工 學 碩 士 學 位 論 文

시뮬레이터 기반의 클러스터 메일 서버의 성능 분석

Simulation-Based Performance Analysis of Cluster E-mail Server

2002年 2月

서울大學校 大學院

전기・컴퓨터工學部

李 春 承

초 록

인터넷 사용자가 해마다 늘어나고, 생산과 소비에 대한 동기가 인터넷 메일로 집중됨에 따라 대용량 메일 메시지를 안정적으로 처리할 수 있는 서버들의 집합이 필요했고, 그 대안으로 메일 서버 클러스터링 방법이 사용되고 있다. 더욱이 90년대 들어 시스템의 개발주기가 빨라지고 시스템의 복잡도가 이전에 비해 더욱 심화되고 있는 상황에서 실제 메일 서버의 동작을 검증 및 분석 할 수 있는 메일 서버 시뮬레이터가 필요하다. 이러한 메일 서버 시뮬레이터는 시스템 개발 시간을 단축시킬 수도 있고 운영체제나 시스템의 물리적인 재구성 없이 새로운 알고리즘을 적용하려 하는 시스템을 개발하는데 장점이 있다.

따라서 본 연구논문에서는 SNU PeaCE라고 불리우는 시스템 설계 툴의 DE 도메인을 이용하여 클러스터링 기반의 메일 서버의 성능과 특징을 시뮬레이터의 결과를 이용하여 분석하였고, 그 결과를 실제 시스템과 비교하여 시뮬레이터의 정확도와 메일 서버가 허용할 수 있는 최대의 사용자수를 예상하였다.

키워드 : 클러스터(링), 메일 서버, 시뮬레이션(터), PeaCE
학　번 : 2000-21230

차 례

제 1 장 서 론 ··· 10
1.1 연구 배경 ··· 10
1.2 연구 내용 ··· 12
1.3 논문 구성 ··· 12
제 2 장 메일 시스템 ··· 13
2.1 에이전트(Agencies) ··· 13
2.1.1 MRA (Mail Retrieval Agent) ··· 13
2.1.2 MUA (Mail User Agent) ··· 14
2.1.3 MTA (Mail Transfer Agent) ··· 15
2.1.4 MDA (Mail Delivery Agent) ··· 16
2.2 SMTP (Simple Mail Transfer Protocol) ··· 17
2.2.1 정의 ··· 17
2.2.2 SMTP 모델 ··· 17
2.3 POP3 (Post Office Protocol 3) [3][4] ··· 18
2.3.1 정의 ··· 18
2.3.2 POP3 모델 ··· 18
제 3 장 DE(Discrete Event) 도메인 ··· 20
3.1 DE 도메인의 신택스(Syntax)와 시맨틱(Semantic) ··· 21
3.2 계산모델(Computational Model)로써의 DE 도메인 ··· 22
3.2.1 시간과 분산 시뮬레이터 ··· 22
3.2.2 PeaCE에서의 DE 모델 ··· 22
3.2.3 동시성(Concurrency)과 직렬화(Serialization) ··· 23

제 4 장 시뮬레이터의 설계 및 구현 ········· 26
4.1 설 계 ········· 26
4.1.1 자료 구조 ········· 26
4.1.2 최상위 수준 스키메틱 ········· 34
4.1.2.1 MailFileRead ········· 35
4.1.2.2 MailLVS ········· 36
4.1.2.3 MailAuthDB ········· 37
4.1.3 Mail Node의 설계 ········· 37
4.1.3.1 MailRoute ········· 38
4.1.3.2 MailPOPDaemon ········· 39
4.1.3.3 MailSendmail ········· 39
4.1.3.4 MailMDA ········· 42
4.1.3.5 MailScheduler ········· 42
4.1.4 Mail System의 설계 ········· 47
4.1.4.1 MailNodeThread ········· 48
4.1.4.2 MailOpThread ········· 50
4.1.4.2.1 MailOpThreadFront ········· 51
4.1.4.2.2 MailOpThreadMid ········· 52
4.1.4.2.3 MailOpThreadBack ········· 53
4.1.4.2.4 MailBufferCache ········· 55
4.1.5 MailHub의 설계 (네트워크 모델링) ········· 57
제 5 장 실 험 ········· 59
5.1 실험 환경 ········· 59
5.1.1 메일 시스템 환경 ········· 59
5.1.2 시뮬레이터 환경 ········· 60

5.2 SPECmail2001 ········ 60
5.2.1. 시뮬레이터 입력로드의 정당성 ········ 60
5.2.2 SPECmail 부하 ········ 61
5.2.3 SPECmail QoS (Quality of Service) ········ 61
5.3 실험 측정 내용 ········ 63
5.3.1 메일 시스템 시간 및 네트워크 성능 측정 ········ 63
5.3.2 시뮬레이터 실험 결과 ········ 71
5.3.2.1 단일 메일 응답시간 비교 ········ 71
5.3.2.2 Throughput 측정 ········ 73
5.3.2.3 QoS 측정 ········ 74
5.2.2.4 스케쥴링 정책의 변경 실험 ········ 76
5.2.2.5 네트워크의 영향 실험 ········ 78
제 6 장 결론 및 향후의 과제 ········ 80
참 고 문 헌 ········ 81

그림 차례

그림 1 메일박스로의 접근 ··· 14
그림 2 에이전트를 통한 메일 시스템 ··· 16
그림 3 DE star와 galaxy ··· 22
그림 4 최상위 수준의 시뮬레이터 스키메틱 ··· 34
그림 5 Mail node의 내부 설계 스키메틱 ··· 38
그림 6 DE 스타 ··· 43
그림 7 스케쥴러 테이블 ··· 44
그림 8 Mail system 내부 설계 스키메틱 ··· 47
그림 9 Operation thread 갤럭시의 내부 설계 스키메틱 ··· 50
그림 10 sendmail 처리 시간 ··· 65
그림 11 MDA 메일 전달 시간 ··· 65
그림 12 node thread accept time (1KB～256KB) ··· 66
그림 13 node thread accept time (512KB～4096KB) ··· 66
그림 14 node thread read time (1KB～512KB) ··· 67
그림 15 node thread read time (1024KB～4096KB) ··· 67
그림 16 work list queue insert time (1KB～64KB) ··· 68
그림 17 work list queue insert time (128KB～4096KB) ··· 68
그림 18 work list queue waiting time (1KB～512KB) ··· 69
그림 19 work list queue waiting time (1024KB～4096KB) ··· 69
그림 20 operation thread processing time ··· 70
그림 21 노드별 TCP 데이터 전송 시간 ··· 70
그림 22 시뮬레이터와 실제 시스템간의 응답시간 비교(I) ··· 71

그림 23 시뮬레이터와 실제 시스템간의 응답시간 비교(II) ············ 72
그림 24 노드수가 1,2,4,6일 때 throughput 결과(I) ··························· 73
그림 25 노드수가 1,2,4,6일 때 throughput 결과(II) ························· 74
그림 26 노드수가 1,2,4,6일 때 QoS 결과(I) ····································· 75
그림 27 노드수가 1,2,4,6일 때 QoS 결과(II) ··································· 76
그림 28 스케쥴링 정책에 따른 throughput 비교 ······························ 77
그림 29 스케쥴링 정책에 따른 QoS 비교 ·· 77
그림 30 네트워크의 성능변화와 throughput 변화 ····························· 79
그림 31 네트워크의 성능변화와 QoS 변화 ··· 79

제 1 장 서 론

1.1 연구 배경

90년대 중·후반 들어 국내뿐만 아니라 국제적으로도 인터넷 사용자의 확산과 그에 따른 메일 이용자 수의 폭발적인 증가로 인해 막대한 양의 메일을 안정적으로 처리해야할 필요성이 대두되었다.

한국 인터넷 정보센터의 2001년 12월 현재 자료를 보면,[16] 국내 유효 인터넷 이용자 수는 전체 인구수의 56.6%인 2,438만명으로 그 증가 추세가 과거에 비해 점차 감소하고 있으나 여전히 높은 비중을 차지하고 있고[별첨1], 특히 하루도 빠짐없이 인터넷을 이용하고 있는 사람은 전체 응답자의 62.4%로 매년 증가하고 있으며, 그 이용시간도 주당 12.08시간으로 꾸준히 증가하는 추세이다. 인터넷 사용 주 이용 목적에서도 '메일 사용'이 전체 비율 중 18.9%로 자료정보 검색, 게임에 이어 3위를 차지하고 있을 만큼 보편화되었다.[16] 메일 보유율도 2000년 12월 대비 8.1%증가한 84.4%로, 한달 평균 약 2개의 메일을 이용하는 비율이다.

이러한 메일 사용자의 증가에 비하여 메일 서비스를 처리하는 서버의 성능 향상과 확장성의 능력은 그렇지 못한 현실이다. 특히 작업 부하량이 증가하면 할수록 메일 서버 시스템의 성능상의 문제점은 기하급수적으로 늘어난다. 더구나 요즘에 사용되는 메일들은 과거의 텍스트 기반이 아닌 이미지, 동영상, 사운드 등이 혼합된 형태의 멀티미디어 메일이 점차 많아지고, 크기가 큰 첨부 화일을 포함하는 메일이 다량으로 도착할 경우, 메일 서버의 성능을 일시적, 혹

은 영구적으로 마비시킬 수 있을 정도여서 유지보수가 쉽고 값싸면서 대용량 메일 부하 처리를 얼마나 안정적으로 처리할 수 있는지, 그러한 서비스 수준을 보장하면서 최대 얼마의 사용자를 시스템이 수용할 수 있는 지에 대한 보다 구체적인 메일 서버 설계가 무엇보다 중요하다.

