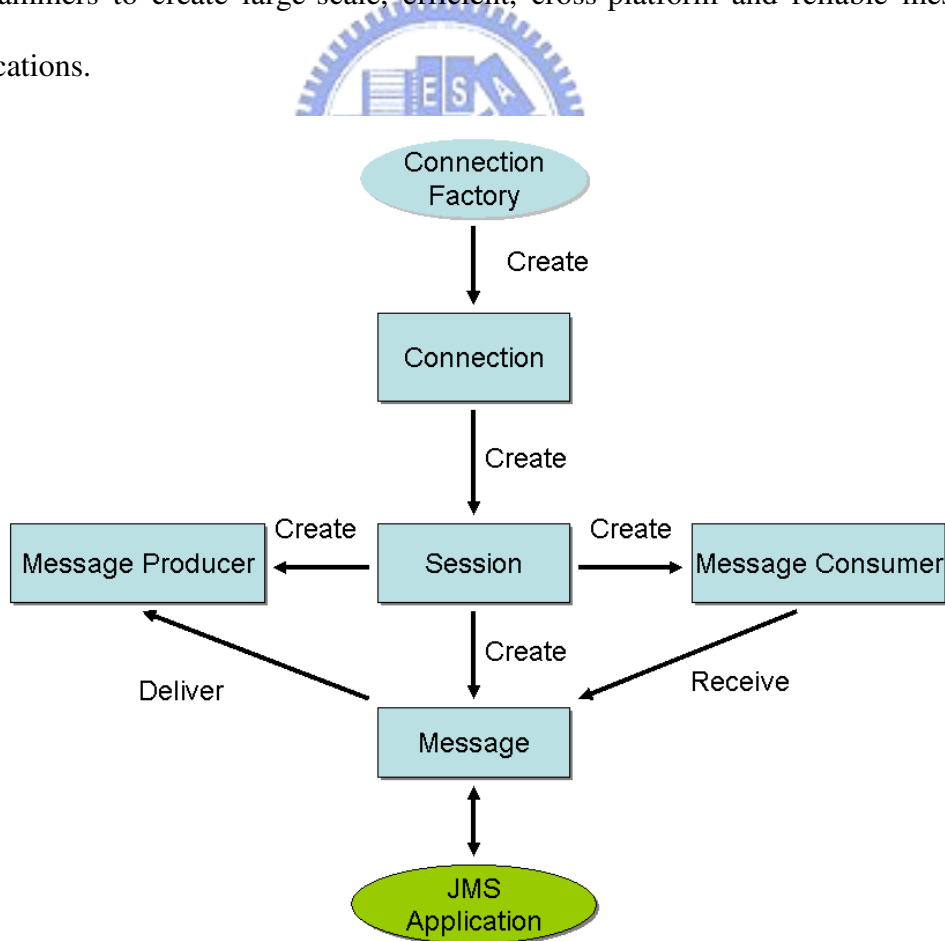
Message Oriented Middleware (MOM) is a popular technology among enterprise computing. Software applications or components can communicate with each other via messaging. Messages are composed by body and header, and are readable by receivers. The purpose of messaging is for one application to inform some specific events to another application. The responsibility of MOM is to ensure messages are sent appropriately and hide the network complexity from application programmers. Besides, most of the MOM support fault tolerant, load balancing, scalability and transaction to provide a reliable messaging platform.

MOM systems usually define some virtual destinations for message exchange. Messages are sent to a specific virtual destination, which might be a queue, a topic, or a specific network address. Applications interested in certain messages could get messages from some specific mechanisms. Moreover, the communication between applications is loosely coupled. It means that senders and receivers do not have to be

available at the same time, In other words, they could communicate asynchronously. The only things those user applications need to know are the message and message notification interfaces.

## 2.2 Java Message Service (JMS)

Java Message Service (JMS) proposed by Sun Microsystems is a set of Java API allows applications to read, write, and deliver messages. It defines only a common set of application programming interfaces and associated semantics that allow programs written in Java to communicate with each other via JMS architecture. JMS hides the network complexity from application programmers to make it become much easier for programmers to create large-scale, efficient, cross-platform and reliable messaging applications.



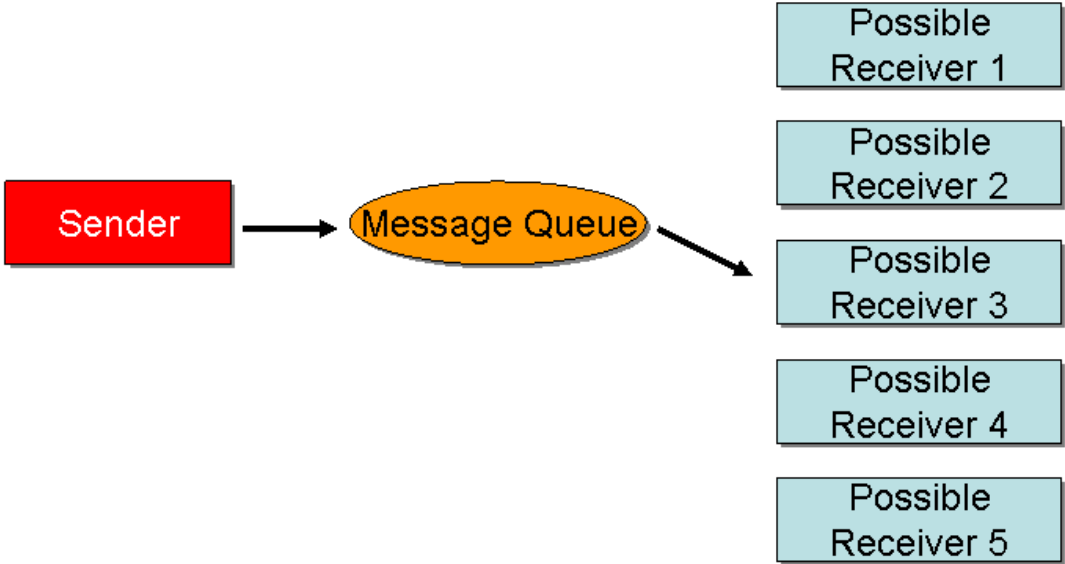
*Figure 2-1. JMS Programming Model*



Figure 2-1 above shows the abstract of JMS programming model, JMS application could play as a Message Producer or Consumer or even both of them at a time. And no matter which role they are, they must create the Connection and Session object instance through Connection Factory Object, and once Session object instance is ready, they could create Message Producer or Consumer object instance through Session object according to its requirement, and finally create the Message object instance for delivery and notification. Message Producers usually deliver messages to a specific Queue or Topic, and the JMS system would monitor such virtual destination object and notify the registered Message Consumers, and that's why we call such mechanism as 'Message Driven Model'.

There are two different models in JMS.

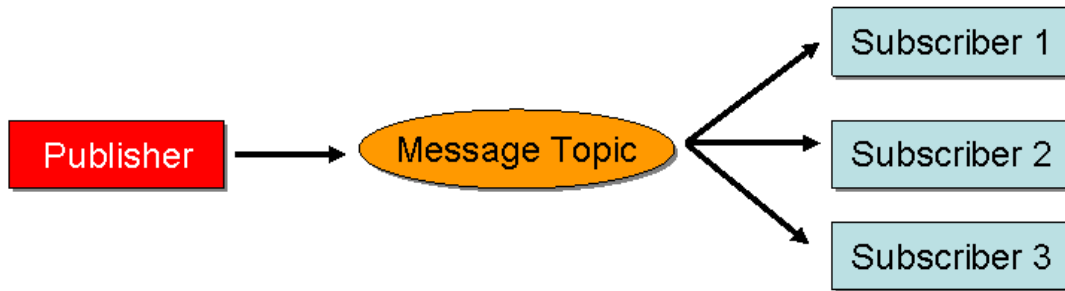
**(1). Point-to-Point (PTP) Messaging Model**



***Figure 2-2. JMS PTP Messaging Model***

As we could see from the figure above, sender could send out a message into a specific message queue, thus one and only one of the possible receivers could get this message.


## (2). Publisher / Subscriber Messaging Model



*Figure 2-3. JMS Publisher/Subscriber Messaging Model*

From the figure above, we could tell that sender could send out a message destined to a specific message topic, and all of the subscribers could get this message asynchronously.

## 2.3 IP Multicast



IP Multicast is a way to deliver a message into a group of receivers at a time. In this protocol we would feed the field of destination MAC address in Ethernet frame with a specific broadcast MAC address and the field of destination IP address inside IP header could be a specific group address ( 224.xxx.xxx.xxx ~ 239.xxx.xxx.xxx ), the group address is the ID for message grouping. However because of the nature of multiple receivers, while compared to UNICAST, it would be much more difficult to maintain the message synchronization in MULTICAST. In order to avoid some kind of Broadcast Storm Attack, most routers and gateways would not forward the MULTICAST traffic to WAN, even if there is already a well-defined multicast routing protocol in RFC and even when the specific MULTICAST routing daemon has already implemented on the router / gateway device, the MULTICAST routing functionality might be disabled by default, and thus MULTICAST usually works only

inside LAN environment, it could not communicate with each other across WAN.

So far there are many multicast specific researches:

**(1). Multicast Requirements, Performance, Survey and Taxonomy [5][6][7][18]**

These papers discuss some critical issues in the MULTICAST domain.

**(2). Multicast Transport Protocol [4]**

It describes a protocol for reliable transport that utilizes the multicast capability of applicable lower layer networking architectures

**(3). Internet Group Management Protocol (IGMP) [15][16][17]**

It's a protocol to manage the multicast group membership.

**(4). Protocol Independent Multicast (PIM)[14][21]**

It's a protocol that provides 1-to-1, 1-to-many, many-to-many transmission via multicast protocol.

**(5). Multicast Source Discovery Protocol (MSDP) [19]**

It is a computer network protocol in the Protocol Independent Multicast (PIM) family of multicast routing protocols.

**(6). Source-Specific Multicast (SSM) [20]**

It provides a network layer service through "channel", identified by an SSM destination IP address (G) and a source IP addresses.

**(7). Distance Vector Multicast Routing Protocol (DVMRP) [27]**

It's a protocol that defines the way for multicast routing daemons to exchange routing information.

**(8). Negative-acknowledgment Oriented Reliable Multicast (NORM) Protocol**

It's a protocol that adapts the negative-acknowledgment (NACK) and Forward Error Correction (FEC) technology to provide an efficient, multicast based, decentralized, reliable multicast based transmission service.

## **2.4 Forward Error Correction (FEC)**

Forward Error Correction (FEC) is a widely adapted technology in Telecommunication to control message corruptions, lost and provide a way for message recovery. It usually appends some parity contents into the source message, and while receiving enough parity contents, we could successfully recover the original source message. There are 2 different kinds of FEC mechanism:

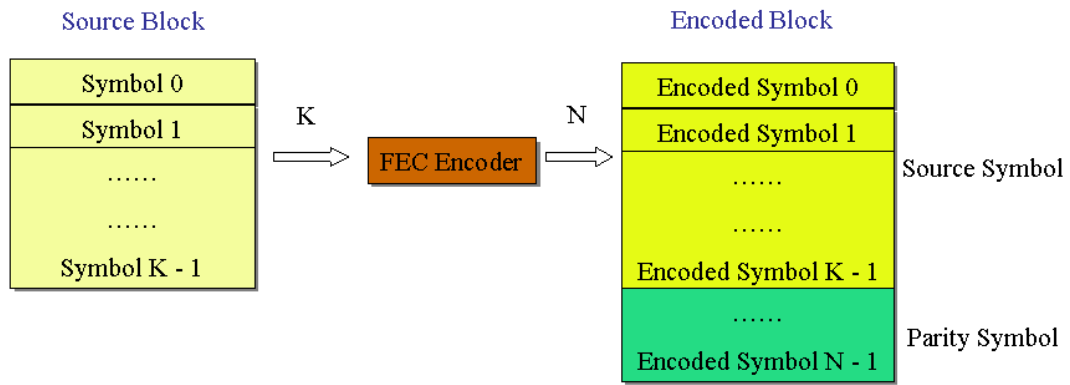
### **(1). Systematic FEC**

The original source symbol would appear among the encoded symbols.

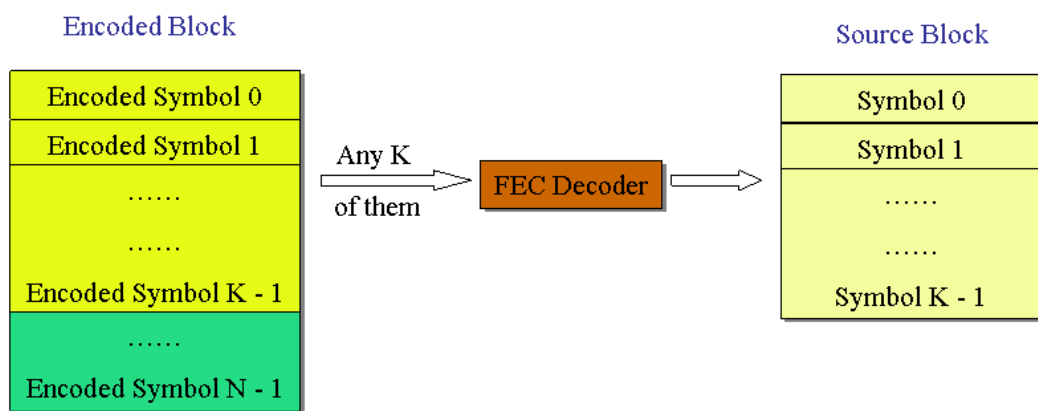
### **(2). Non-Systematic FEC**

The original source symbol would not appear among the encoded symbols.





**Figure 2-4. Systematic FEC Encoding**



**Figure 2-5. Systematic FEC Decoding**

Figure 2-4, 2-5 show us the generic Systematic FEC operation model, its basic concept is: Given a source block divided into any K of equal parts of source symbols, we could produce N pieces of encoded symbols through a FEC encoder; and given any K of the encoded symbols, we could re-produce the original source block. There are many types of FEC codes, but the most important by far is Reed-Solomon coding because of its widespread use on the Compact disc, the DVD, and in computer hard drives.