고성능 메일 처리를 위한 시스템으로써 쉽게 접근할 수 있는 방법은 고성능의 단일 컴퓨터를 사용하는 것이지만 초기 구입 비용이 너무 크고, 유지 보수하기도 힘들다. 이는 향후 시스템 성능을 높히려 할 경우 전체 시스템을 새로 구매해야 하기 때문이다. 이에 대한 일반적인 해결책으로 PC 클러스터링 기법이 사용되고 있다. 클러스터링 기술은 여러 컴퓨터 노드들이 동일한 작업을 독립적으로 수행함으로써 유지보수 비용의 단점도 해결하는 동시에 고성능 단일 컴퓨터에 근접하는 성능과 값싼 확장 비용을 이룰 수 있다. 예를 들면, 클러스터링 기술을 적용한 리눅스 서버(리눅스 클러스터)의 경우 슈퍼컴퓨터에 비해 가격은 50분의 1 정도로 저렴하면서 성능은 60%에 달한다. 현재 폭주하고 있는 전자 메일 서버의 설계는 가격 대 성능비가 우수하면서 가용성과 확장성, 안정성적인 측면을 갖춘 PC 클러스터링의 방법으로 이루어지고 있다.

따라서 본 본문에서는 시스템 동작의 정확성을 보장하는 설계 툴을 이용하여 클러스터 기반 메일 시스템이 어떤 특성을 보이며, 성능 향상을 위한 인자가 무엇이며 그들을 어떻게 조합하여야 가장 안정적이며 최대의 성능을 낼 수 있는 메일 시스템을 구성할 수 있는지를 시뮬레이터 수준에서 구현하여 그 성능 분석 측정에 중점을 두고 있다.

1.2 연구 내용

본 연구 논문은 클러스터 기반의 메일 시스템을 통합설계 툴인 SNU PeaCE의 DE 도메인[8, 11]을 이용하여 기술하였으며, 시뮬레이터의 공정한 입력 검증 자료로 SPECmail2001의 로드를 사용하여 검증을 마친 후, 클러스터 메일 시스템이 어떤 특성이 있는지 스케쥴링과 스래드의 개수, 네트워크와 디스크의 상태에 따른 상관관계를 살펴봄으로써 컴퓨터 노드를 증가시켰을 때 시스템이 SPECmail QoS를 만족하면서 최대 성능과 사용자수를 시뮬레이션 기반에서 제시하고자 하는데 있다.

1.3 논문 구성

논문의 구성은 클러스터 메일 시뮬레이터의 구현을 위한 先배경지식으로 메일 시스템의 기본적인 개념을 서술하고, 통합 설계 툴인 PeaCE의 개념을 DE 도메인을 위주로 설명한 다음, 구현한 시뮬레이터의 구조와 성능 실험, 결과 분석 순서로 기술했다.

제 2 장 메일 시스템

논문에 기술된 내용을 이해하기 위해 본 제2장에서는 메일 시스템을 구성하고 있는 에이전트들과 기본적인 통신 프로토콜의 종류에 대해서 기본적인 동작과 함께 설명하겠다.

2.1 에이전트(Agencies)

다음에 설명된 내용은 메일 시스템을 개념적으로 이해하는데 중요한 구성요소이다. 크게 MRA, MUA, MTA, MDA와 같이 4부분으로 나눠지고 각각은 서로 연관되어 동작된다.

2.1.1 MRA (Mail Retrieval Agent)

사용자가 네트워크을 통해 떨어진 메일 박스로부터 메일을 검색하는 서비스 프로그램이다. 사용자들은 이 프로그램을 통해 메일 시스템에 접속한 후, 자신의 메일박스를 직접 검색 할 수 있다. 검색한 메시지는 TCP 110번 포트를 거쳐 사용자가 사용하고 있는 로컬머신으로 저장되게 된다.

일반적으로 MRA는 클라이언트/서버 방식으로 작동되며 클라이언트 프로그램은 MUA로 통합된다. 인터넷에서는 MRA를 위한 두 가지 프로토콜이 존재하는데 현재 가장 많이 쓰이고 있는 것으로 POP3(Post Office Protocol version 3)와 IMAP4(Internet Message Access Protocol version 4)가 있다.[5]

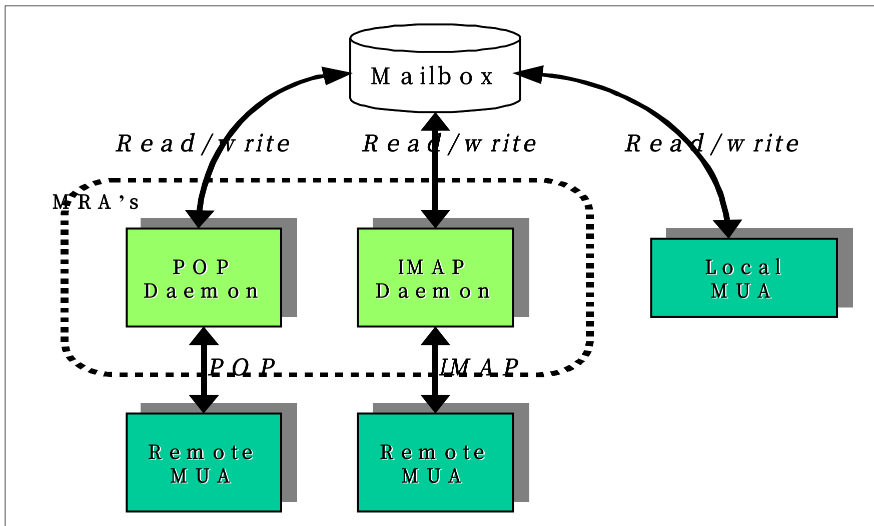


그림 1. 메일박스로의 접근

2.1.2 MUA (Mail User Agent)

MUA는 사용자에게 가장 직접적으로 보여지는 메일 시스템 부분으로 메일을 보내거나 자신에게 메일이 도착했을 때, 사용자에 의해서 사용되는 클라이언트 프로그램이다. 예를 들면, 마이크로 소프트사의 outlook express같은 프로그램을 말한다. 일단 메일이 자신의 메일박스에 도착하면 MUA는 메일박스에 직접 혹은 네트워크를 통한 MRA를 이용해 접근하여 자신의 메일박스로 수신된 메일을 로컬저장 장치로 옮겨 올 수 있게 할뿐만 아니라, 사용자가 메일 프로그램을 조작할 수 있는 모든 기능을 제공한다.[4]

2.1.3 MTA (Mail Transfer Agent)

MTA는 메일을 직접 전송하지 않고, 메일 메시지에 대해서 어디로 전송할 것인지 결정하고, 실제적인 데이터의 전송을 위해 메시지를 MDA에게 넘겨주는 역할을 한다. 좀 더 자세하게 말하면 MTA는 보내고자 하는 메일에 대해서 최종적인 목적지를 결정하지 않고, 최종 목적지로까지 전달하기 위해 거쳐야할 다음 MTA를 명시할 뿐이다. 따라서 MTA는 보내고자 하는 메일을 전송할 MTA의 형식에 알맞게 변형을 시킨다. 대표적인 프로그램으로 sendmail[10]과 qmail이 많이 사용된다. sendmail은 Unix나 Linux에서 /usr/lib/에 위치하고 있으며, TCP 포트 25을 차지하고 데몬으로 동작한다. /var/spool/mail은 수신된 메일의 임시 메일박스로 한 명의 사용자에 대해서 하나의 파일로 관리되고 있으며, 보내고자 하는 메일이 있을 경우 메일 헤더와 본문을 쪼개서 /var/spool/mqueue에 저장한다.

하나의 메일이 25번 포트로 들어오면 sendmail 데몬은 포트를 감시하고 있다가 TCP 커넥션을 유지하는 프로세스를 fork한다. 커넥션이 유효하게되면 수신자가 해당 서버에 있는지를 검사하는 프로세스가 fork되며, 그 결과 응락이 수락되면 들어오면 메일 본문을 저장할 프로세스가 역시 fork되어 위에서 설명한 /var/spool/mail에 저장하고, 마지막으로 임시로 저장된 메일을 실제로 사용자 메일박스에 저장시키는 일을 담당하는 MDA로 쓰일 프로세스가 fork되어 처리된다. 결과적으로 하나의 메일이 수신되기 위해서 4번의 fork가 일어나고 그 과정에서 생성된 4개의 프로세스들은 메일이 완전히 메일박스에 저장될 때까지 유지되며, 저장이 완료되고 나면 위에서 기술한 수신순서와 반대의 순서로 생성된 프로세스들이 종료하게

된다.

2.1.4 MDA (Mail Delivery Agent)

사용자가 가지고 있는 메일박스에 메일을 기록하기 위하여 MTA에 의해서 fork()로 호출되어 사용되는 작은 프로그램이다. MDA가 메일 메시지를 받았을 때, MTA의해 자신의 로컬 머신에 기록하던지 아니면 다른 MTA로 보내줘야 한다. 따라서 메일을 다루는데 실제적인 책임을 지고 있으며, 그 과정중에 여러 개의 MDA가 존재하고 그들은 메일 메시지 하나당 처리를 위해 MTA에 의해 fork된다.

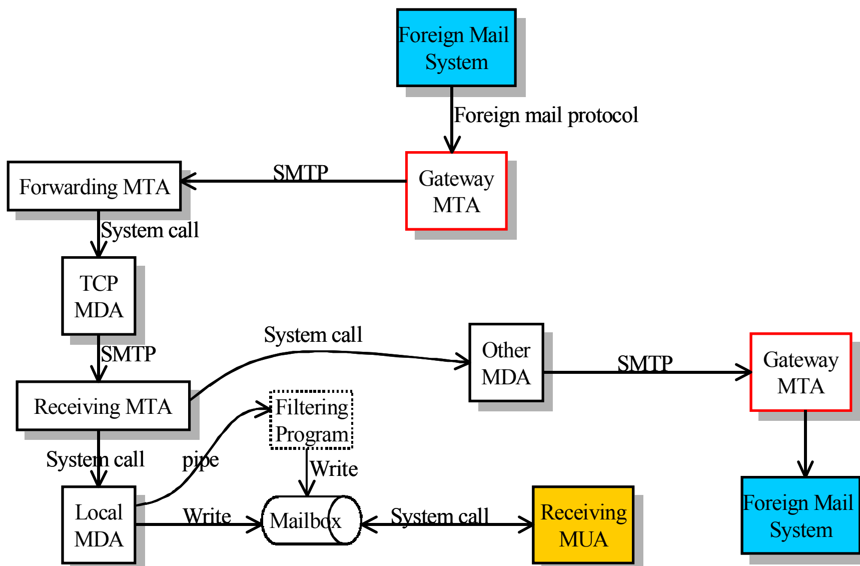


그림 2. 에이전트를 통한 메일 시스템

본 연구에서 구현된 시뮬레이터는 실제로 이들 에이전트들 중

MTA와 MDA에 대한 동작을 시뮬레이터 상에서 하나의 기능 블럭 (functional block)으로 작성하여 그들의 관계를 명세 하였다.

2.2 SMTP (Simple Mail Transfer Protocol)

2.2.1 정의

SMTP는 간단한 메일 전송 규약으로, 메일을 보내기를 희망하는 클라이언트들과 보내진 메일을 받아들이는 서버사이에서 사용되어지는 통신 프로토콜로 RFC821에 정의되어 있다.[2] 프로토콜을 이루고 있는 코드는 US-ASCII로 한정되고, 두 컴퓨터가 상호 접속을 어떤 방법으로 하는지에 대한 규정과 메일을 전송하기 위해 사용하는 제어 명령으로 이루어져 있다. TCP/IP 환경에서 메일 응용 프로토콜로 사용되고 있으며, 주로 유닉스 시스템간의 전자 우편용으로 널리 사용되고 있다.

SMTP에서 사용하는 명령어는 클라이언트가 서버에게 보내는 명령어로써, 서버는 사람이 읽을 수 있는 문자열로써 응답하거나, 그 내용이 미리 약속된 숫자화된 응답 코드를 클라이언트에게 넘겨준다.

2.2.2 SMTP 모델

3.1.3의 sendmail의 메일 송・수신을 추상적으로 설명한 단원으로 두 기계사이에 SMTP 통신을 완성하기 위해 각 기계에 프로세스가 하나씩 존재하는데 메일을 보내는 쪽은 송신자 프로세스(Sender

SMTP Process)라 부르고, 수신하는 쪽은 수신자 프로세스(Receive SMTP Process)라고 한다. 송신자 SMTP는 메일 전송 서비스 연결을 초기화하며 응답들을 수신하고 메일의 전송을 담당한다. 이에 반하여, 수신자 SMTP 프로세스는 송신자로부터의 연결이 설정되기를 기다린다. 이것은 송신자 SMTP로부터 SMTP 명령을 수신하고, 응답들을 반송하며, 메일 수신에 관한 일련의 동작을 수행한다.

사용자가 메일을 전송하기 위해 요청을 보내면, 그 요청의 결과로써 송신자 SMTP는 수신자 SMTP에 대해서 양방향 전송 채널을 설정한다. 수신자 SMTP는 송신자 SMTP의 요청에 의해 생성되고 SMTP의 응답은 명령에 응답하여 전송자 SMTP에게 되돌려 진다.

2.3 POP3 (Post Office Protocol 3) [3][4]

2.3.1 정의

POP3은 인터넷에서 주기적 혹은 비주기적으로 메일박스가 있는 메일서버에 접속하여 저장된 메일을 조회하거나 자신의 컴퓨터로 가져옴으로써 서버의 내용을 삭제하는 프로토콜이다. POP3 클라이언트는 매 번 POP3 세션이 일어날 때마다 초기화되고, POP3 서버는 단지 클라이언트가 접속을 요청하기만을 기다린다.

2.3.2 POP3 모델

POP3은 동작 특성과 관련하여 인증상태, 처리상태, 수정상태등

세 가지의 상태로 나누어 설명될 수 있다.

인증상태 (Authorization State)는 한 클라이언트가 자신의 메일박스가 있는 POP3 서버로 접근하기 위하여 반드시 거쳐야 되는 상태로 리눅스에서 기본적으로 사용하고 있는 ipop3d 프로그램은 /usr/sbin/아래에 있다. 가장 기본적인 인증 방법은 사용자 이름과 비밀번호를 이용하는 방법인데, 리눅스의 경우 /etc/passwd과 /etc/shadow에 명시되어 있다.

처리상태 (Transaction State)는 일단 사용자에 대한 인증 과정이 종료가 되면 메일박스를 조작할 수 있게 허용하는 단계로, 메일박스에 있는 메시지들에 대해서 읽거나 다른 메일박스로 그것들을 옮기는 작업들을 수행한다.

수정상태 (Update state)는 메일박스로부터 처리상태동 DELE 명령이 사용돼 삭제 표시된 메시지를 실제적으로 지우는 상태이다. 이 상태는 오직 삭제 명령이 표시된 메시지가 있는 경우에만 동작되며, 만일 메시지가 삭제 표시가 되었는데 RSET 명령에 의해 리셋 되었으면 수정상태로 들어가지 않는다.

제 3 장 DE(Discrete Event) 도메인

2장에서 서술된 메일 시스템을 시뮬레이터로 구현하여 동작시키기 위한 시뮬레이션 환경이 반드시 필요하다. 일반적으로 클러스터 기반 메일 시스템과 같은 분산형 시스템을 명세할 때 주로 사용되는 환경이 분산 혹은 이산 사건 시뮬레이션 환경이고 [17], 대표적으로 사용되는 툴이 Ptolemy[9]의 discrete event(DE) domain이다.

본 논문에서는 Ptolemy의 discrete domain을 사용하고 있는 PeaCE를 이용해 구현하였다. PeaCE는 Ptolemy extension as Codesign Environment의 약자로 서울대학교 전기·컴퓨터공학부 통합설계 및 병렬처리(CAP, Codesign And Parallel processing) 연구실에서 개발하고 있는 이산적인 계산 모델의 시뮬레이션을 위한 통합설계 환경이다. PeaCE는 1990년부터 개발된 Ptolemy 프로그램을 기반으로 했으며, 본 연구 논문에서 언급하고 있는 PeaCE DE 도메인은 Ptolemy DE 도메인과 동일한 개념을 가지고 있다.

DE 도메인은 PeaCE안에 있는 여러 도메인 중 하나로 시뮬레이션을 위한 엔진이 구현되어 있는 부분으로 개발자는 메일 시스템을 구성할 요소들을 기술하고 그들에 대한 관계를 명시해줌으로써 수행되어야 할 동작과 수행될 순서를 정할 수 있다. 본 장에서는 시뮬레이터 디자인의 스키메틱(schematic)을 이해하기 위해 PeaCE에서 정의된 용어와 계산모델로써의 DE 도메인에 대하여 간략히 설명하였다.

3.1 DE 도메인의 신택스(Syntax)와 시맨틱(Semantic)

DE 도메인에서 사용되고 있는 블럭은 시스템에서 명세하고자 하는 시스템내의 특정 부분을 구분하여 프로그래머가 실제로 작성한 코드가 들어 있는 것으로 '스타(star)'라고 불리고, 이 스타는 더 이상 기능상으로 쪼개질 수 없는 블럭(atomic block)이다. 또한 하나 이상의 스타들이 모여 또 하나의 계산 블럭을 만들 수도 있는 데 이것을 '갤럭시(galaxy)'라고 한다. 갤럭시는 다른 갤럭시와 스타들의 조합으로 이루어져 있기 때문에 시스템 명세를 계층적으로 표현할 수 있게 한다.

동작에 대한 블럭을 만든 후 블럭간에 발생하는 관계성은 그 둘을 연결한 '아크(arc)'를 어떻게 연결하는가에 달려있다. 특히, DE 모델에서는 하나의 아크는 정해진 시간에 발생한 이벤트[1]들의 집합이다. 그림3은 하나의 스타에서 생성된 이벤트들이 어떻게 스타와 갤럭시 사이에서 전달되는 지를 보여주고 있다.

---

1) 이벤트(Event, 혹은 파티클 Particle) : 이벤트 시뮬레이션 상에서 흔히 일컬어지는 말로, 실제 메일 시스템에서 사용되는 하나의 메일 수신은 시뮬레이터 상에서 하나의 이벤트 혹은 파티클로써 표현된다.