## **2.5 Negative-acknowledgment Oriented Reliable Multicast (NORM)**

The major objective of NORM protocol is to provide a one-to-many or many-to-many reliable, large-scale and robust bulk data transfer via multicast. The NORM protocol design provides support for distributed multicast session participation with minimal coordination among senders and receivers. NORM allows senders and receivers to dynamically join and leave multicast sessions at will with minimal overhead for control information and timing synchronization among participants. The following are notable characteristics of NORM protocol:

- (1). Providing message synchronization via common message header information.
- (2). NORM is designed to be self-adapting to a wide range of dynamic network conditions with little or no pre-configuration.
- (3). The protocol is purposely designed to be tolerant of inaccurate timing estimations or lossy conditions that might occur in many networks including mobile and wireless.
- (4). The protocol is designed to exhibit convergence and efficient operation even in situations of heavy packet loss and large queuing or transmission delays.

## **2.3 Fast Java Messaging (FJM)**

Fast Java Messaging is a former reference design of FJM2. It is a JMS system based on IP multicast protocol, negative acknowledgement and has a NACK based flow control. FJM offers only the publisher/subscriber model of JMS and its major objective is to provide a fast and reliable Java Message Service.

There are several key characteristics of FJM architecture:

**(1). It's a distributed system.**

Because FJM is a distributed design, there is no any centralized management node, such as a message dispatch server, in the entire FJM system. And thus message dispatch server is no longer the performance bottleneck of the system. FJM user applications exchange messages via only the JMS topics, which physically stands for a specific multicast addresses per topic. With benefit of IP multicast protocol, the number of packet delivery can be suppressed dramatically.

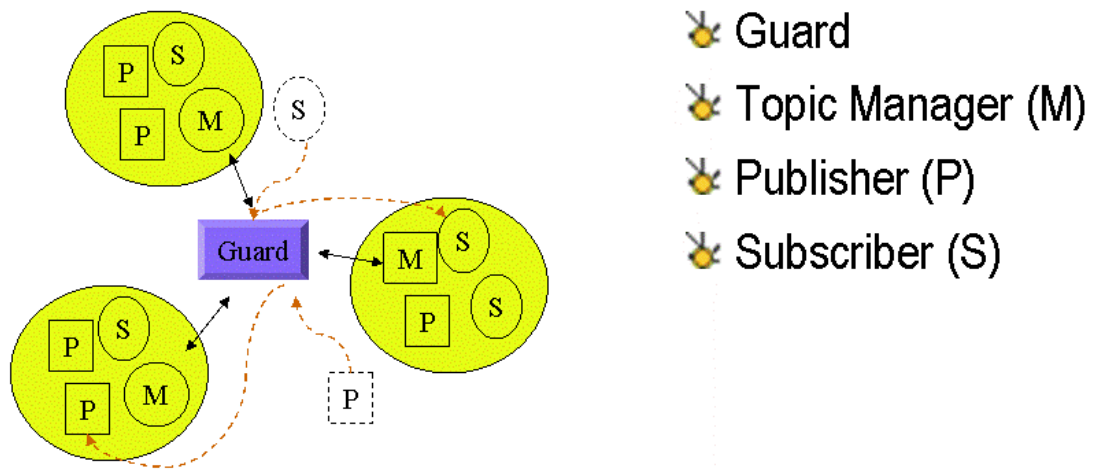
**(2). It adapts negative acknowledgement (NACK) for message reliability**

FJM adapts negative acknowledgement (NACK) rather than positive acknowledgment (ACK) approach to provide a reliable multicast transmission mechanism. Under NACK approach, subscribers send out NACK to publisher only when message loss occurs. And thus NACK could dramatically decrease the number of acknowledge packet delivery for message repair process.

**(3). It provides a NACK based flow control**

FJM also provides a NACK based flow control. A NACK message may indicate some states of subscribers immediately, such as system busy or buffer under-run .....etc. When publisher receives NAK from subscribers, it can slow down the message transmission rate, to let subscriber catch up with it. If publisher has not yet received any NAK message for a period of time, it may speed-up the transmission rate to gain a better throughput. With the help of NAK-based flow control scheme, FJM can provide a reasonable solution to the message exchange rate diversity between publishers and subscribers.

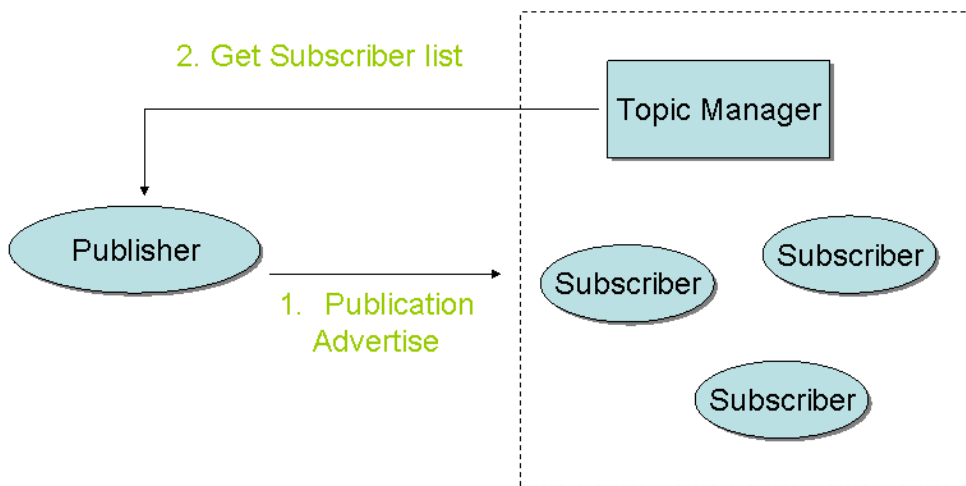
#### (4). Membership Management and Multicast-based Leader Election Protocol



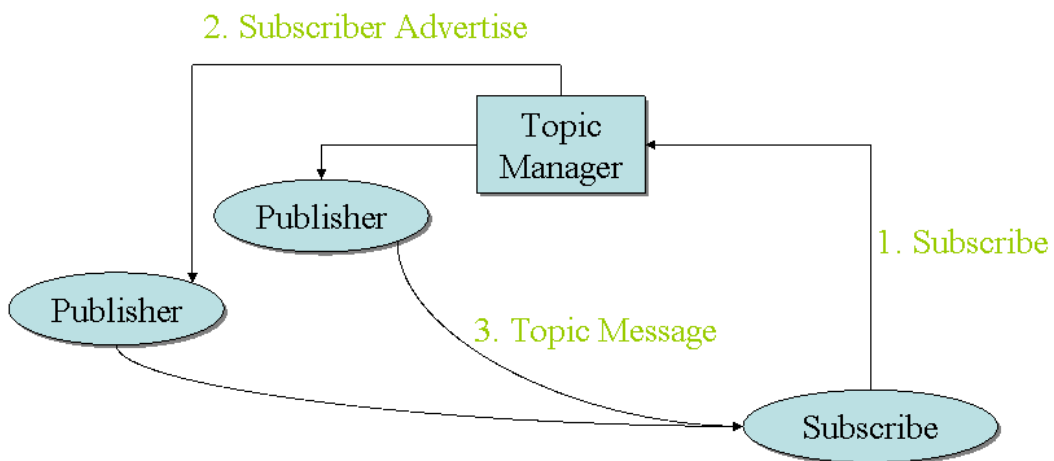
*Figure 2-6. FJM Membership Topology*

From figure above, we could tell that there are four roles in membership protocol: **Guard**, **Topic Manager**, **Publisher**, and **Subscriber**. Guard can be taken as a centralized manager. While there is a new publisher/subscriber wants to join the system, it must first contact with the Guard daemon for system information. And Guard is not dedicated to any participant. A Guard could be chosen from all the existing Topic Managers through the Guard election protocol. Finally let's take two simple examples to see how FJM process the publisher and subscriber registration procedures:





*Figure 2-7. FJM Publisher Registration Flow*



*Figure 2-8. FJM Subscriber Registration Flow*

**(5). Topic Address Binding**

Traditional JMS systems usually dispatch the Topic messages by the message header or content via a single data link. In FJM we propose a way to bind up the topic by a specific multicast group address to reduce software overhead.

However there are some incorrect view points in FJM, and they are the reasons those make me to propose a brand new design – Fast Java Messaging 2 (FJM2):

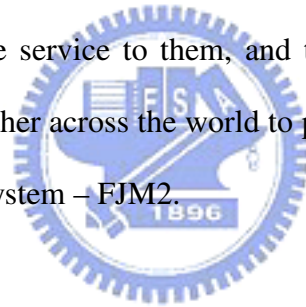
- (1). The performance of FJM is not good enough while adapting into 100Mbps network environment, because it requires at least 50 million seconds delay in inter message gap, or the system would crash immediately, and it's the root cause of the poor performance.
- (2). User application must be aware of the limitation described in (1), and a reasonable delay must be applied into every message publish routine, or FJM would encounter a serious system fault and goes crash.
- (3). While member-ship changes, the FJM program would hang up and consume the entire system resource. (CPU utilization is almost 100%)
- (4). Topic address binding would never benefit from the hardware dispatch for (Address, Port) binding, it's a misunderstanding in the assumption. Because most of the Ethernet Adapters in the world could only handle layer two (MAC Layer) protocol, while IP address sits inside layer three, and port number is in layer four, and even in the recently system-on-a-chip (SoC) router design, layer four switching functionality is rarely available and never be implemented in a full specification. Beside this, Topic address binding is really a painful characteristic that make management, configuration become much more complicated and also make it difficult to implement and maintain a FJM tunnel daemon to provide WAN traversal ability.

---

## CHAPTER 3 Fast Java Messaging 2

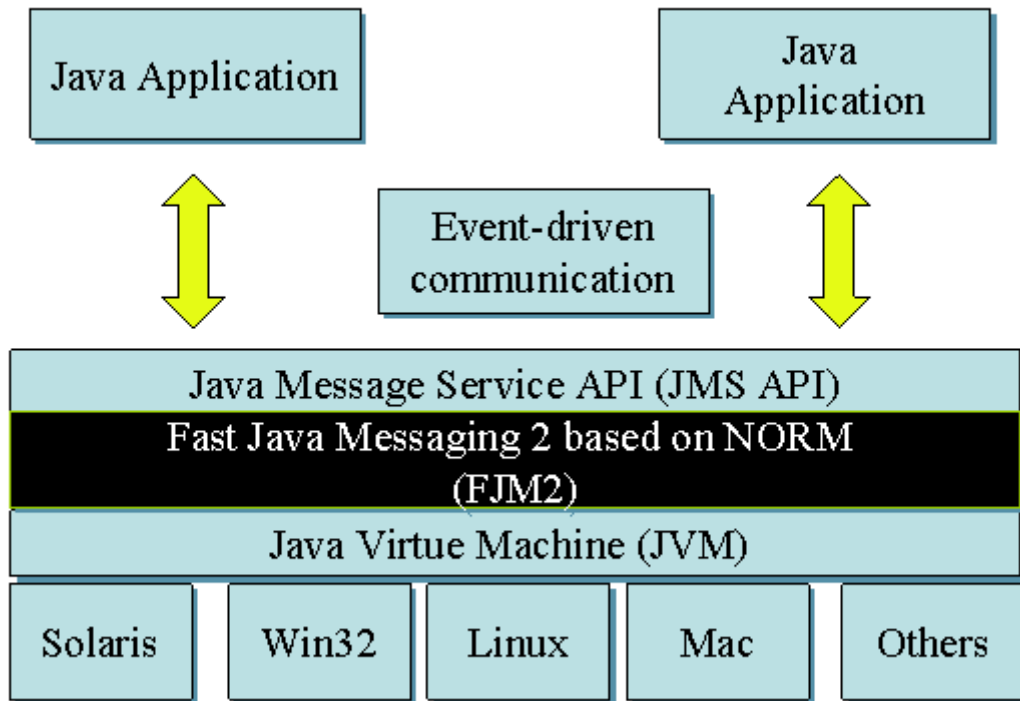
---

In this chapter, I would like to introduce a de-centralized Java Message Service System to you. Its name is Fast Java Messaging 2 (FJM2), it's a pure Java design and thus has a cross-platform characteristic in natural, and its major objective is to provide a de-centralized, reliable, efficient, multicast based Java Message Service (JMS) with minimal configuration overhead. And in order to provide the WAN traversal ability for this multicast based system, here we also propose a micro multicast tunnel design - Fast Java Message 2 Daemon (FJM2D), and we also introduce a small web based administration program - FJM2Admin, which provides FJM2D node list exchange service to them, and thus FJM2D could automatically communicates with each other across the world to provide a large-scale coverage for this multicast based JMS system – FJM2.



### 3.1 FJM2 System Architecture

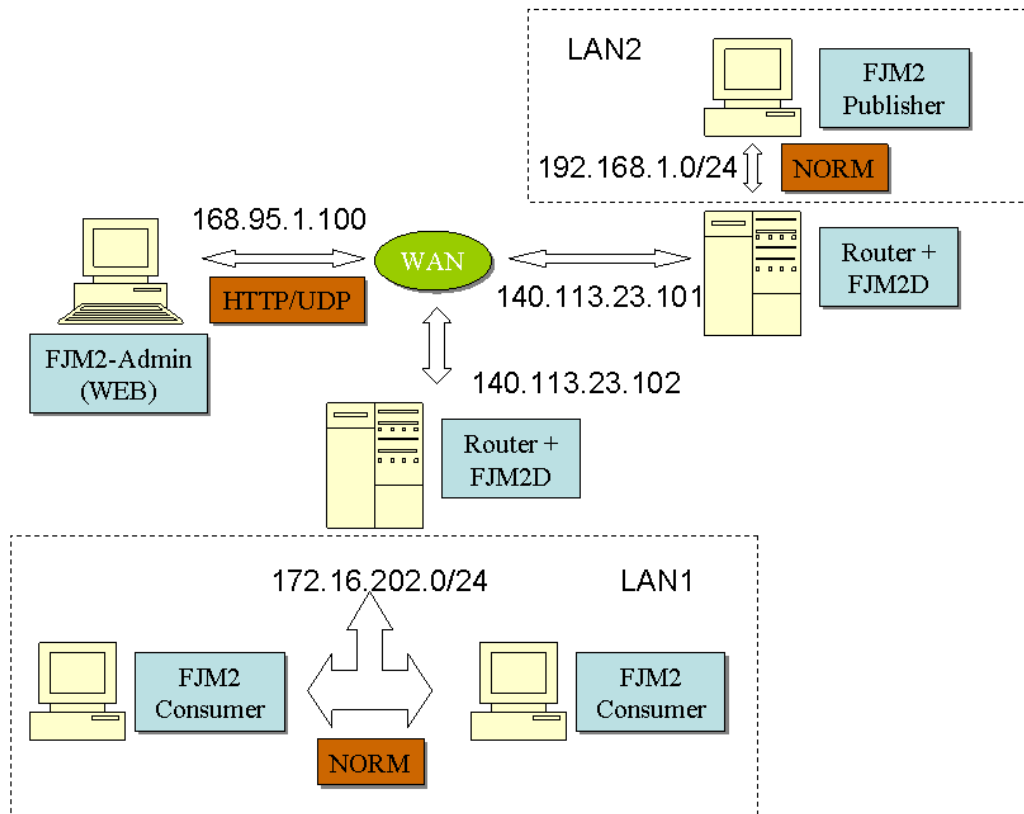
#### 3.1.1 FJM2 Architecture Overview



*Figure 3-1. FJM2 Architecture Overview*

From figure 3-1 above, we could tell that Fast Java Messaging 2 (FJM2) is a Message Oriented Middleware (MOM) that abstracts the complexity of network programming to shorten the time for a robust network application development. The only thing that application developers need to know is the set of JMS API, all the underlying technology hides in a black-box called JMS from programmers, if they want, programmers could even change the underlying products to gain better performance or stability, and only a few program modification needs to be done. In other words FJM2 provides not only the solution for the Time-to-Market issue, but also an opportunity to developer for all the benefits come from the cross-platform characteristic.

### 3.1.2 FJM2 Physical Architecture Overview



*Figure 3-2 FJM2 Physical Architecture*

In figure 3-2, we could have a rough abstraction of the entire FJM2 physical architecture, and the following characteristics could also be noticed:

- (1). FJM2-Admin Daemon MUST have a public IP address to make every FJM2D be able to locate and communicate with it.
- (2). FJM2D usually runs at the router which might has some firewall functionalities, in such case, we might need to do some specific firewall configuration to make the communications between FJM2D and FJM2-Admin become valid.
- (3). FJM2D MUST locate on the edge gateway, so that they could forward the FJM2 specific multicast traffic onto a specific FJM2D endpoint across WAN.

### 3.2 FJM2 Message Transmission Model

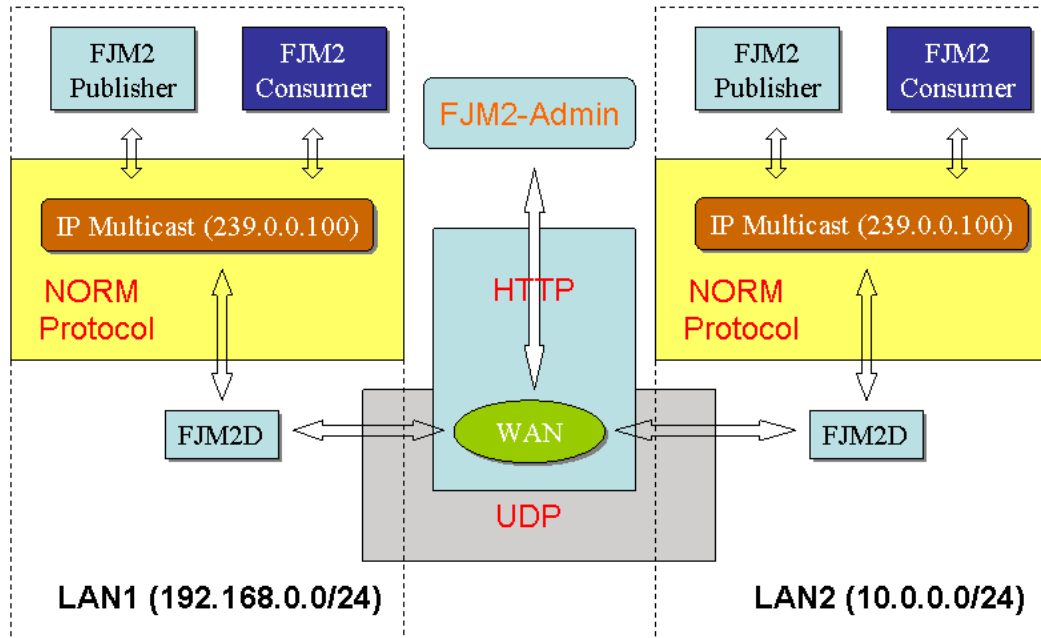


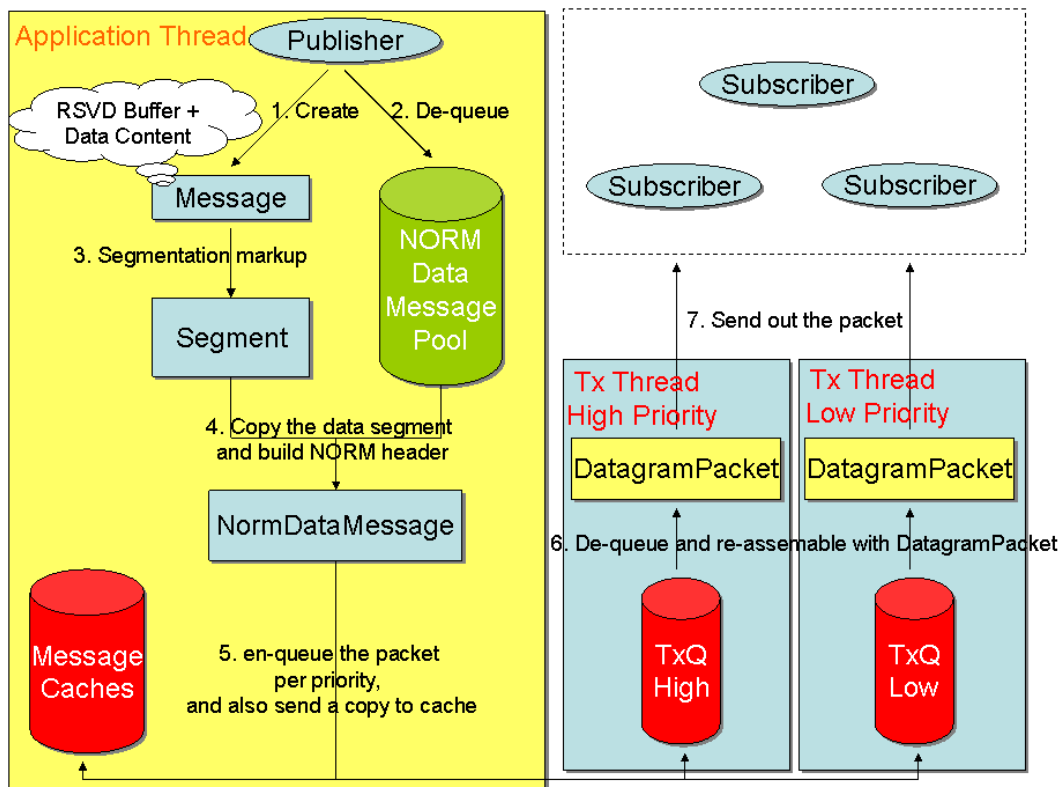
Figure 3-3. FJM2 Message Transmission Model

From figure 3-3, we could realize:

- (1). There is no necessary to have a centralized daemon to help FJM2 Publishers and Consumers to communicate with each others inside LAN environment
- (2). We MUST install a FJM2D on the edge gateway of the local network to make it possible to forward the FJM2 message to another FJM2D across WAN for us.
- (3). FJM2-Admin serves only FJM2D, it would never communicate with FJM2 daemons directly, so that the configuration requirement could be minimized.
- (4). The communicate protocol between FJM2 daemons (Publishers and/or Subscribers) is a NORM protocol based on IP multicast and NACK.
- (5). The communicate protocol between FJM2 and FJM2D is a NORM protocol based on IP multicast and NACK.

- (6). The communicate protocol between FJM2D daemons is a NORM message delivered by UNICAST.
- (7). The communicate protocol between FJM2-Admin and FJM2D daemons are UNICAST UDP (FJM2D List Advertisement) and HTTP protocol (Registration and Query Procedures).
- (8). The local network behinds FJM2D could be a different subnet.

### 3.3 FJM2 Message Publish Program Flows



*Figure 3-4. FJM2 Message Publish Program Flows*

From figure 3-4, we could realize how it flows in FJM2 software components while publishing messages, this figure does not include the procedure of message repair process, and thus it is merely a general program flow upon message publishing.

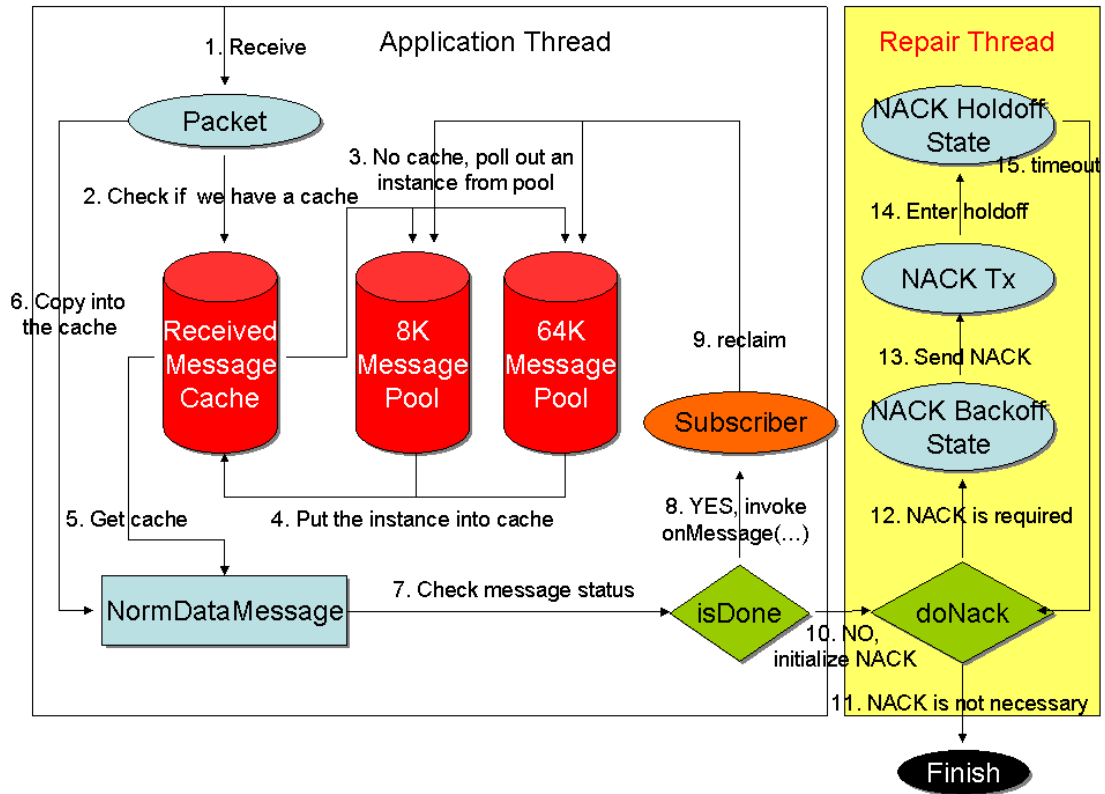
There is no necessary to determine a multicast leader (Guide) as what we've done in previous FJM design, the FJM2 daemon could immediately enter normal message handling procedures after startup, and while membership changes, only a little redundant, incomplete message cache would be occupied in consumer/subscriber's memory for a period of time, it would be free upon cache expire. So the entire system overhead is much lower than the previous FJM, and thus it would be much easier for FJM2 to develop a stable messaging service than FJM. Now let's take a look at the flows:

- (1). Publisher creates a Message object instance and fills in the data into its body.
- (2). Publisher de-queue a NormDataMessage object instance from the message pool, this object instance is the container for the message.
- (3). Segmenting the message to fit in the message payload that best fits in current network environment, however the segmentation here generates only the marks about how the fragments should be taken.
- (4). Copying the segmented message into the NormDataMessage object instance which de-queued from the pool in step (2), and finally build up the header for it.
- (5). En-queue the prepared NormDataMessage object instance into the proper transmission queue according to its message priority, and finally make a copy of the message into the MessageCaches for a possible message repair process that might be initialized later.
- (6). Each transmission queue has a dedicated stand-alone monitoring thread that has a proper thread priority that fits to the queue priority, while the queue is not empty, and the thread would de-queue a message from it and process a generic transmission routine, and once queue becomes empty, the thread would be placed into the wait pool of its monitoring queue and enter idle state until someone notifies it.



- (7). Once the transmission thread de-queue a message from the queue successfully, it could send it out through the socket library.

### 3.4 FJM2 Message Subscribe/Consume Program Flows



*Figure 3-5. FJM2 Message Subscribe/Consume Program Flows*

From figure 3-5, we could realize how it flows in FJM2 software components while subscribing/consuming messages, this figure also includes the procedure of message repair process. There is no necessary to determine a multicast leader (Guide) and the registration processes those are necessary in previous FJM, the FJM2 daemon could entering normal message handling procedures immediately after startup, and while membership changes, only a little redundant, incomplete message cache would be occupied in consumer/subscriber's memory for a period of time, it would be free upon cache expire. So the entire system overhead is lower than FJM, and thus it

would be much easier for FJM2 to develop a stable messaging service than FJM.