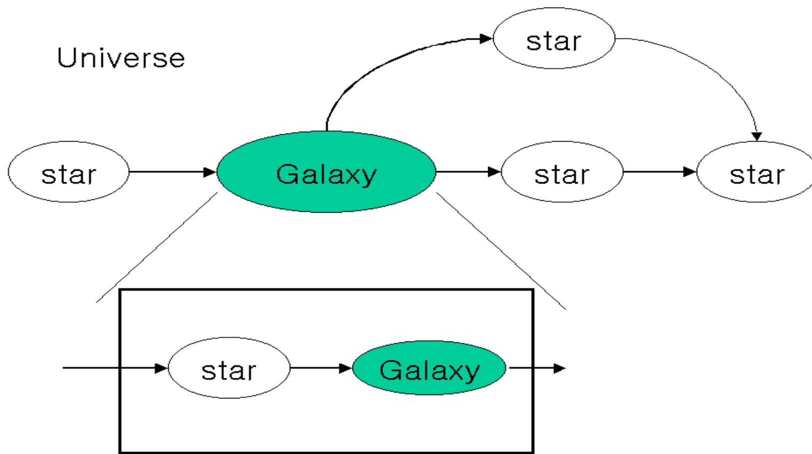


그림 3. DE star 와 galaxy

3.2 계산모델(Computational Model)로써의 DE 도메인

3.2.1 시간과 분산 시뮬레이터

시간은 동시성(concurrency)를 표현하는 자연스런 개념적 모델을 제공한다[7]. 특히 클러스터 상에서 구현된 시간에 대한 이산적인 동작을 가지는 분산형 시스템을 모델링 하기에 적합하며, 특히 병렬 이산 이벤트 시뮬레이터를 제작하는데는 상당히 긍정적이다.

3.2.2 PeaCE에서의 DE 모델

Ptolemy에서 처음으로 제안한 DE 도메인은 큐잉 네트워크, 통신 네트워크, 상위수준의 컴퓨터 아키텍쳐를 시간 중심으로 시뮬레이션 하기 위한 환경을 제공한다.[8]

하나의 DE 블럭안으로 들어온 이벤트는 출력 이벤트를 생성하면서 임의의 시간 지연을 갖을 수 있고, 그렇지 않을 수도 있다. 각 이벤트들이 스타나 갤럭시 사이에서 전달되기 위해서는 각 이벤트마다 자신이 수행되야 되는 시간을 가지고 있는 타임 스템프(time-stamp)를 사용한다. DE는 위와 같은 이벤트에 대한 시간 관리를 위해 두 개의 double형을 가진 멤버변수들을 제공하는데 도착시간과 완료시간이다. 그 중 DE 블럭 안으로 이벤트가 들어오는 시간이 도착시간이고, 블럭내의 코드를 수행하고 출력 이벤트가 발생되는 시점의 시간이 완료시간이다.

또한 DE 스타는 시간 운용에 있어서 두 개의 부류로 구분될 수 있기도 하는데, 기능(functional)스타와 지연(delay)스타이다. 입력 이벤트와 출력 이벤트의 시간 지연이 없는 블럭 즉, 완료시간과 도착시간이 똑같은 시간을 갖는 스타가 기능스타이고, 완료시간과 도착시간 사이에서 임의의 지연시간을 갖는 블럭이 지연스타이다. 결국 어떤 블럭이 기능적인지 지연 시간적인지를 구성하는 것은 전적으로 시스템을 설계하는 디자이너가 결정해야 될 사항이고, 이들 스타에 대한 동작순서는 내부적으로는 전역 스케쥴러(Global Scheduler) 안에 있는 단일 전역 이벤트 큐(Single Global Event Queue)내에서 시간 순으로 정렬 저장되어 관리되며, 다음에 수행될 이벤트는 큐안에 저장되어 있는 이벤트들 중 가장 빠른 시간을 갖는 이벤트 하나가 인출되어 수행된다.

3.2.3 동시성(Concurrency)과 직렬화(Serialization)

위에서 우리는 DE 블럭이 시간을 다루는데 있어서 기능스타와 지

연스타로 구분됨을 보았다. 하지만 DE 도메인 자체는 기본적으로 시간상 이벤트의 동시 수행을 전제하고 있기 때문에 둘 이상의 블럭에서 동시 혹은 앞선 이벤트가 미쳐 완료되기 전에 또 다른 이벤트가 입력으로 들어왔을 경우, 그 블럭에서의 지연시간이 두 입력 이벤트의 도착시간에 그대로 더해지면 수행을 마친 완료시간은 임의의 시간에 동시에 실행(concurrent execution)되었다고 말한다.

두 입력 이벤트 A, B가 블럭 안에서 수행되는데 걸리는 지연시간 $serviceTime_A$와 $serviceTime_B$를 갖는다고 가정하면, A의 완료시간 T_A는 입력시간 $arrivalTime_A$에 서비스 지연 시간을 더한 시간이 되며, T_B의 경우도 아래와 같이 나타낼 수 있다.

$$T_A = arrivalTime_A + serviceTime_A$$

$$T_B = arrivalTime_B + serviceTime_B$$

만일 이벤트 B가 이벤트 A가 끝나기 이전에 들어왔다고 했을 때, 이벤트 B는 $T_A - arrivalTime_B$ 만큼의 시간이 중복되어 동시에 수행된 결과와 같다. 따라서 DE 도메인 내에서 하나의 하드웨어 블럭 디바이스를 모델 하고자 하는 경우, 입력으로 들어오는 이벤트들에 대해서 반드시 직렬화(serialization)시켜야 하며, 완료시간 계산은 다음과 같이 변경해야 한다.

$$\text{if}(\, arrivalTime_B > T_A \,) \quad T_B = arrivalTime_B + serviceTime_B$$

$$else \quad T_B = T_A + serviceTime_B$$

즉 , B 이벤트가 블럭에 들어오는 시각에 A 이벤트가 종료된 시각보다 늦은 시각에 들어오면 B의 도착시간에 서비스 지연시간을 더해주고, 그렇지 않고 종료되기 전에 들어온 경우는 이벤트 A가 종료된 다음에 수행 되야 하기 때문에 $arrivalTime_B$를 T_A로 변경하여야 된다.

제 4 장 시뮬레이터의 설계 및 구현

4.1 설 계

시스템에 대한 설계는 가장 최상위 레벨을 기준으로 계층적으로 정의하였다. DE 스타 블럭의 이름은 스타 블럭 하단에 굵은 글씨체로 씌여져 있고, 아크의 연결은 입출력 포트 홀을 통해 이루어진다.

4.1.1 자료 구조

시뮬레이터 내에 많은 자료 구조가 정의되어 있지만 가장 대표적으로 사용되는 자료 구조를 기술하였다.

① typedef struct MailType

```
typedef struct MailType {
        double LocalTime;
        unsigned int Opcode;
        unsigned int Recipients;
        char RecipientList[LISTSIZE];
        char SMTP_POP[LISTSIZE];
} MailType;
```

입력 파일에서 읽어들인 내용을 시뮬레이터 안에서 하나의 이벤트로써 동작하기 위해 만들어지는 구조체이다.

LocalTime

milli second단위 메일 이벤트 도착시간으로 실제 메일 시스템 주변에서 흔히 볼 수 있는 메일 도착 시간이다.

Opcode

이벤트가 SMTP인지 POP인지를 구분하는 필드로써 SMTP의 경우 1, POP의 경우 2의 값을 취한다.

Recipients

메일의 수신자의 수, 단일 수신자이면 1이다.

RecipientList

다중 수신자의 모든 이름이 하나의 문자열로써 저장되어 있고, 수신자 각각은 세미콜론으로 구분되어 있다.

SMTP_POP

문자열의 타입을 가지며, SMTP의 경우 메일 이벤트의 크기를 Byte 단위로 나타내고 있으며, POP 이벤트의 경우는 pop을 하기 위한 메일박스내의 메일 번호를 나타내고 있다.

② typedef struct MESSAGE_ID

```
typedef struct MESSAGE_ID {
        int NodeNumber;
        int MailIndex;
} MESSAGE_ID;
```

메일 이벤트가 시뮬레이터 안에서 다른 이벤트들과 구분하기 위해 사용되는 아이디로 노드 번호와 메일 이벤트 번호의 조합으로 이뤄져 있다. 메일 시스템내의 실제 메시지 아이디는 메일이 sendmail에 도착하면 도착 됐을 때의 날짜와 수신자의 조합으로 이뤄져 있지만 시뮬레이션 수준에서의 모델은 그렇게 자세히 표현할 필요는 없고

단지 시뮬레이터 동작 과정 중에서 다른 이벤트와 구분할 수 있을 정도의 수준이면 된다.

NodeNumber

해당 메일 커넥션을 처음으로 받은 노드의 번호를 나타내고 있으며, LVS에서 라운드 로빈에 의해 결정된다. 이후 해당 이벤트가 로컬에서 처리되지 않고, 리모트로 전송될때도 그대로 유지하여, 다른 노드에서 동일한 아이디를 갖는 이벤트와의 충돌을 피할 수 있게 하였다.

MailIndex

메일 이벤트를 수신했을 때 주어지는 값으로 SMTP 이벤트와 POP 이벤트에 대해서 순차적으로 할당된다.

③ typdef struct MailTypeExtension

```
typedef struct MailTypeExtension {
        struct MESSAGE_ID MessageID;
        double LocalTime;
        unsigned int Opcode;
        unsigned int Recipients;
        char RecipientList[LISTSIZE];
        char SMTP_POP[LISTSIZE];
        float ArrivalTime;
        float Delay;
} MailTypeExtension;
```

MailTyepExtension은 시뮬레이터 안에서 이벤트에 대한 동작을 좀 더 유용하게 사용하기 위해 MailType의 확장된 형태의 구조체이다. DEMailSendmail 스타에서 생성되며, MailType의 필드와 이름이 같은 것은 MailType의 내용이 그대로 들어있다.

LocalTime, Opcode, Recipients, RecipientList, SMTP_POP

MailType의 내용과 동일

ArrivalTime

메일 이벤트가 스케쥴됐을 때, 해당 스타 내에 도착한 시간을 표현한 필드이다. *LocalTime*과 다른 점은 *LocalTime*은 이벤트가 메일 시스템 내로 처음 들어온 시간은 나타내고 있어 시뮬레이터 수행 중간에 변하지 않는 값인데 반해, *ArrivalTime*은 다음 스타로 넘어갈 때마다 변하는 값이다.

Delay

이벤트가 한 스타에서 다음 스타로 넘어가기 위해 현재 스타에서 수행 돼야할 시간이다. 이 시간은 스케쥴러 안에서 소모가 되고, 다른 스타 내에서 다시 리셋되어 스케쥴된다. 시뮬레이터가 얼마나 정확하게 동작하는가는 이 지연시간 값을 어떻게 실제 시스템에 가깝게 근사 시킬 수 있는 지에 달려 있다.

④ typedef struct MailTypeTag

```
typedef struct MailTypeTag {
        char Tag[15];
        struct MailTypeExtension Payload;
} MailTypeTag;
```

메일 이벤트의 시뮬레이션 내에서의 동작을 나타내기 위해 붙여진 특성 필드이다.

Tag

이벤트가 어떤 특성을 가지고 있는지를 나타내는 것으로, 시뮬

레이터 안에서 총 5가지가 있다. 이 태그는 node thread에서 부여한다. 태그의 종류는 다음과 같다.

Unknown
해당 이벤트에 대해서 인증을 필요로 함을 나타냄.

Local
인증 후 수신자의 메일박스 위치가 현재의 노드일 경우를 나타내고 *Unknown*이 변경된 것이다.

Remote
인증 후 수신자의 메일박스 위치가 현재의 노드가 아닐 경우, 즉 네트워크를 통해서 다른 노드로 전송을 해야 함을 나타내고, 전송후 Reply를 기다리고 있다.

Forwarding
송신 노드의 *Remote*와 수신 노드의 *Remote* 이벤트를 구분하기 위해 이벤트가 네트워크를 지나는 동안 DEMailHub에서 변경된 것으로, 송신자의 입장에서는 *Remote*이지만 수신자 입장에서는 *Forwarding*이기 때문이다. *Local*과 *Forwarding* 이벤트의 수신자 메일 박스는 모두 수신한 노드가 가지고 있다.

Reply
Forwarding 이벤트를 받았을 경우, 전송한 노드로 수신이 잘 완료되었다는 것을 알려주기 위해 전송한 노드로 다시 되돌려 보내는 이벤트로써 acknowledge 역할을 하는 이벤트이다.

Payload
MailTypeExtension 타입을 가지는 구조체를 담고 있다.

⑤ typdef struct MailTypeHub

```
typedef struct MailTypeHub {
        int sourcePort;
        int destPort;
        struct MailTypeTag TagPayload;
} MailTypeHub;
```

네트워크 수행에 관한 정보를 담고 있는 구조체로 이벤트 발송 노드 번호와 수신 노드 번호를 가지고 있다.

sourcePort
송신 노드의 노드 번호를 가지고 있다.

destPort
수신 노드의 노드 번호를 가지고 있어, DEMailHub 스타가 *Remote* 혹은 *Reply* 이벤트에 대해 올바로 라우팅을 할 수 있도록 하기 위한 필드이다.

⑥ typdef struct MailTokenOut

```
typedef struct MailTokenOut {
        struct MESSAGE_ID MessageID;
        char RecipientName[ID_SIZE];
        float ArrivalTime;
        float Delay;
} MailTokenOut;
```

각 스타에서 스케쥴러로 이벤트를 처리하기 위해 보내는 토큰 정보로, 스케쥴러는 스케쥴러 테이블을 구성하여 스케쥴링 한다.

MessageID, Delay
위와 동일

RecipientName
수신자의 이름이 들어 있다.

ArrivalTime
이벤트가 해당 스타에 도착한 시간을 가지고 있다.

⑦ typdef struct MailTokenIn

```
typedef struct MailTokenIn {
        struct MESSAGE_ID MessageID;
        char RecipientName[ID_SIZE];
        float CompletionTime;
} MailTokenIn;
```

MailTokeOut과 반대되는 토큰으로 스케쥴러 안에서 스케쥴이 모두 완료되면 그 토큰을 준 스타로 다시 토큰을 돌려보내 그 스타가 다음 동작을 할 수 있도록 하기 위한 구조체이며, MailTokenOut을 발생한 스타는 MailTokenIn을 기다리고 있다.

MessageID, RecipientName
위와 동일

CompletionTime
스케쥴링 완료된 이벤트의 시간을 가진 필드로 해당 스타안에 있는 이벤트를 다음 스타로 전달해야하는 시간을 가지고 있다.

⑧ typedef struct SCHEDULER_TABLE

```
typedef struct SCHEDULER_TABLE {
```

```
            struct MESSAGE_ID MessageID;
            char RecipientName[ID_SIZE];
            float ArrivalTime;
            float CompletionTime;
            float Delay;
            float Remainder;
            unsigned int Count;
            char From[ID_SIZE];
            char To[ID_SIZE];
            int Priority;
        } SCHEDULER_TABLE;
```

각 스타에서 스케쥴러에게 필요한 토큰을 MailTokenOut 타입으로 받으면, 스케쥴러는 SCHEDULER_TABLE을 만들어 스케쥴을 관리한다.

MessageID, RecipientName, ArrivalTime, CompletionTime, Delay

위와 동일

Remainder

이벤트가 Delay를 가지고 스케쥴러 안으로 들어오면 스케쥴러는 정해진 시간만큼만 수행하고 다른이벤트를 스케쥴링 하게 되는 데, 그 때 한번에 완료되지 못한 나머지 시간을 담고 있는 필드이다. 이 필드값이 0이 되면 그 이벤트는 스케쥴 완료 되었다고 말한다.

Count

Remainder의 값을 보고 앞으로 몇 번의 스케쥴링 횟수를 받아야만 스케쥴 완료가 되는 지를 나타내는 필드로, Remainder를 시간할당량으로 나누었을 때의 ceil값을 취한다.

From

토큰이 어느 스타에서 들어온 지를 나타내는 필드이다.

To

토큰이 스케쥴 완료가 되었을 때 어느 스타로 돌려보내져야 되는 지를 나타내는 필드이다.

Priority

스케쥴링 정책과 관련이 있는 필드로, 만일 스케쥴링 정책이 라운드 로빈이 아닌 우선순위기반 스케쥴링 정책을 사용했을 경우 스케쥴링 테이블안에 있는 이벤트들의 우선순위를 나타내는 값이다. 이 값은 시뮬레이터 시작 전에 사용자가 임의로 바꿀수 있다. 스케쥴링 정책이 라운드 로빈일 경우 사용되지 않는다.

4.1.2 최상위 수준 스키메틱

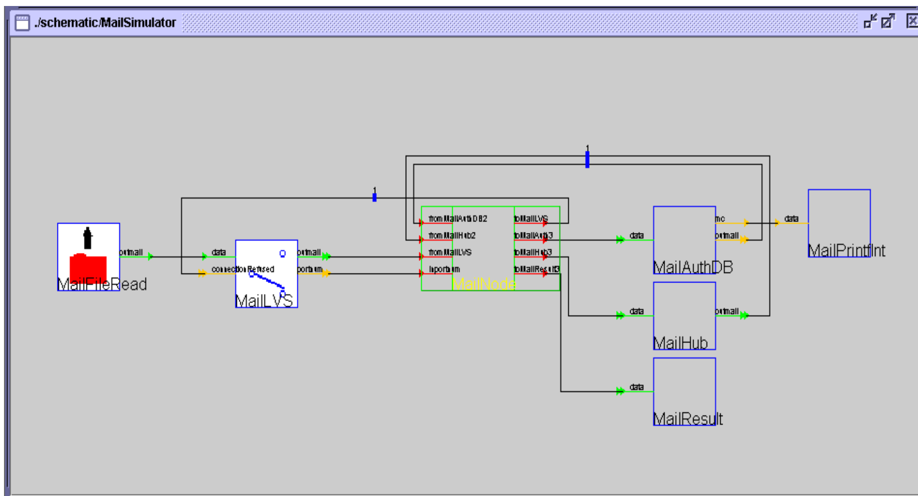


그림 4. 최상위 수준의 시뮬레이터 스키메틱

가장 최상위 레벨로써 MailFileRead는 파일에 씌여진 입력파일을

하나씩 읽어 MailLVS에게 전달한다. refireAtTime(completionTime)를 사용하여 다른 스타의 invoke없이 정해진 시간에 스스로 실행될 수 있다. MailNode라고 씌여진 갤럭시는 하나의 컴퓨터 노드를 모델한 블럭으로 내부에 리눅스 시스템을 시뮬레이션 할 수 있는 시스템 모델 블럭과 메일 서버의 구체적인 동작을 모델한 메일 서버블럭을 가지고 있어서, MailLVS에서 라운드 로빈으로 던져주는 메일 이벤트를 처리한다. 제 5장의 실험에서는 이 노드의 개수가 실험상황에 따라서 변경되기도 한다.

MailAuthDB는 SMTP와 POP 접속에서 일어나는 사용자 인증을 담당하는 독립된 데이터베이스 컴퓨터를 모델링 한 것으로 사용자의 메일박스가 위치하고 있는 메일 노드를 돌려준다

MailHub 스타는 입력으로 들어온 메일 이벤트의 메일박스 위치가 현재 자신의 노드가 아닌 경우, 클러스터로 묶여진 시스템 네트워크를 통해 해당 노드로 전달되어 동작하는데 소요되는 네트워크 지연시간과 포트 연결 스위칭을 모델링한 네트워크 모델 부분이다. 나머지 MailPrintInt은 시스템에서 사용되고 있는 변수의 값을 보기 위해 만들어 놓은 임시 출력 스타이다.