Now let's take a look at the flows:

- (1). Application receives a packet (symbol) from the network, and then put it into the packet buffer.
- (2). Build a key for message cache from the header of the received packet, and the use for the cache lookup, if there is a corresponding cache to this key, then the program flows to step (5), else flows into step (3).
- (3). Poll out a message container from the message pool, and the message pool would be selected according to the total message size recorded in its common header information.
- (4). Put the message container instance into the message cache.
- (5). Treat the object which is just fetch from message cache or polled from the message pool as a NormDataMessage object.
- (6). Copy the data inside the packet buffer into the NormDataMessage object.
- (7). Let's check if the message is complete or not, if it's true then flows into step (8), else flows into step (10).
- (8). If the message is already a complete one, invoke onMessage(...) of all the registered subscriber applications.
- (9). When onMessage(...) returns, reclaim the message object to the pool.
- (10). If the message is not yet complete, it would try to initialize NACK process, and if the NACK process is really necessary it then flows into step (12), if it isn't then flows to step (11)
- (11). Although the message is incomplete, the NACK process is still not yet necessary in this case, so terminate this receive event handle.
- (12). If the message is incomplete and it's time for NACK process, initializes NACK process and enters NACK back-off state.

- (13). During NACK back-off state, the subscriber would monitor the NACK appears on the network, and if it match the one recorded in its own NACK items, it suppress the NACK for that item. When back-off timeout occurs, the program would enter NACK transmission state.
- (14). In NACK transmission state, the program would send out the NACK message through socket library, and then enter hold-off state.
- (15). The hold-off state is the way to avoid repeated NACK message in a given delay timeout.
- (16). While hold-off timeout occurs, the program would flows back to NACK decision state to determine the next step is (11) or (12).

### 3.5 FJM2 Management Program Flows

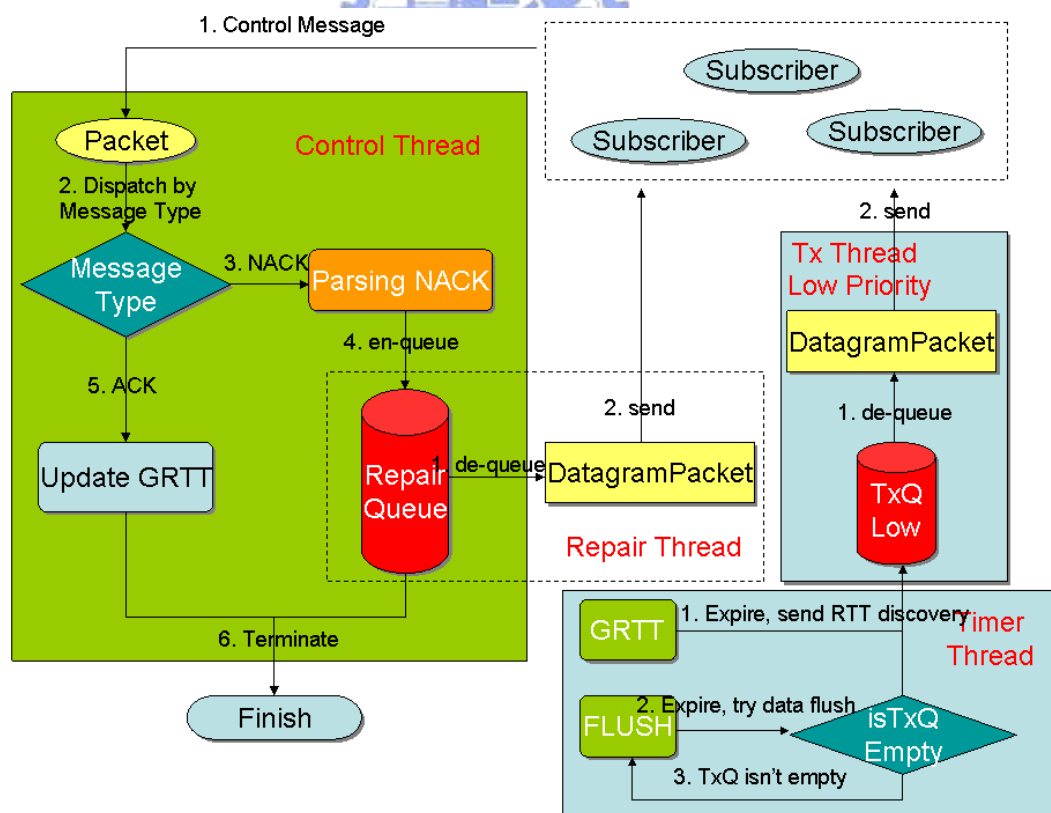


Figure 3-6. FJM2 Management Model and Flows

In figure 3-6, there are 4 threads involved in FJM2 management / control process:

(1). Control Thread

It's responsible for two different functions in FJM2, the first is message repair functionality and the other one is Greatest Round-Trip Time (GRTT) calculation. The program flows of this thread are :

1. Receive control message from FJM2 subscriber/consumer
2. Dispatch the message handle routine by its message type
3. If this control message is a NACK message, let's parse the NACK message to get NACK item list.
4. En-queue the NACK item list generated from step (3) into the repair queue, and goes to step (6) to terminate this operation.
5. If this is a ACK message, update the Greatest Round-Trip Time (GRTT).

(2). Repair Thread

This thread would monitor the repair queue, if it's not empty then de-queue a message and send it out, other-wise put itself into the wait pool of the repair queue and enter idle state until someone notifies it.

(3). Timer Thread

This thread would periodically activates registered routines, and right now there are two different timer-based routines:

1. GRTT

The objective of this routine is to gather and generate the greatest round-trip time (GRTT) from all the existing FJM2 subscribers.

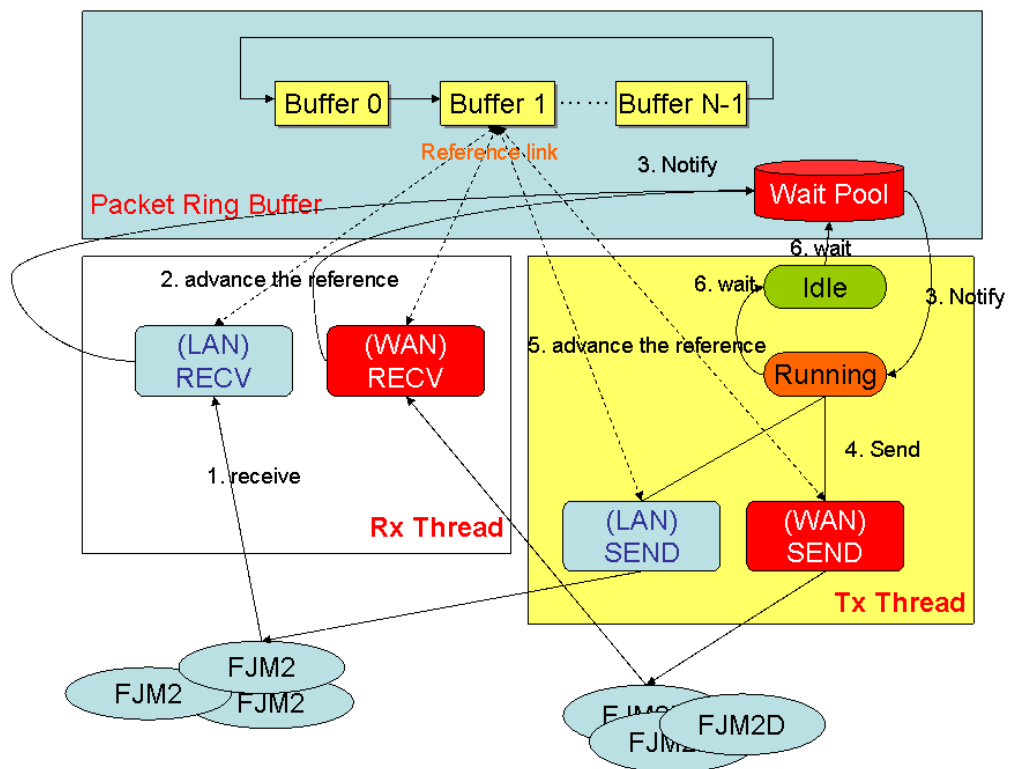
2. FLUSH

The objective of this routine is to tell all the existing FJM2 subscribers that this transmission is finished while transmission queue is empty.

(4). Transmission Thread with Low Priority

This thread is same as the one we discussed in session 3-3. *FJM2 Message Publish Program Flows.*

### 3.6 FJM2D System Architecture



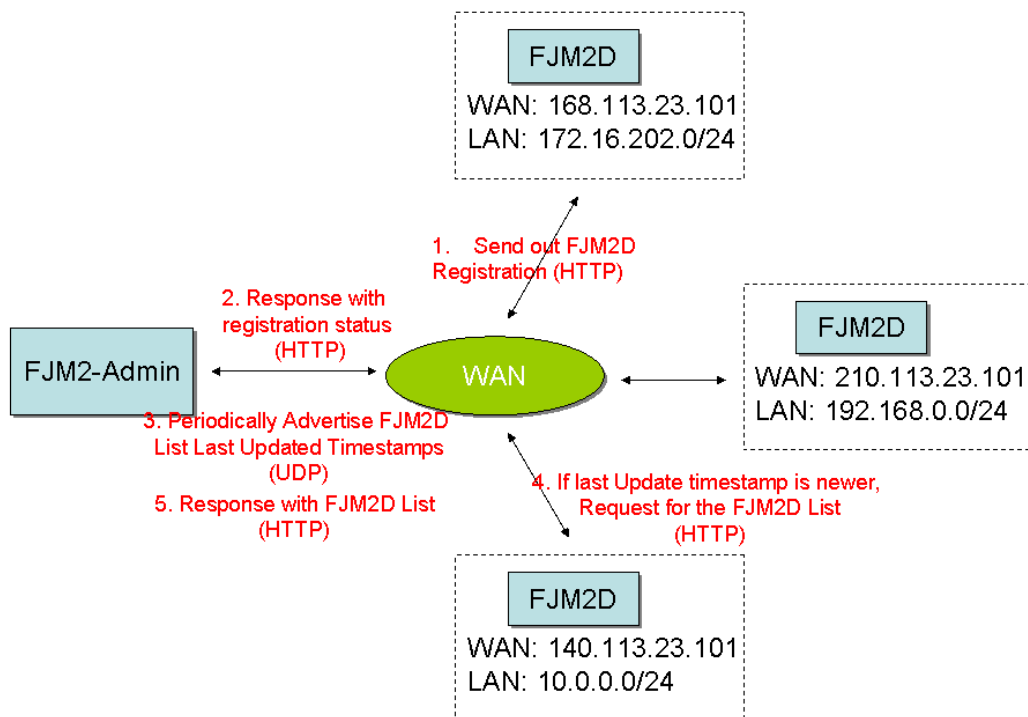
**Figure 3-7. FJM2D Architecture and Operation Flows**

Figure 3-7 gives us a rough picture of FJM2D architecture, it consists two threads: Rx Thread (It's also the main thread), Tx Thread. Here let's take a look at how it works:

- (1). FJM2D receives messages from FJM2 daemons, and place the data into the packet ring buffer
- (2). FJM2D advances the free pointer which points to the packet ring buffer

- (3). FJM2D notify the thread which is inside the wait pool of packet ring buffer.
- (4). FJM2D transmission thread is back to the running state, and then send out packet which referenced by the occupied pointer.
- (5). FJM2D transmission thread advance the occupied pointer
- (6). FJM2D transmission found that there is no more packet waiting for transmit, it would then goes back to the wait pool.

### 3.7 FJM2-Admin System Architecture



*Figure 3-8. FJM2-Admin Architecture and operation flows*

FJM2-Admin is a Web Application Archive (WAR) defined in Java 2 Enterprise Edition (J2EE), it's only a plug-in for Java Application Server, and of course, all the stability and robustness the application could offer comes from the Java Application

Server. As we could see in Figure 3-3, there are 2 different protocol used in the communication between FJM2D and FJM2-Admin: UDP and HTTP. UDP stands for the management information advertisement, while HTTP stands for the general data link for FJM2D and FJM2-Admin. Finally let's take look on the operation flows:

- (1). While startup, FJM2D would initializes a registration process via HTTP.
- (2). FJM2-Admin returns a registration response to FJM2D via HTTP.
- (3). FJM2-Admin would periodically advertise the last modified timestamps of the active FJM2D list to all the registered FJM2D via UDP protocol.
- (4). FJM2D would send out a query for the latest FJM2D list if it found that its own list is out-of-date.
- (5). While receiving the query request, FJM2-Admin would give FJM2D a reply to inform it the latest FJM2D list.



Through the operation describes above, the FJM2D nodes could communicate with each others cross the world automatically, and the only thing that we have to configure manually is the address of the FJM2-Admin.