4.1.2.1 MailFileRead

동작 설명

파일에서 입력을 받아 MailLVS에게 읽어들인 시간간격에 맞추어 이벤트를 넘겨주는 시작 역할을 한다. DE 도메인에서 소스 스타에 해당하는 것으로 시뮬레이터를 처음 시작할 때 첫번째 이벤트와 두

번째 이벤트를 동시에 읽어 다음 이벤트의 순차적인 실행을 보장한다. DE 도메인 커널 메소드인 refireAtTime(completionTime)를 사용하여 다른 스타의 이벤트 없이 스스로 깨어나 실행된다.

4.1.2.2 MailLVS

동작 설명

MailLVS의 역할은 클러스터 시스템 앞단에서 외부에서 들어온 메일의 연결 접속을 정해진 내부의 분산정책으로 클러스터 내부 노드들로 이벤트를 연결한다. 흔히 알려진 클러스터링 로드 분산정책은 랜덤과 라운드 로빈, 가중치 기반 라운드 로빈, 최소 접속 스케쥴링, 가중치 기반 최소 접속 스케쥴링이 있다. 이중에서 가장 많이 쓰이는 방법이 라운드 로빈 스케쥴링으로 서버의 내부 처리 상황이나 클러스터 네트워크 시스템 등을 고려하지 않고, 단순히 연결 접속 요청을 돌아가면서 순차적으로 전달하는 방법이다. 구현이 간단하여 서버의 성능이 동일할 경우 가장 효율적인 것이 장점이고, 본 시뮬레이터에서도 기본적으로 라운드 로빈으로 설정했다. 나머지 정책들은 클러스터 노드 중 특정한 서버의 성능이나 동작환경 설정이 뛰어난 경우, 그쪽으로 연결을 많이 시도하는 동적 스케쥴링 방법으로써, 최소 접속 스케쥴링은 가장 접속이 적은 서버로 보내는 방식이고, 가중치 기반 최소 접속은 이 둘을 합쳐놓은 것이다.

MailLVS에서는 *portnum*을 통해 뒤에 있는 메일 서버노드들에게 자신들이 몇 번째 노드인지를 알려준다. 메일 노드 자체는 자신이 몇 번째 노드인지를 스스로 알 수 없고, MailLVS에 자신이 몇 번째

순서로 접속돼 있는지에 따라 결정이 되기 때문에 MailLVS가 해당 노드에게 반드시 그 노드번호를 직접 넘겨줘야 한다. 이것은 이후 operation thread가 인증 과정에서 MailAuthDB에게 자신의 노드번호를 넘겨주기 위해 필요하다.

4.1.2.3 MailAuthDB

동작 설명

MailAuthDB에서는 메일 이벤트의 *Tag* 필드가 *Unknown*일 경우, 인증을 위해 MailOpThread에서 보내어진 이벤트를 수신한다. MailAuthDB안에는 클러스터 시스템내의 모든 사용자에 관한 인증 내용을 가지고 있으며, 메일 시스템과는 독립된 노드로써 표현된다. 이 노드는 SMTP의 인증 뿐만 아니라, POP 이벤트 처리에서 필요한 인증도 함께 수행한다.

4.1.3 Mail Node의 설계

아래의 그림은 하나의 메일 서버 컴퓨터를 모델링한 모습이다. MailLVS를 통해 들어온 메일 이벤트는 MailRoute 스타에서 *Opcode*에 따라 SMTP의 경우 MailSendmail안으로 들어가고 POP 메일 이벤트의 경우는 MailPOPDaemon으로 전달되어진다. MailSendmail, MailMDA등은 2장에서 설명한 SMTP 프로세스를 모델한 것이고, MailSystem은 메일 서버의 구체적인 동작을 표현한 갤럭시이다. 이들은 모두 MailScheduler에 의해서 스케쥴된다.

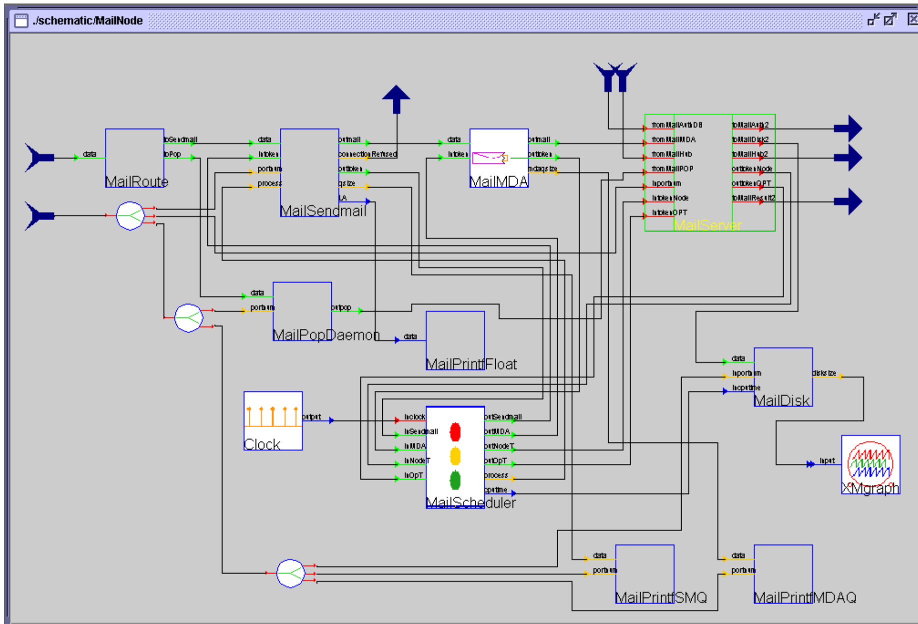


그림 5. Mail node의 내부 설계 스키메틱

4.1.3.1 MailRoute

동작 설명

MailRoute는 실제 컴퓨터 노드에 있는 구성요소를 모델링한 것은 아니다. SMTP 연결 요청은 TCP 25번 포트를 사용하지만 POP3 연결은 TCP 110번 포트를 이용한다. 하지만 시뮬레이터 상에서 TCP 포트를 번호단위로 구분할 필요도 없을 뿐만 아니라, MailFileRead에서 읽은 이벤트들은 SMTP와 POP이 함께 섞여 있기 때문에 시뮬레이터 내부에서는 이들을 구분해서 연결해 주면 된다.

4.1.3.2 MailPOPDaemon

동작 설명

POP 데몬 스타 역시 TCP 110번 포트로 들어오는 POP 이벤트를 메일 서버 내부로 보내주기 위해 만든 스타이다. 실제 메일 시스템에서의 POP의 연결 접속도 operation 스레드가 담당하고 있기 때문에 sendmail, MDA를 거치지 않고 바로 메일 시스템으로 보내는 역할을 담당한다.

4.1.3.3 MailSendmail

동작 설명

'sendmail'은 SMTP 연결이 들어오는 들어올 경우 자식 프로세스를 fork하여 전자우편을 받은 뒤, 지난 1분 동안 전자우편을 처리하기 위해 실행중인 프로세스들의 평균 개수[2)]를 검사한다.
sendmail은 현재의 la를 검사하여 그 값이 QueueLA보다 크면 아래의 식을 계산하여 만족하면 메시지를 큐잉하고, 그렇지 않으면 전송하며, 이 값이 RefuseLA값보다 크면 SMTP 연결을 거부한다.

---

2) load average(lg)
지난 1분동안 메일 메시지를 처리하기 위해 실행중인 프로세스의 평균 개수로 QueueFactor와 QueueLA 값들의 조합으로 메시지가 큐잉이 될것인지 전달이 될 것인지를 결정.

msgpri >= QueueFactor/(la - QueueLA+1) (식1)

msgpri는 메시지 priority를 나타내는 것으로 sendmail이 이 우선순위 값에 따라서 메일 처리의 선후가 결정된다. QueueFactor값(q)은 default 600,000으로 priority 결정을 위한 multiplying factor이고, QueueLA값(x)은 default 8로 주어져있다.

msgpri = nbyte - (class * z) + (recipients * y) (식2)

여기서 n은 메시지의 byte 크기, recipients는 수신자의 수른 나타내며 enumeration 타입의 class는 도착한 메일 메시지를 성격에 따라 분류시켜 놓은 것으로, 다음과 같은 값을 갖는다. default는 0이다.

enum class {first-class, special-delivery, junk, bulk, list};

여기서 size는 전자우편의 크기, receipients는 수신자의 숫자를 나타내며 class는 default 0, ClassFactor(z)는 default 1800, RecipientFactor(y)는 default 30,000이다. refuseLA는 default 12이다. 이렇게 하여 결정된 메일은 *outtoken*을 발생시켜 스케쥴러가 MailMDA로 이벤트 발생을 지시한다. MailMDA로 이벤트를 발생시킬 때, 다중 수신자의 경우 RecipientList의 내용을 파싱해서 지정된 수신자에게 메일이 배달될 수 있도록 여러개의 MDA를 fork한다.