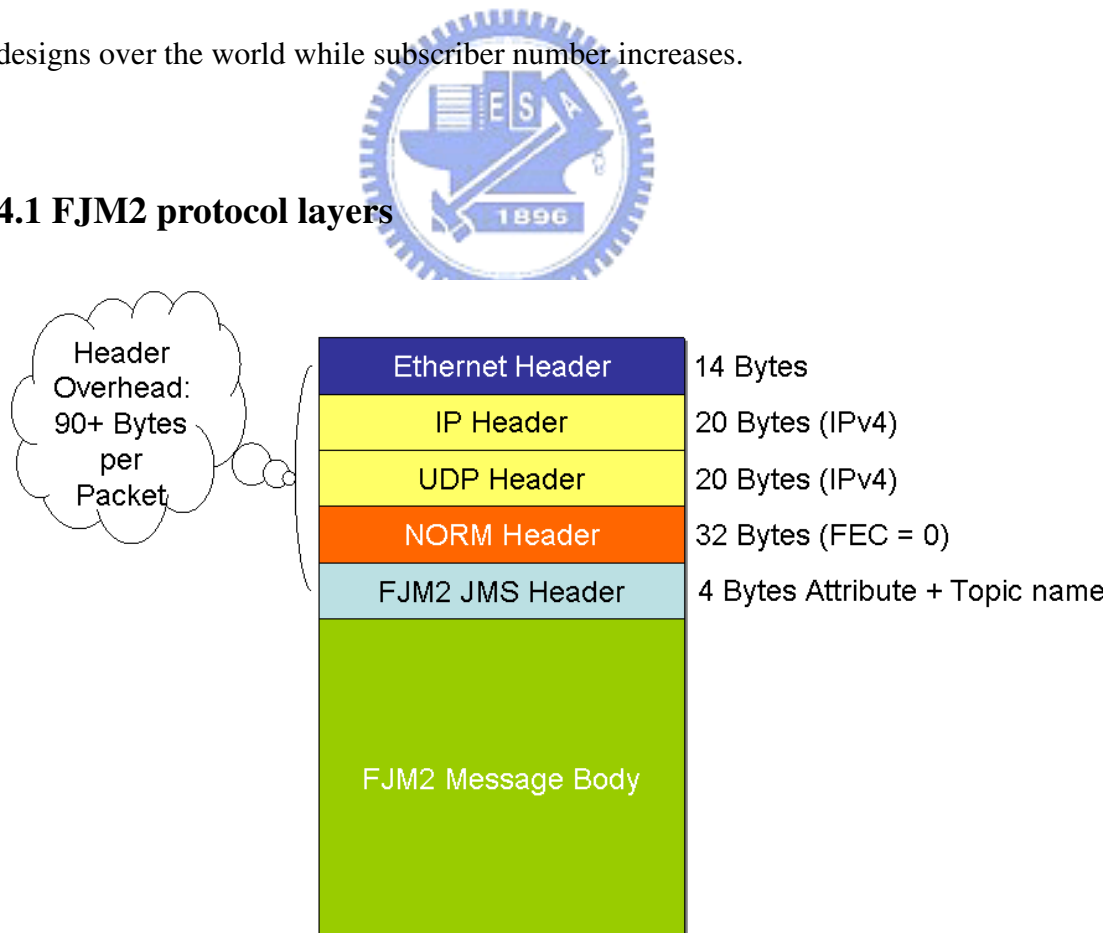
---

## CHAPTER 4 FJM2 Protocol

---

In this research, we propose a distributed JMS provider architecture named FJM2. FJM2 is based on NORM protocol has the improved performance than the previous FJM and included new features, such as WAN traversal capability and automatically FJM2D self-configuration. FJM2 is a JMS implementation which is implemented in pure java to provide the best portability. It offers only publish /subscriber messaging model through multicast protocol. Due to the distributed architecture of FJM2, load balance can be achieved, and the message server is no longer the bottleneck of the system, so now it could reach a much better performance than all of the server based designs over the world while subscriber number increases.

### 4.1 FJM2 protocol layers



*Figure 4-1. FJM2 Protocol Layers*



In figure 4-1 at last page, we could tell that the minimum FJM2 header overhead is 90+ bytes per packet, in other words, the maximum transfer unit (MTU) in the local network must be larger than this value minus the size of Ethernet Header (14), while compared to the general UNICAST packet, we lost around 2.7% performance in case of MTU = 1500 with 1-to-1 transmission.

Beside general TCP/IP header overhead, we still have two header overheads for two different protocols:

**(1). NORM Protocol**

This is the multicast based transport protocol that we adapts as the underlying protocol for FJM2. It provides the reliability, distributed, large-scale messaging architecture for FJM2 to form a Java Message Service (JMS) Provider.

**(2). FJM2-JMS Protocol**

It is an additional header for FJM2 to adapt alone with the header of NORM to build up a JMS compatible message object.



## **4.2 NORM protocol – A brief overview**

Negative-acknowledgment (NACK) Oriented Reliable Multicast (NORM) Protocol is defined in RFC 3940, 3941. It's designed to provide a reliable transport of data from one or more senders to a group of receivers over an IP multicast network. Its major objective is to provide an efficient, scalable, and robust bulk data transfer over multicast network. It also support for distributed multicast session participation with minimal coordination among senders and receivers.

### **4.2.1 NORM Delivery Service Model**

A NORM protocol instance (NormSession) is defined within the context of participants communicating connectionless packets over a network using pre-determined addresses and host port numbers. Generally, the participants exchange packets using an IP multicast group address, but UNICAST transport may also be established or applied as an adjunct to multicast delivery. In the case of multicast, the participating NormNodes will communicate using a common IP multicast group address and port number that has been chosen via means outside the context of the given NormSession.

NORM provides for three types of bulk data content objects (NormObjects) to be reliably transported. These types include:

- (1). Static computer memory data content (NORM\_OBJECT\_DATA)
- (2). Computer storage files (NORM\_OBJECT\_FILE)
- (3). Non-finite streams of continuous data content (NORM\_OBJECT\_STREAM).

The distinction between NORM\_OBJECT\_DATA and NORM\_OBJECT\_FILE is simply to provide a "hint" to receivers in NormSessions serving multiple types of content as to what type of storage should be allocated for received content (i.e., memory or file storage).

### **4.2.2 NORM Scalability**

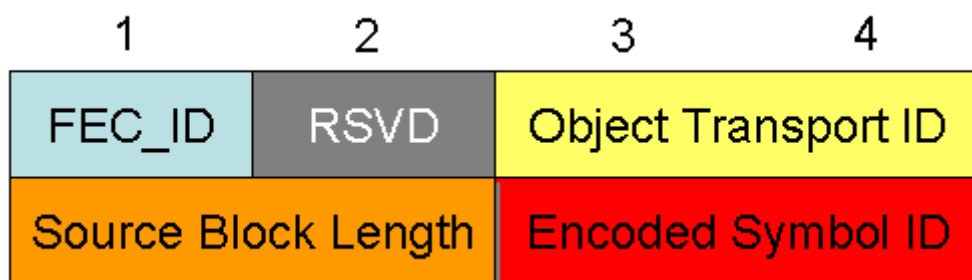
Group communication scalability requirements lead to adaptation of negative acknowledgment (NACK) based protocol schemes when feedback for reliability is required. NORM is a protocol centered around the use of selective NACKs to request repairs of missing data. NORM provides for the use of packet-level forward error correction (FEC) techniques for efficient multicast repair and optional proactive transmission robustness. FEC-based repair can be used to greatly reduce

the quantity of reliable multicast repair requests and repair transmissions in a NACK-oriented protocol. The principal factor in NORM scalability is the volume of feedback traffic generated by the receiver set to facilitate reliability and congestion control. NORM uses probabilistic suppression of redundant feedback based on exponentially distributed random back-off timers. NORM dynamically measures the group's roundtrip timing status to set its suppression and other protocol timers. This allows NORM to scale well while maintaining reliable data delivery transport with low latency relative to the network topology over which it is operating.

### 4.3 NORM Protocol in FJM2

#### 4.3.1 Forward Error Correction Algorithm in FJM2

In FJM2, we adapts Compact No-Code FEC scheme in the low level transport protocol – NORM Protocol. It's a Fully-Specified FEC scheme corresponding to FEC Encoding ID 0. And it does not require FEC encoding or decoding. Instead, each encoding symbol consists of consecutive bytes of a source block of the object. The FEC Payload ID consists of two fields, the 16-bit Source Block Number and the 16-bit Encoding Symbol ID.



**Figure 4-2 Compact No-Code FEC ("fec\_id" = 0) "fec\_payload\_id" Format**

Figure above is the fec\_payload\_id format used for Compact No-Code FEC in FJM2, it is one word smaller than Small Block, Systematic ("fec\_id"=129).

The reason why I choose Compact No-Code FEC is due to performance consideration. Generic FEC Encode/Decode is really a time consume process. Please refer to the test below for detail:

**(1). Testing Environment**

Item	Description
CPU	Intel Pentium M 740 (1.73 GHz)
SDRAM	1 Giga-Bytes
Java Runtime	Sun Microsystems 1.4.2_04
Java FEC Library	Onion Networks Java FEC Library v1.0.3 [29]

**(2). Performance Matrix**

Symbol	512 Bytes	1024 Bytes	1440 Bytes
Source Data			
1 Kbytes	0 ms	0 ms	0 ms
2 Kbytes	0 ms	0 ms	0 ms
4 Kbytes	0 ms	0 ms	0 ms
8 Kbytes	0 ms	0 ms	0 ms
16 Kbytes	1 ms	0 ms	0 ms
32 Kbytes	6 ms	2 ms	1 ms
64 Kbytes	35 ms	11 ms	7 ms

Therefore in best case, Java FEC library could encode  $(1000 / 7 = 142.8)$  64Kbytes Message per second, while  $100 \text{ Mbps} = (100 / 8 * 1024 / 64 = 200)$  64Kbytes Message per second. And thus we could have a conclusion that while Java FEC is adapted, the overall system would never exceed  $100 * (142.8 / 200) = 71.4 \text{ Mbps}$

### **4.3.1 NACK Algorithm in FJM2**

The NACK Algorithm used in FJM2 is almost exactly the same as the one described in NORM protocol, except the sender NACK process. Because in FJM2 we adapt Compact No-Code FEC scheme as the symbol algorithm, so that sender NACK suppress is meaningless to FJM2, and thus the sender NACK algorithm in FJM2 is merely sending out a repair symbol as soon as possible when the it got NACK message and it still has the message symbol cache for this lost message, and there is no any timer used in sender NACK process.



---

## CHAPTER 5 FJM2 PERFORMANCE ANALYSES

---

In this chapter, I'll show you the network performance benchmark of Java V.S C. And we will how fast it could be, could it meet our requirement, and what's the root cause here. And does MULTICAST run as fast as UNICAST in 1-to-1 transmission? Finally we will have a throughput benchmark on FJM2, SonicMQ [30], Fiorano [31] and iBus [32].

### 5.1 Test Environment

#### (1). Hardware Equipment

ID	Processor	RAM	Operation System
PC1	Pentium M 740 1.73GHz	1 GB	WINXP Home SP2
PC2	Pentium 4-M 1.80GHz	512 MB	WINXP Professional SP2
PC3	Pentium(R) 4 1.80GHz	256 MB	WIN2000 Professional SP4
PC4	Pentium M 740 1.73GHz	1 GB	Mandrake 9.2 (Linux 2.4.22)
PC5	Pentium 4-M 1.80GHz	512 MB	Mandrake 9.2 (Linux 2.4.22)

#### (2). Java Runtime Version

Windows: IBM Java(TM) 2 Runtime Environment "1.4.2"

Linux: IBM Java(TM) 2 Runtime Environment "1.5.0"

#### (3). Tool-chain Information

Windows: lcc-win32

Linux: gcc 3.3.1

## 5.2 UDP Performance Benchmark on Loop-back Interface

This is a throughput benchmark of 1-to-1 transmission on loop-back interface, and here we would also try to figure out what's the performance difference between Java and C on Windows XP.

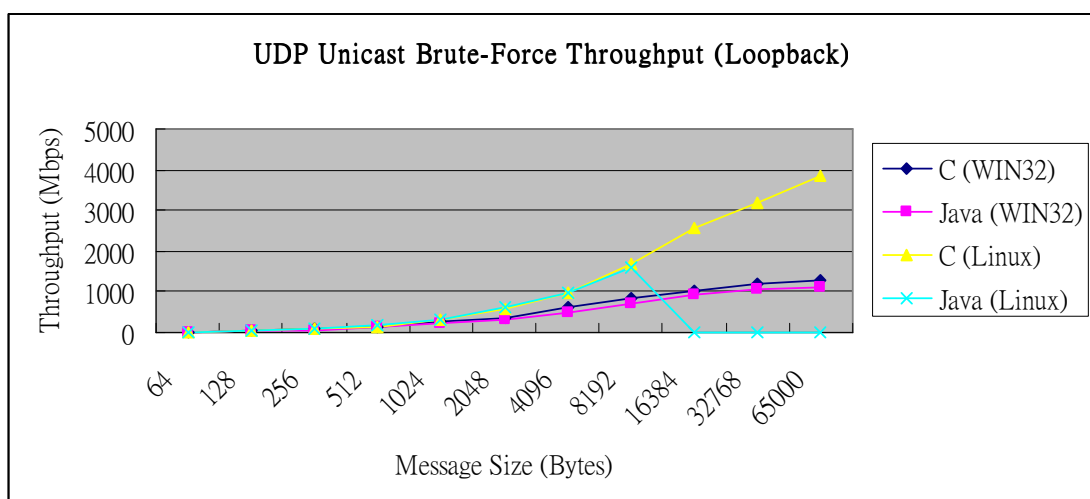
### (1). Test Scenario

1. Launch the test program, and then follow the parameters to generate a fixed-size message buffer with a randomly generated content.
2. The sender program would try its best to send out the whole message in a single socket function – `sendto(...)`
3. The receiver program would enter a infinite loop and use a single socket function – `recvfrom(...)` to receive message, and then calculate the performance value without message verification.

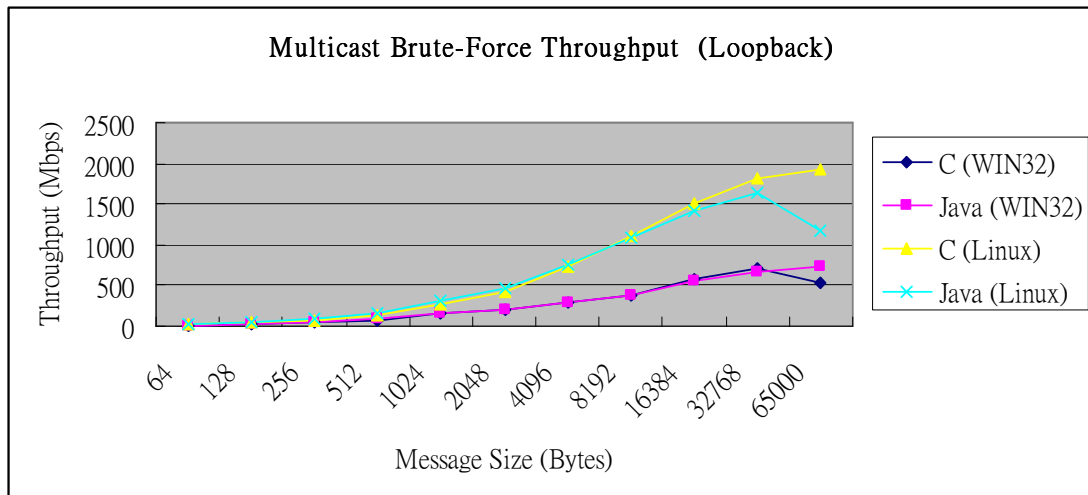
### (2). Test Objective

This test would show us how fast it could be while we have an unlimited network bandwidth, and how much difference between C and Java.