Pseudo Code

```
현재의 load average값을 읽는다;

if(data 포트에 새로운 이벤트가 존재하면) {

   MailType의 패킷을 MailTypeExtension으로 이벤트 패킷을 확장한다;
   /* 수신된 이벤트에 대한 큐잉모드를 결정하는 부분이다. */

   if(QueueLA < CurrentLA && CurrentLA < RefuseLA) {
        /* 아래의 값들을 가지고 msgpri(메시지 우선순위 값)을 계산한다. */
        msgpri = 메일 이벤트의 크기-(CLASS*ClassFactor) +
                  (메일 이벤트의 수신자 수)*RecipientFactor);

       if(msgpri > (QueueFactor/(CurrentLA-QueueLA+1))) {
             queueingmode = true; /* 큐잉 모드로 동작 세팅 */
        } else { queueingmode = false; /* 큐잉 모드 동작 비 세팅 */}
        } else if(CurrentLA >= float(RefuseLA)) {
                 /* SMTP 커넥션을 거절 */
                 connrefused = true;
        }else {  /* 즉시 배달 세팅 즉, 큐잉 모드 동작 비 세팅 */
           queueingmode = false;
        } //end if-else statement

        if(메모리가 full이 아니면) {
           MTEBuffer에 들어온 메일 이벤트를 저장한다;
        }
        /* 이벤트의 우선순위에 따라 선택정렬을 이용하여 정렬한다. */
        QueueSort(MTEBuffer);

        if(현재의 queueingmode가 거짓 && MTEBuffer가 차 있으면) {
           /* 스케쥴러로 토큰을 발생시킨다. */
           if(보내고자하는 메일 이벤트의 수신자가 2이상이면) {
                 다중 수신자의 명 수 만큼 반복해서 동일한 토큰을 만들어 스
                 케쥴러에 이벤트를 발생시킨다.
           }

/*****************************************************************
 ** 수신된 메일에서 다중 수신자가 있을 경우, 메일 시스템이 인식하는
 ** 메일의 개수는 1개로 보지만, 다중 수신자의 수만큼 MDA가 호출되어
 ** 메일을 처리하기 때문에 ActiveTable안에는 오직 하나의 메일 이벤트
 ** 만이 저장되고, 내부적으로 다중 수신자의 수를 카운트하고 있어서
 ** 스케쥴러에서 해당 토큰이 들어올 때마다 카운트를 줄여나가 카운트가
 ** 0이 되면 그 때 ActiveTable에서 삭제한다.
 *****************************************************************/
```

```
        보낸 이벤트의 첫 번째 수신자 이름에 대해서 ActiveTable에 저장한다;
    } //end if(queueingmode != true)
} //end if

if(intoken 포트에 새로운 이벤트가 존재하면) {
   MDA로 이벤트를 발생시킨다;
   ActiveTable에서 들어온 토큰에 해당되는 이벤트를 찾아 카운트를 하나 줄임;
   if(카운트가 0이 되면) {
        ActiveTable에서 완전히 삭제한다;
    }
  } //end if(intoken.dataNew)

  MTEBuffer와 ActiveTable의 크기를 출력한다.
```

4.1.3.4 MailMDA

동작 설명

MailSendmail로부터 받은 이벤트에 대해 MailMDA는 배달 지연시간에 대한 값을 할당한 후 토큰을 통해 스케쥴러에게 넘겨준다. 외적인 동작은 MailSendmail과 같다. 매번 스케쥴될 때마다 *mdaqsize*가 출력된다.

4.1.3.5 MailScheduler

동작 설명

메일 시뮬레이터에서 가장 중요한 스타인 MailScheduler는 리눅스 스케쥴러의 역할을 담당한다.

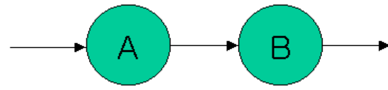


그림 6. DE 스타

위와 같이 DE 스타 블럭이 있을 때, A의 입력으로 이벤트 a가 α라는 지연시간을 가지고 수행되고, B 내에서 수행되고 있는 b는 β의 지연시간을 가지고 수행된다고 했을 때, DE 도메인에서 동작하는 A, B는 DE global scheduler의 global clock의 입장에서는 동시에 처리된다고 본다. - PeaCE의 DE 도메인은 기본적으로 동시수행 시멘틱을 내포하고 있다. - 이것은 하드웨어를 모델링하여 동작시키다는 측면으로는 자연스런 개념이지만, 소프트웨어를 OS상에서 동작시키는 것을 모델링한다는 측면에서는 오동작을 야기한다. 왜냐하면 OS는 정해진 시간 단위(Linux의 경우 10㎳)만큼만 하나의 프로세스를 스케쥴링 하기 때문이다. 따라서 MailScheduler가 하는 역할은 이벤트 a, b의 시간 진행을 둘 중 어느 하나를 정해진 시간 간격만큼만 수행시키고 나머지 하나는 수행시간을 진행시키지 않게 하는데 있다. 구현된 시뮬레이터에서는 *MailTokenIn, MailTokenOut*이라는 구조체를 사용하여 스타와 스케쥴러 사이에서 데이터를 전달한다.

MailScheduler는 *inclock*을 통해 10㎳마다 자신을 트리거 시켜 스케쥴러가 실행되게 하고, MailSendmail, MailMDA, MailNode-Thread, MailOpThread들에서 들어온 *MailTokenOut* 타입의 구조체를 통해 스케쥴할 대상과 얼마의 지연시간을 갖는지를 얻어온다. 스케쥴러안으로 들어온 메일 이벤트는 아래의 예로 보인 스케쥴링 테이블을 구성하여 라운드 로빈으로 10㎳만큼 시간을 진행시킨다.

Scheduler Time = 26.00 TIMEQUANTUM = 10.00

ID	Recipient Name	Arrival Time	Completion Time	Delay	Remainder	Count	From	To
1	test2	5.00	18.00	13.00	7.00	1	snedmail	MDA
0	test1	6.00	20.00	17.90	11.90	2	MDA	OPT
2	test3	10.00	22.00	10.00	6.00	1	sendmail	MDA
3	test4	15.00	24.00	18.00	16.00	2	sendmail	MDA
4	test5	20.00	26.00	16.00	14.00	2	sendmail	MDA

그림 7. 스케쥴러 테이블

Arrival Time은 이벤트가 처리해야할 스타 안으로 들어온 시간을 나타내고, Completion Time은 그 이벤트가 MailScheduler속에서 스케쥴링될 당시의 시간을 나타내는 것으로 MailScheduler에서 빠져나갈 때의 *completionTime*으로 적용된다. Delay는 스타에서 처리되야만 하는 시간을 나타내는 것이고, Remainder는 앞으로 남은 TIMEQUANTUM의 시간이 된다. C 코드로 이들을 간단히 표현하면, 다음과 같다.

```
Count = int(ceil(Remaind/TIMEQUANTUM));
Remaind -= TIMEQUANTUM;
schedulerTime += TIMEQUANTUM;
```

그러나, 만일 어느 한 이벤트의 Remaind가 TIMEQUANTU보다 작을 경우, slack time이 발생하게 된다. 이럴 경우, 리눅스 스케쥴러는 다음 프로세스에게 남은 slack time을 넘겨주기 때문에 MailSchdeuler에서도 Remaind를 TIMEQUANTUM이 아닌 소비된 시간으로 변경하는 것이 더 정확하다.

```
Remaind -= timeConsume;
```

마지막으로, From과 To는 그 이벤트가 어디서 들어와서 앞으로 어디로 보내져야 되는지를 나타내는 설명부분이다.

Pseudo Code

```
CPU_RUN_TIME  = 0.0; /* local value */
TIMEQUANTUM = 10.0; /* constant value */

if(inOpT 포트에 새로운 데이터가 존재하면) {
   스케쥴러 테이블에 삽입한다;
}
if(inNodeT 포트에 새로운 데이터가 존재하면) {
   스케쥴러 테이블에 삽입한다;
}
if(inMDA 포트에 새로운 데이터가 존재하면) {
   스케쥴러 테이블에 삽입한다;
}
if(inSendmail 포트에 새로운 데이터가 존재하면) {
   스케쥴러 테이블에 삽입한다;
}

if(테이블이 비어있지 않으면) {
   /* switch 문장에서 결정된 스케쥴링 정책에 따라서 스케쥴링한다. */
   /* 기본적으로 round-robin이다.
   switch(SCHEDULING_POLICY) {
        case 0 : // The scheduling policy is ROUND-ROBIN.
                 // no sort.
                 break;
        case 1 : // The scheduling policy is PRIORITY.
                 selectionSort();
                 break;
        default :
                 Error::abortRun("No Policy");
                 break;
   } //end switch
```

```
 if(현재 이벤트의 스케쥴 잔량 시간 >= TIMEQUANTUM) {
        소비 시간 = TIMEQUANTUM;
        잔량 시간에서 소비 시간만큼 뺀다;
        스케쥴러 시간을 소비한 만큼 증가시킨다;
        CPU_RUN_TIME = 소비한 시간;
 }
else if(현재 이벤트의 스케쥴 잔량 시간 == TIMEQUNTUM) {
        소비 시간 = TIMEQUANTUM;
        잔량 시간에서 소비 시간만큼 뺀다;
        스케쥴러 시간을 소비한 만큼 증가시킨다;
        FinEvent에 그 이벤트를 삽입한다;
        FinEvent의 크기를 하나 증가시킨다;
        스케쥴러 안에서 그 이벤트를 삭제한다;

        else {
        /* 현재 이벤트의 스케쥴 잔량이 TIMEQUANTUM보다 작다면 */
              소비 시간 = 잔량 시간;
              스케쥴러 시간을 소비 시간만큼 증가시킨다;
              FinEvent에 그 이벤트를 삽입한다;
              FinEvent의 크기를 하나 증가시킨다;
              스케쥴러 안에서 그 이벤트를 삭제한다;
              슬랙 시간 = TIMEQUANTUM - 소비 시간;

              if(스케쥴러 안에 이벤트가 존재하면) {
                 while(슬랙 시간이 0이거나 잔여 이벤트가 없어질 때까지) {
                     스케쥴링 정책에 의한 다음 스케쥴링 대상을 스케쥴한다;
                  } //end while
               } //end if
        } //end else
        if(FinEvent에 하나 이상의 종료 이벤트가 들어있으면) {
           /* 스케쥴러 테이블의 디렉션을 보고 어디로 토큰을
              돌려져야 하는지 결정 */
           if( strcmp(FinEvent[j].To, "MDA") == 0)
                  outSendmail.put(completionTime) << env;
           if( strcmp(FinEvent[j].To, "NodeThread") == 0)
                  outMDA.put(completionTime) << env;
           if( strcmp(FinEvent[j].To, "OPT") == 0)
                  outNodeT.put(completionTime) << env;
           if( strcmp(FinEvent[j].To, "Next") == 0)
                  outOpT.put(completionTime) << env;
        } //end if
```

```
} //end if
```

4.1.4 Mail System의 설계

그림 8. Mail System 내부 설계 스키메틱

구현한 메일 서버 갤럭시의 내부 schematic모습이다. MailMDA에서 전달받은 메일 이벤트는 MailNodeThread가 받아서 준비 큐(ready queue)에 저장한 다음 MailOpThread에게 넘겨주는 역할을 한다. node thread는 반드시 임의의 시간에 반드시 한 개만 수행되어야 하는데 반해 operation thread는 사용자에 의해서 정의된 임의의 개수만큼 동시에 실행될 수 있다.