### (3). Test Benchmark Figures



**Figure 5-1. UDP UNICAST Throughput on Loop-back Interface**



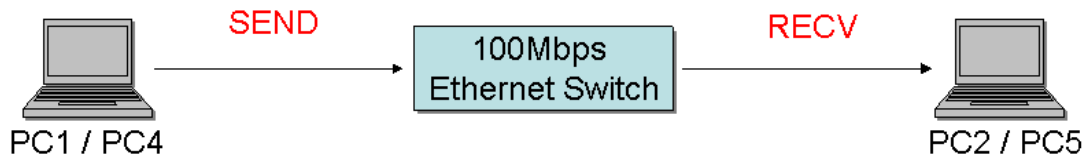
**Figure 5-2. UDP MULTICAST Throughput on Loop-back Interface**

#### **(4). Test Analysis**

1. Linux obviously outperforms Windows platform.
2. UNICAST always a little bit outperforms MULTICAST in a 1-to-1 transmission pair.
3. When message size is less than 32 Kbytes, Java could run as fast as Native C program in low level socket operations.
4. In UNICAST transmission, while message size exceeds 8 Kbytes, the Java (Linux) would hang up.
5. In MULTICAST transmission, while message size exceeds 32 Kbytes, the performance of Java (Linux) and Native C (WIN32) would start to descend.
6. Throughput grows with message size.
7. No matter in program written in C or Java, it could easily fulfill the performance requirement of 100Mbps Ethernet.



### 5.3 UDP Performance Benchmark on 100Mbps Ethernet Interface



*Figure 5-3. 1-to1 performance test connection topology*

Figure 5-3 shows us the connection topology in this test, and here we would do some benchmark to figure out what's the difference between Java and C on Windows XP.

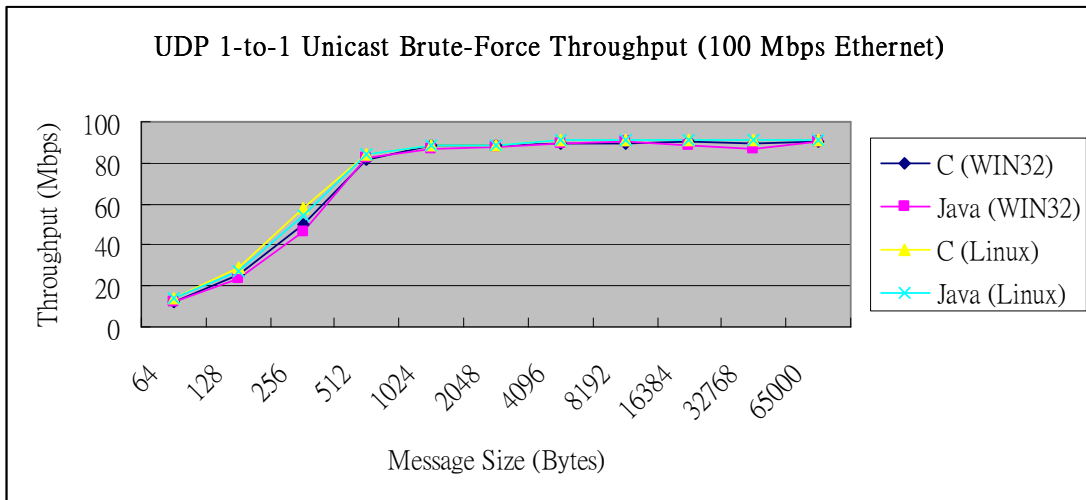
#### (1). Test Scenario

1. Launch the test program, and then follow the parameters to generate a fixed-size message buffer with a randomly generated content.
2. The sender program would try its best to send out the whole message in a single socket function – `sendto(...)`
1. The receiver program would enter a infinite loop and use a single socket function – `recvfrom(...)` to receive message, and then calculate the performance value without message verification.

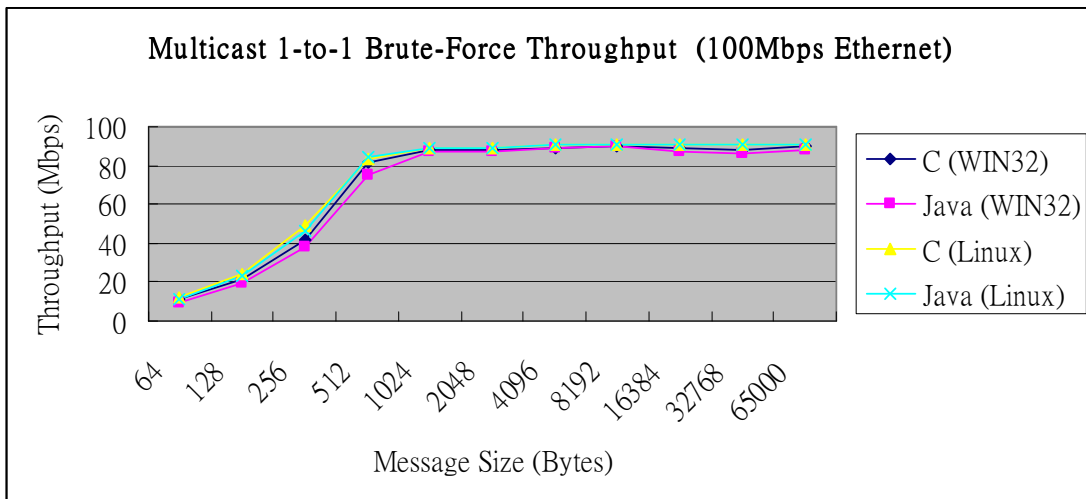
#### (2). Test Objective

This test would show us how fast it could be under a 100Mbps Ethernet environment, and how much difference between C and Java.

### (3). Test Benchmark Figures



**Figure 5-4. UDP UNICAST Throughput on 100Mbps Ethernet Interface**



**Figure 5-5. UDP MULTICAST Throughput on 100Mbps Ethernet Interface**

### (4). Test Analysis

1. Java program could run almost as good as native C program under 100Mbps Ethernet network.
2. Java (WIN32) is a little bit slower than the others while message size is smaller than 1024 bytes.
3. The throughput grows very fast in the range from 256 to 1024 Kbytes.

4. The system reaches the peak performance around the range from 1024 to 2048 Kbytes, and the margin is not noticeable.
5. The actual peak throughput under 100Mbps Ethernet is ~90Mbps.

## **5.4 JMS Performance Benchmark on Loop-back Interface**

This is a throughput test of 1-to-1 transmission on loop-back interface, and there are four JMS systems would be tested: FioranoMQ 2006, SonicMQ v7.0, iBus//MessageBus 5.0, and our FJM2. FioranoMQ and SonicMQ are server and TCP based products, while iBus and FJM2 are multicast and UDP based designs.

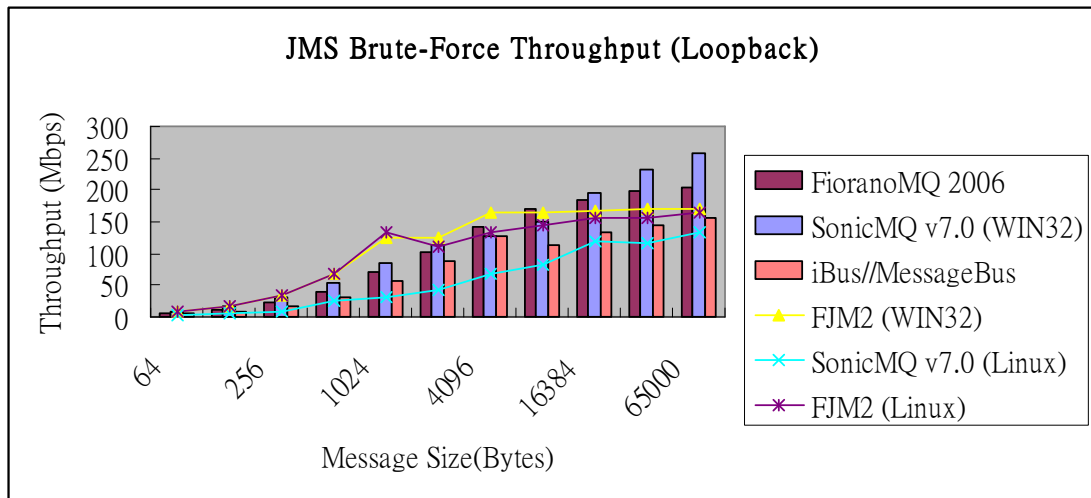
### **(1). Test Scenario**

1. This is a throughput test of 1-to-1 transmission on loop-back interface.
2. We would benchmark four JMS system, two of them (Fiorano, SonicMQ) are server based, and others (iBus, FJM2) are multicast based.
3. Launch the test program, and then follow the parameters to generate a fixed-size message buffer with a randomly generated content.
4. The sender program would try its best to send out the whole message in a single JMS function – publish(...)
5. The receiver program would be notified through the JMS callback function – onMessage(...) to receive message, and then calculate the performance value without message verification.

### **(2). Test Objective**

This test would show us how fast these JMS system could be under a perfect network environment with a unlimited bandwidth and no any message lost.

### (3). Test Benchmark Figure



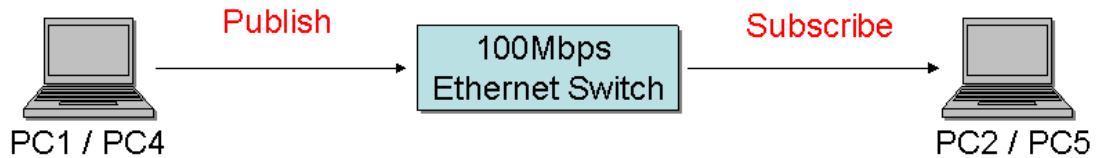
*Figure 5-6. JMS throughput on Loop-back Interface*

### (4). Test Analysis

1. IBM Java Runtime Environment could get much better performance in Windows than in Linux platform.
2. FJM2 outperforms all the other systems while message size is less than 8Kbytes, however its peak performance is only around 170 Mbps, while Fiorano is 202 Mbps and SonicMQ could run up-to 257 Mbps.
3. FJM2 could almost reach its peak throughput while message size equals 4 Kbytes.
4. SonicMQ could get better performance as long as the message grows.
5. FJM2 obviously outperforms iBus//MessageBus which is also a multicast, UDP based design.
6. SonicMQ has the best performance in loop-back test.

## 5.5 JMS Performance Benchmark on 100Mbps Ethernet Interface

### 5.5.1 One-to-One Benchmark on 100Mbps Ethernet Interface



*Figure 5-7. JMS 1-to1 performance test connection topology*

Figure above shows us the connection topology in this test, and there are four JMS systems would be tested: FioranoMQ 2006, SonicMQ v7.0, iBus//MessageBus 5.0, and our FJM2. FioranoMQ and SonicMQ are server based products, while iBus and FJM2 are multicast based.

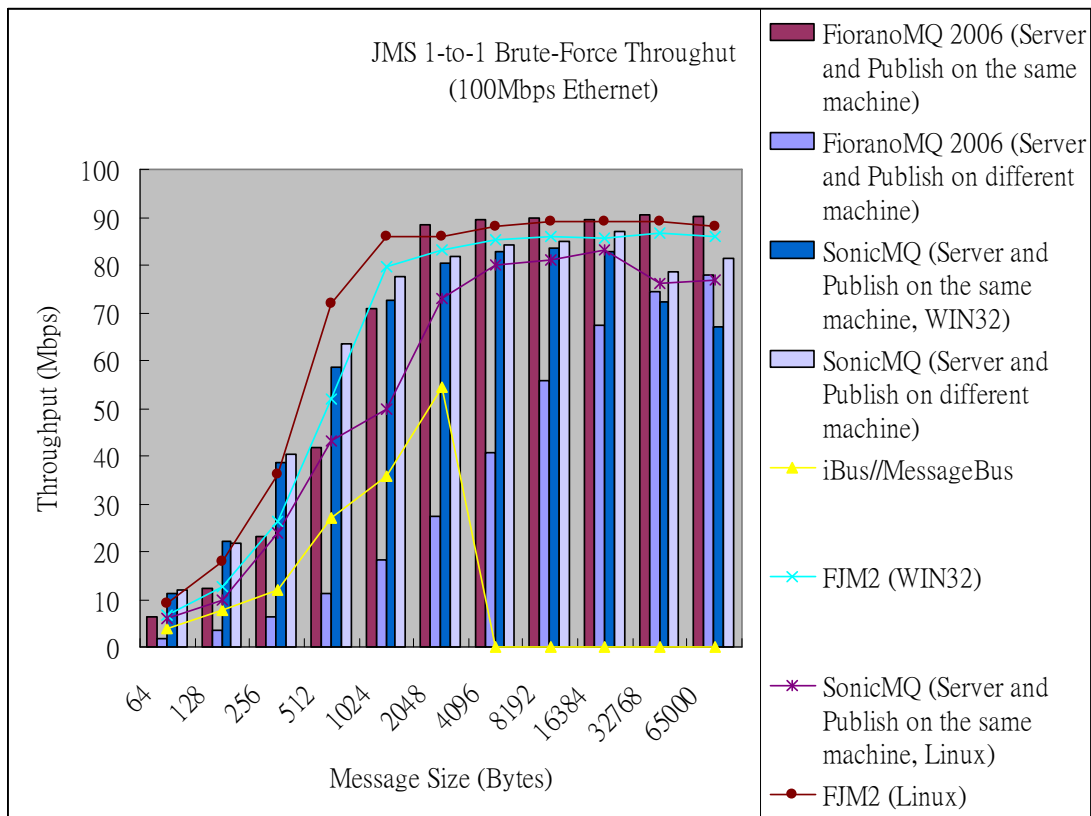
#### (1). Test Scenario

1. Launch the test program, and then follow the parameters to generate a fixed-size message buffer with a randomly generated content.
2. The sender program would try its best to send out the whole message in a single JMS function - `publish(...)`
3. The receiver program would be notified through the JMS callback function – `onMessage(...)` to receive message, and then calculate the performance value without message verification.

#### (2). Test Objective

This test would show us how fast these JMS system could be under a 100Mbps Ethernet network environment for 1-to-2 transmission, and also the performance matrix between several JMS products.

### (3). Test Benchmark Figure

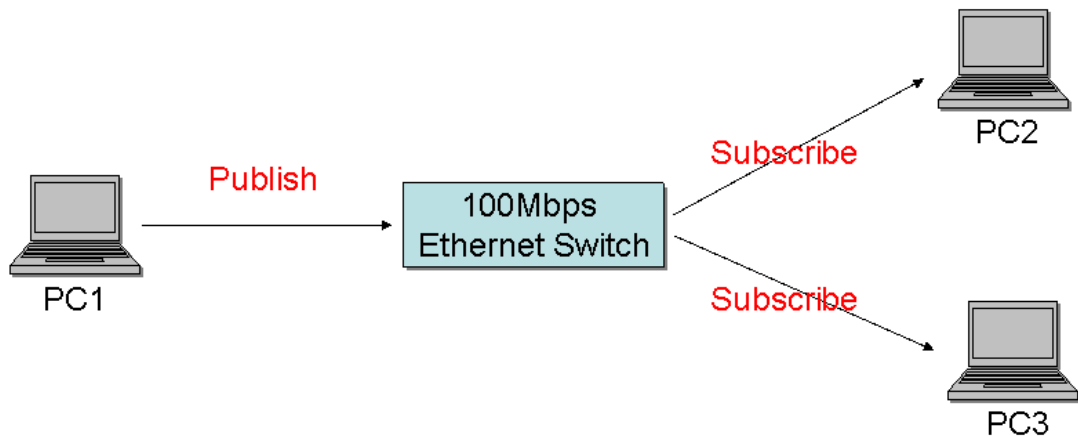


**Figure 5-8. JMS 1-to-1 Throughput on 100Mbps Ethernet**

### (4). Test Analysis

1. The average throughput of FJM2 (Linux) outperforms all the other products.
2. The performance of FioranoMQ would vary with the relative location between publisher program and the server.
3. FJM2 (WIN32) runs almost as good as FioranoMQ (WIN32) while publisher and server deployed at the same machine.
4. There is a serious problem on memory management in iBus//MessageBus, if user application do not limit itself not to send the message too fast, the system would crash while message equals or larger than 4 Kbytes.

## 5.5.2 One-to-Two Benchmark on 100Mbps Ethernet Interface



*Figure 5-9. JMS 1-to-2 performance test connection topology*

Figure above tells us the connection topology in this test, and there are four JMS systems would be tested: FioranoMQ 2006, SonicMQ v7.0, iBus//MessageBus 5.0, and our FJM2. FioranoMQ and SonicMQ are server based products, while iBus and FJM2 are multicast based.

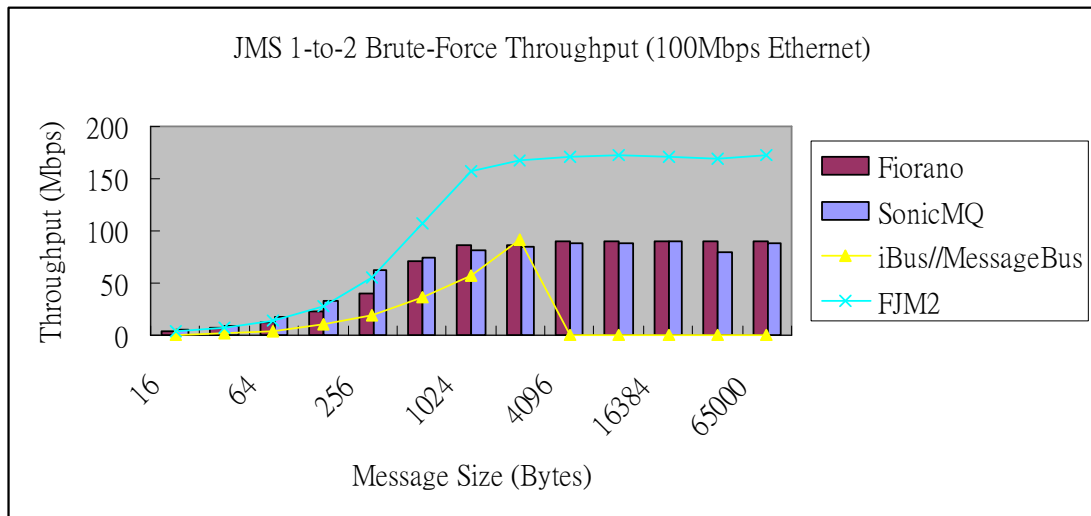
### (1). Test Scenario

1. Launch the test program, and then follow the parameters to generate a fixed-size message buffer with a randomly generated content.
2. The sender program would try its best to send out the whole message in a single JMS function - publish(...)
3. The receiver program would be notified through the JMS callback function - onMessage(...) to receive message, and then calculate the performance value without message verification.

### (2). Test Objective

This test would show us how fast these JMS system could be under a 100Mbps Ethernet network environment for 1-to-2 transmission, and also the performance matrix between several JMS products.

### (3). Test Benchmark Figure



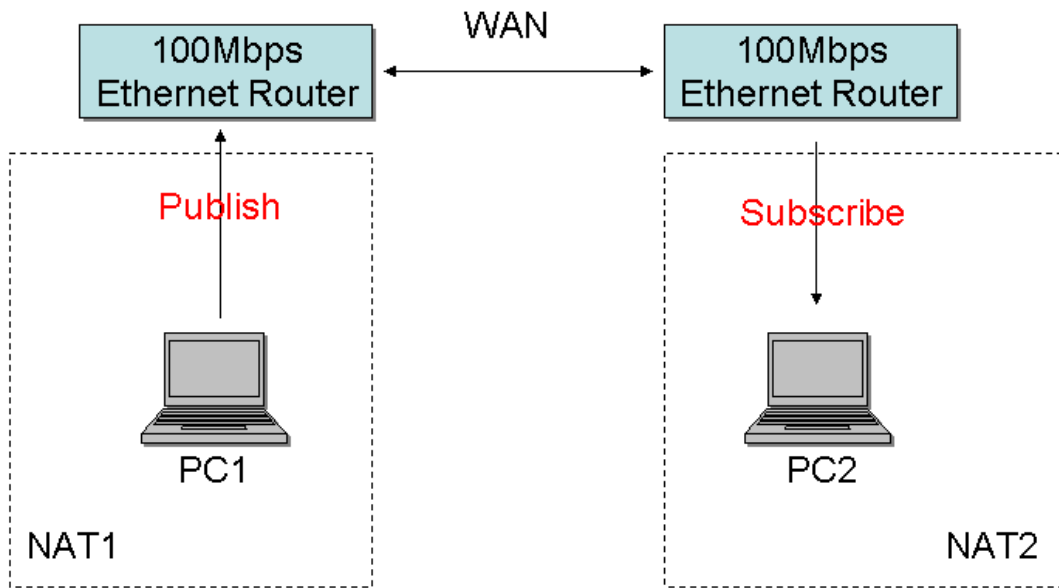
*Figure 5-10. JMS 1-to-2 Throughput on 100Mbps Ethernet*

### (4). Test Analysis

1. FJM2 obviously outperforms all the other JMS system while message size equals or larger than 512 bytes, and more subscribers the difference grows.
2. SonicMQ outperforms all the other JMS systems while message size less than 512 bytes.
3. Although iBus//MessageBus is the only one multicast based JMS system except FJM2, it has a poor performance and serious problem on memory management.



### 5.5.3 1-to-1 Benchmark on 100Mbps Ethernet Interface across WAN



*Figure 5-11. FJM2 1-to-1 WAN performance test connection topology*

Figure above tells us the connection topology in this test, and here we are going to verify the WAN traversal ability of FJM2 system.

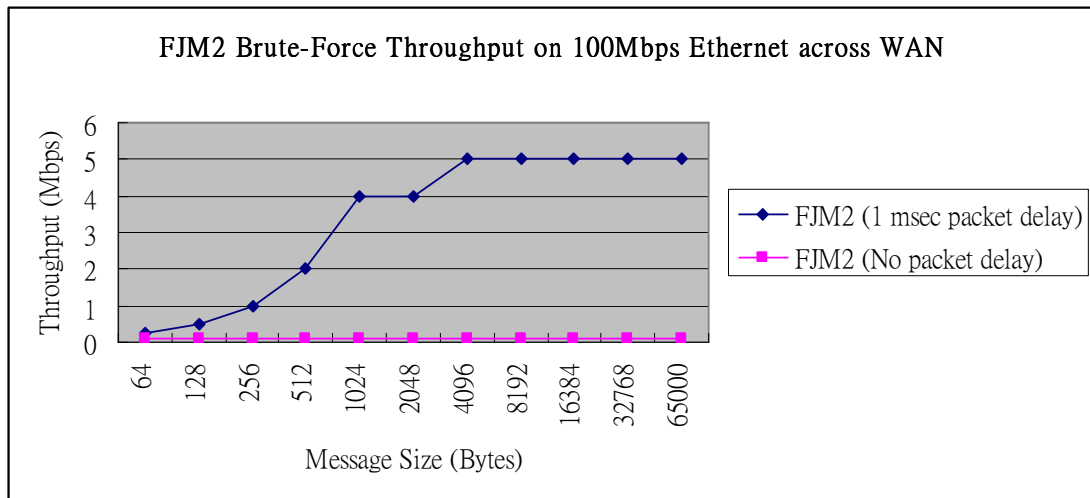
#### (1). Test Scenario

1. Embedded FJM2D into firmware of the router, and launch it.
2. Launch the test program, and then follow the parameters to generate a fixed-size message buffer with a randomly generated content.
3. The sender program would try its best to send out the whole message in a single JMS function - publish(...)
4. The receiver program would be notified through the JMS callback function – onMessage(...) to receive messages, and then calculate the performance value without message verification.

#### (2). Test Objective

This test would show us how fast these JMS system could be under a 100Mbps Ethernet network environment for 1-to-1 transmission across WAN.

### (3). Test Benchmark Figure



*Figure 5-12. FJM2 WAN Throughput on 100Mbps Ethernet*

### (4). Test Analysis

1. The performance of FJM2 across WAN is very poor, we still have many things need to do to improve the performance
2. Obviously now the Ethernet switch on the router now is being overshoot, and thus the performance is so poor.
3. In order to tune the performance well, we have to implement a flow control mechanism in FJM2

## 5.6 Conclusion

FJM2 is a successful multicast based JMS system which shows how much benefits we could gain while the number of receiver grows, and it also acts almost as good as the server based products even in a 1-to-1 transmission, however it still lacks of a flow control to avoid packet over-shoot which leads to the poor performance across WAN.

---

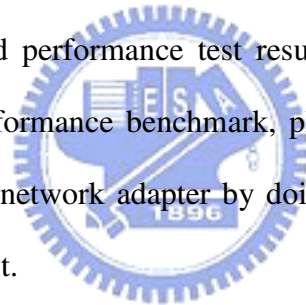
## CHAPTER 6 FJM2 Implementation Issues and Analyses

---

In this chapter, we would discuss some issues discovered in the development process and what's the major factor that limits the system performance.

### 6.1 Choosing a Right Network Adapter

During performance test, I have found that some entry level Ethernet adapter has a bad implementation for multicast in Windows NDIS Driver (Ex: Realtek RTL8651 Fast Family), these driver would consume almost the entire processor resource even when the platform does not join the multicast destination group address, and thus it's impossible to have a good performance test result on these platforms. Therefore before processing any performance benchmark, please make sure that you have a right platform and a right network adapter by doing some basic multicast or UDP UNICAST performance test.



### 6.2 Familiar with your Switch and Router

#### 6.2.1 Turn off or increase the value for Multicast flow control

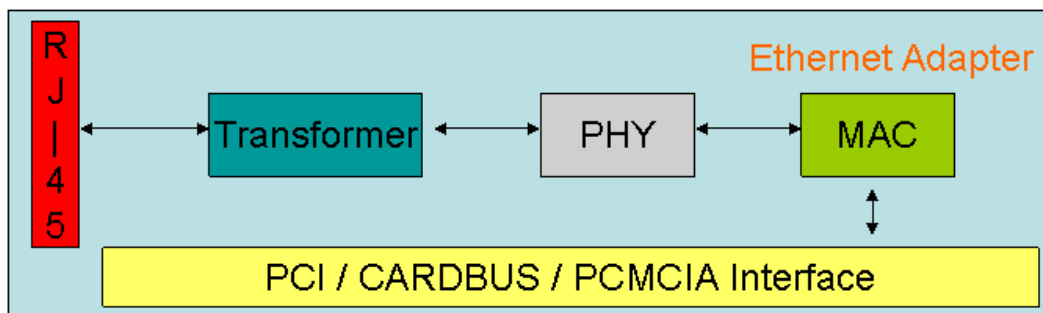
Some Ethernet Switches have the capability to apply bandwidth flow control for multicast, and some of them would turn on this function by default for security considerations. Therefore while deploying multicast based systems, such as FJM2, it would be better to turn off or increase the value of the bandwidth flow control for multicast on the Ethernet Switch, however the way to configure the hardware would differ from vendor to vendor, it it's necessary you might have to contact with hardware vendor for detail.

## 6.2.2 The processor speed limits the tunnel performance

FJM2 provides not only the LAN-to-LAN multicast service, it also provides the capability for WAN traversal ability through the help of FJM2D. However in order to make it works, the FJM2D must be deployed on the edge gateway which usually is a embedded system with a limited system resource. And there are at least two factors would limit the WAN traversal performance.

### (1). Hardware Circuit Design

Before further discussion, we first take a look at a simplified functional block of a Ethernet adapter.



*Figure 6-1. Simplified Ethernet Block Diagram*

The one whom the host driver communicates with is MAC (Media Access Controller) component, and data path between MAC and host driver could be PCI or CARDBUS or PCMCIA or any other proper peripheral interface, in the world of embedded system, everything is a SoC (system-on-a-chip), it means that the processor would contains not only the processor core but also some component, such as Ethernet MAC, and thus the cost reduction could be easily archived and the circuit would be simpler and more efficient.

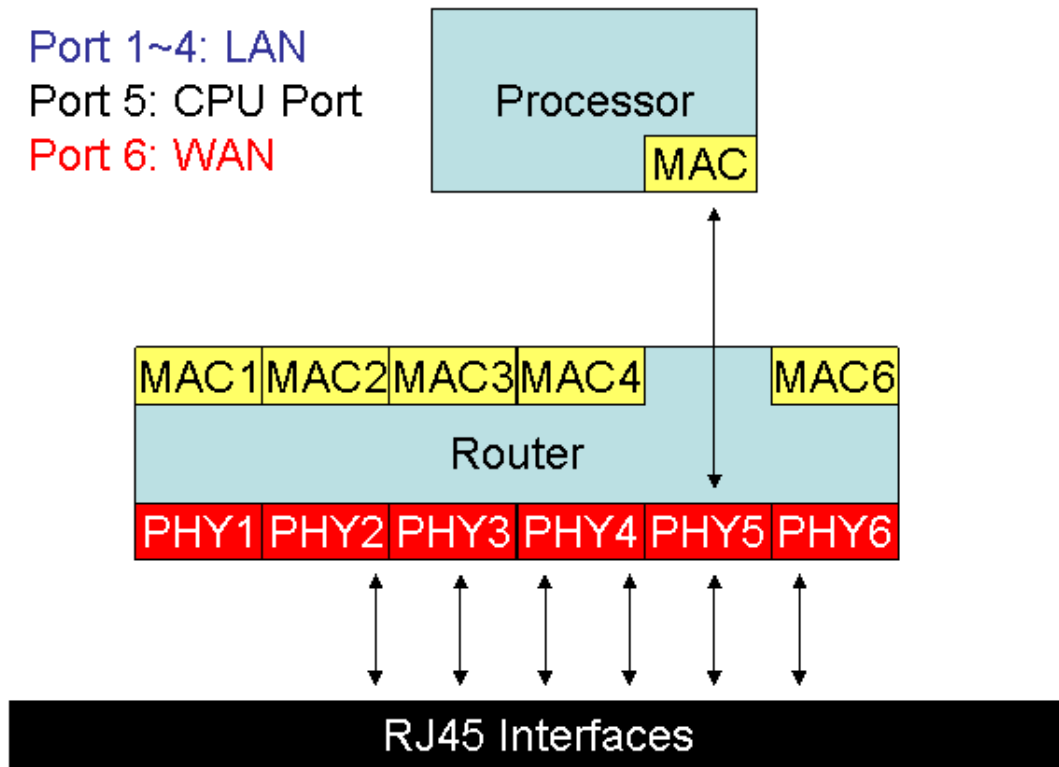


Figure 6-2. Router with only 1 Ethernet MAC

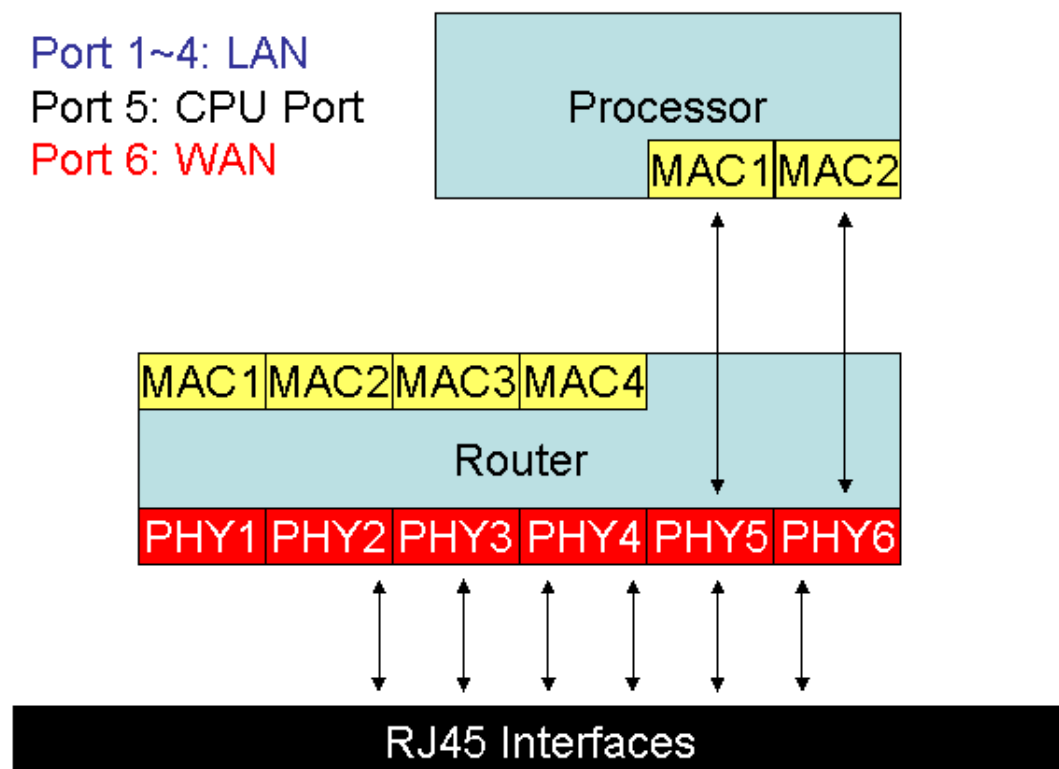


Figure 6-3. Router with 2 Ethernet MAC

Figure 6-2 and 6-3 show us two possible hardware design for a Ethernet router. In figure 6-1, we have only 1 Ethernet MAC inside the router SoC, and the way to distinguish WAN and LAN traffic is through 802.1Q (Tagged VLAN), in other words, internally the host would always receiver and transmit tagged VLAN packets rather than normal Ethernet frame, and it is transparent to all the users connected with it. However since we have only one data path here, in case of 100Mbps Ethernet, performance benchmark could never exceed  $100 / 2 = 50$  Mbps, actually it would even be a little bit slower than 50 Mbps. In figure 6-2, we have 2 separate Ethernet MAC dedicated for LAN and WAN access, we don't need 802.1Q here to distinguish LAN and WAN traffic and thus the software overhead is much lower, and the performance would never be half of bandwidth of the transmission media.

## **(2). Processor Speed**

FJM2D usually would be deployed on the router, and this platform usually has limited processor power and resources. Although FJM2D does not consume much memory, the processor speed would dramatically affect the performance of FJM2D, so choosing a good processor is important.

## **6.3 Memory Allocation Reduction**

In Java, since everything is object, and thus we need a lot of object allocation in the entire system, however the object allocation and de-allocation is very expensive, and many of them could be reused, so that FJM2 would pre-allocated a lot of objects and insert the poll for later use when system startup, every time FJM2 request or reclaim a object from a poll, it just de-queue / en-queue a object from a poll, no any memory allocation / de-allocation would occur, and thus we could get a better system performance.

---

## CHAPTER 7 CONCLUSION

---

Publisher/Subscriber Model is a popular communication model in the world. And most of existing JMS products adapt centralized rather than a distributed architecture and most of them does not use pure Java implementation, they usually deploys a native program for the critical section for performance reason and thus lost the ability of cross-platform. Here we introduction a possible method to implement a de-centralized JMS system base on NORM protocol by pure Java.

From the benchmarks in this paper, we could tell that FJM2 is a successful design that has took a great advantage from multicast to have a dramatic performance improvement when the number of receivers grows, and while compared with centralized design, FJM2 could be even better than centralized products for several times, and it also has a good performance value in 1-to-1 transmission. While compared to the existing multicast based JMS system – iBus, FJM2 is very stable and every publisher or subscriber could dynamically join or leave the topic and the action would not system to waste too much resource. However in iBus, it has not only a memory management issue on rapidly message publishing, it would even crash the topic publisher while any one of the subscribers leaves.

In FJM2, there are still some interesting topics for further improvements:

**(1). Persistent Messaging and Transaction**

FJM2 aims to design a NACK based messaging system, however it's impossible for NACK to detect some system failures. Such as system offline, in such case, positive acknowledge is necessary, although FJM2 based on NACK,

it still needs positive acknowledge to archive persistent messaging and fulfill the JMS specification.

**(2). NORM Flow Control**

In current FJM2, we do not apply any flow control algorithm, and thus the performance curve is not very smooth, the publisher might overshoot frequently and leads to a poor performance across WAN. In order to reduce the number of overshoot, a mature flow control is necessary.

**(3). Fast FEC Algorithm**

FJM2 now uses Compact No-Code FEC algorithm rather than a general FEC algorithm with parity symbols to suppress the number of repair messages and efficiently rewind the repair position. The reason why we Compact No-Code FEC dues to performance consideration, if you could found any fast FEC algorithm, it would be much better if you could put it into FJM2.

**(4). Quality of Service**

The message priority provided in FJM2 merely depends on the threading algorithm of the operating system. It would be much better if we could introduce 802.1p [2][3], Type of Service, or Class of Service into the implementation.

**(5). FJM2 Node Management**

This paper has introduced a administration model for all the nodes in FJM2 system, if necessary you may extends the ability of it to control any you want, such as system resource monitor, daemon startup / shutdown ...etc.

Finally I hope this paper would benefits those who interesting about decentralized, multicast based Java Message System. And if it's necessary, you're welcome to contact with for further discussions. And again thanks for your reading.



---

## BIBLIOGRAPHY

---

1. The Institute of Electrical and Electronics Engineers, Inc., "IEEE P802.11n./D1.0 Draft Amendment to STANDARD [FOR] Information Technology-Telecommunications and information exchange between systems-Local and Metropolitan networks-Specific requirements-Part11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Enhancements for Higher Throughput", March 2006
2. The Institute of Electrical and Electronics Engineers, Inc., "802.1D Media Access Control (MAC) Bridges", June 2004
3. The Institute of Electrical and Electronics Engineers, Inc., "802.1Q Virtual Bridged Local Area Networks", May 2003
4. S. Armstrong, A. Freier, and K. Marzullo, "Multicast Transport Protocol," RFC 1301, February 1992.
5. R. Braudes, S. Zabele, "Requirements for Multicast Protocols", RFC 1458, May 1993.
6. B. Whetten, T. Montgomery, and S. Kaplan, "A High Performance Totally Ordered Multicast Protocol," Proc. of Int'l Workshop on Theory and Practice in Distributed Systems, pp.33-57, 1995.
7. K. Obraczka, "Multicast Transport Protocols: A Survey and Taxonomy," IEEE Communication Magazine, January 1998
8. Sun Microsystems, "Java Message Service", Version 1.0.2, November 1999.
9. Sun Microsystems. "Java Message Service", Version 1.1, April 2002.
10. Ash Rofail, Yasser Shohoud, "Mastering COM and COM+," SYBEX, 1999.

11. Object Management Group, "CORBA: Common Object Request Broker Architecture and Specification," Revision 2.4, October 2000.
12. Sun Microsystems, Java 2 Platform, Enterprise Edition (J2EE)
13. Chuan-Pao Hung, Hsin-Ta Chiao, Yue-Shan Chang, Tsun-Yu Hsiao, Tzu-Han Kao, Shyan-Ming Yuan, "FJM: A Fast Java Message Delivery Mechanism based on IP-Multicast", Third International Conference on Communications in Computing 2002
14. D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Lium, P. Sharma, L. Wei, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification", RFC 2362, June 1998
15. S. Deering, "Host Extensions for IP Multicasting", RFC 1112, August 1989
16. W. Fenner, "Internet Group Management Protocol, Version 2", RFC 2236, November 1997
17. B. Cain, S. Deering, I. Kouvelas, B. Fenner, A. Thyagarajan, "Internet Group Management Protocol, Version 3", RFC 3376, October 2002
18. B. Quinn, K. Almeroth, "IP Multicast Applications: Challenges and Solutions", RFC 3170, September 2001
19. B. Fenner, Ed., D. Meyer, Ed., "Multicast Source Discovery Protocol (MSDP)", RFC 3618, October 2003
20. S. Bhattacharyya, Ed., "An Overview of Source-Specific Multicast (SSM)", RFC 3569, July 2003
21. A. Adams, J. Nicholas, W. Siadak, "Protocol Independent Multicast - Dense Mode (PIM-DM): Protocol Specification (Revised)", RFC 3973, January 2005
22. B. Adamson, C. Bormann, M. Handley, and J. Macker, "Negative-acknowledgment (NACK)-Oriented Reliable Multicast (NORM) Protocol", RFC 3940, November 2004

23. B. Adamson, C. Bormann, M. Handley, and J. Macker, "Negative-Acknowledgment (NACK)-Oriented Reliable Multicast (NORM) Building Blocks", RFC 3941, November 2004
24. M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, J. Crowcroft, "Forward Error Correction (FEC) Building Block", RFC 3452, December 2002
25. M. Luby, L. Vicisano, J. Gemmell, L. Rizzo, M. Handley, J. Crowcroft, "The Use of Forward Error Correction (FEC) in Reliable Multicast", RFC 3453, December 2002
26. M. Luby, L. Vicisano, "Compact Forward Error Correction (FEC) Schemes", RFC 3695, February 2004
27. D. Waitzman, C. Partridge, S. Deering, "Distance Vector Multicast Routing Protocol", RFC 1075, November 1988
28. Irving S. Reed, Gustave Solomon, "Polynomial Codes over Certain Finite Fields", 1960
29. Java FEC Library, <http://onionnetworks.com/developers>
30. SonicMQ, <http://www.sonicsoftware.com>
31. FioranoMQ, <http://www.fiorano.com>
32. Softwired iBus, <http://www.softwired-inc.com>