MailNodeThread와 MailOpThread 메일 시스템내부에 있기는 하지만 CPU 시간 할당을 받아 처리가 되는 만큼 다른 구성 블럭들

처럼 리눅스 스케쥴러에 의해 영향을 받아 처리가 된다. 따라서 입력, 출력 토큰을 스케쥴러에게 전달함으로써 MailScheduler에게 자신의 스케쥴 여부를 맡긴다.

4.1.4.1 MailNodeThread

동작 설명

NodeThread는 다른 노드에서 보내진 메일과 내부에서 보내진 메일을 ready queue에 넣는 동작을 한다. 실제 구현에서 NodeThread는 오직 하나만 존재하기 때문에 임의의 시간에 MailScheduler를 살펴보더라도 하나만 나타난다. 이것은 sendmail, MDA, OpThread 등 다른 스타 블럭과는 다른 점이다. 다른 스타 블럭은 시스템 상에서 특정시간동안 2개 이상의 프로세스나 스레드가 생성될 수 있는 멀티 스레드인데 NodeThread는 오직 하나만 활성화될 수 있다.

Pseudo Code

```
/*****************************************************************
 ** MY_NODE_NUMBER가 0과 같으면 Local 버퍼를 먼저 읽고 그렇지
 ** 않으면 Remote 버퍼를 먼저 읽는다.
 ** data 포트로 들어오는 메일 이벤트 태그를 'Unknown'으로 한다.
*****************************************************************/

if(MY_NODE_NUMBER가 0과 같으면) {
   if(data 포트에 새로운 이벤트가 들어오면) {
        NodeThreadBuffer에 들어온 이벤트를 삽입한다;
        이벤트의 태그를 Unknown으로 한다;
        도착시간, node thread에서 수행되는 시간을 기록한다;
        sorcePort와 destPort를 MY_NODE_NUMBER로 초기화한다;
  }
```

```
 if(Rdata 포트에 새로운 이벤트가 들어오면) {
       NodeThreadBuffer에 들어온 이벤트를 삽입한다;
       도착시간, node thread에서 수행되는 시간을 기록한다;
 }
else {  /* MY_NODE_NUMBER가 0과 같지 않으면 */
  if(Rdata 포트에 새로운 이벤트가 들어오면) {
      NodeThreadBuffer에 들어온 이벤트를 삽입한다;
      도착시간, node thread에서 수행되는 시간을 기록한다;
  }
  if(data 포트에 새로운 이벤트가 들어오면) {
    NodeThreadBuffer에 들어온 이벤트를 삽입한다;
    이벤트의 태크를 Unknown으로 한다;
    도착시간, node thread에서 수행되는 시간을 기록한다;
    sorcePort와 destPort를 MY_NODE_NUMBER로 초기화한다;
  }
} //end else


/**********************************************************************
 ** 스케쥴러에서 완료 토큰이 들어오면, ActiveBuffer에서 들어온 이벤트를
 ** 검색하여 삭제하고, operation thread로 그 이벤트를 보낸다.
 ** Node thread는 단일 스레드이기 때문에 스케쥴러 안에 오직 하나만 존재
 ** 하여야 하기 때문에 lock 플래그를 이용해 스케쥴러 안에 node thread 이
 ** 벤트가 두 개 이상 삽입되어지는 것을 방지한다.
 **********************************************************************/
if(intoken 포트에 새로운 데이터가 존재하면 {
  for(ActiveBuffer 인덱스까지) {
    ActiveBuffer에서 들어온 토큰과 같은 id를 인출한다;
    인출된 메일 이벤트를 operation thread로 보낸다;
    lock을 푼다.
  } //end for
} //end if

if(NodeThreadBuffer가 비어있지 않고 && lock이 풀려 있으면) {
  NodeThreadBuffer에서 이벤트를 하나 인출해 토큰으로 만든다;
  토큰을 스케쥴러로 보낸다;
  NodeThreadBuffer안에 있는 이벤트를 ActiveNodeThread안에 저장한다;
  ActiveNodeThread로 보낸 NodeThreadBuffer안에 있는 이벤트를 삭제한다;
  lock을 건다;
} //end if
 NodeThreadBuffer의 크기를 출력한다.
```

4.1.4.2 MailOpThread

MailOpThread의 내부 디자인이다. MailOpThread가 하는 일은 인증을 비롯해 메일 서버가 디스크에 메일을 쓰거나, POP3 세션이 들어왔을 경우 메일박스의 메일을 검색하여 삭제하는 역할을 하며, 다른 노드로 메일 이벤트를 포워딩 하는 동작을 수행한다. 내부에 node thread로부터 들어온 이벤트를 저장하는 worklist queue를 가지고 있다.

MailOpThread 갤럭시로 들어온 메일 이벤트는 스레드 내에서 인증을 하기 전까지 그 메일 이벤트의 메일박스가 어느 노드에 있는지 알 수 없다. 경우에 따라 로컬에 자신의 메일박스를 갖지 않은 경우 다른 노드로 보내주어야 하고, 다른 노드로 보내 준 메일 이벤트에 대한 acknowledgment를 기다려야 한다.

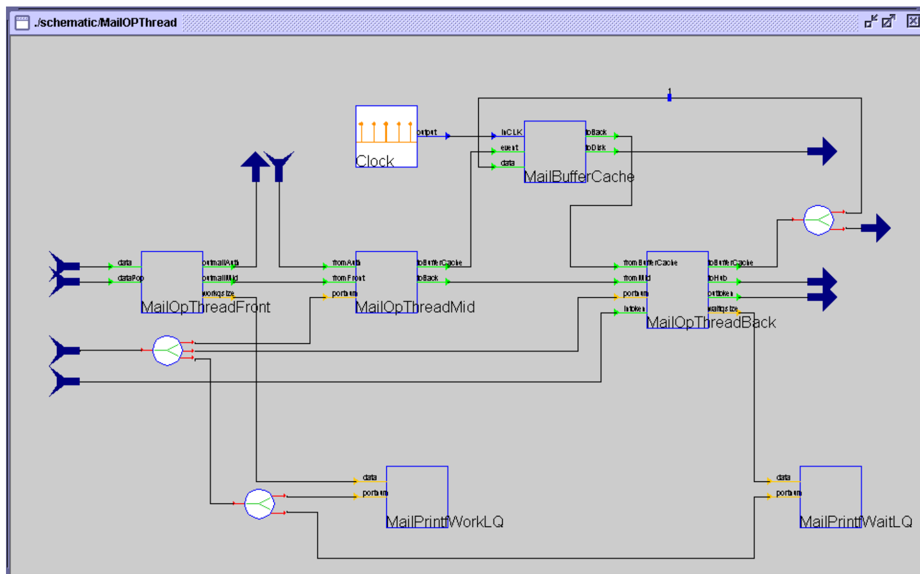


그림 9. Operation Thread 갤럭시의 내부 설계 스키메틱

4.1.4.2.1 MailOpThreadFront

동작 설명

MailNodeThread로부터 이벤트가 들어오면 그 이벤트가 다른 노드로 보낸 메일 이벤트의 reply 이벤트인지 아니면 일반적인 메일 이벤트인지 구별한다. 만일 일반 메일 이벤트이면 worklist queue의 제일 뒤쪽에서부터 순서대로 삽입하지만, reply 이벤트이면 큐의 제일 앞쪽부터 삽입한다. 따라서 OpThread는 큐에 reply 이벤트가 존재하면 일반 메일 이벤트보다 먼저 처리된다. 또한 아직 인증 처리가 안된 메일 이벤트의 태그인 *Unknown*은 MailAuthDB로 인증이 끝난 *Forwarding*, *Relay*같은 메일 이벤트는 MailOpThreadBack으로 전달한다.

Pseudo Code

```
/*****************************************************************
 ** Node thread로부터 들어온 메일 이벤트와 POP 데몬으로부터 직접
 ** 들어온 메일에 대하여 처리한다.
 ** Reply 이벤트는 다른 이벤트들에 대해서 work list 큐 안에서 LIFO로
 ** 동작하고, 다른 이벤트들은 모두 FIFO로 동작한다.
 ** 단, 같은 tag를 가진 이벤트끼리는 FIFO로 동작한다.
*****************************************************************/

 if(data 포트에 새로운 이벤트가 도착하면) {
    들어온 메일 이벤트에 대하여 지연시간을 할당한다;
    if(들어온 메일 이벤트의 tag가 'Reply') {
      WorkListQueue 앞단에 삽입한다;
    } else {
      WorkListQueue 뒤에 순서대로 삽입한다;
```

```
    }
}

if(dataPop 포트에 새로운 이벤트가 도착하면) {
   WorkListQueue 뒤에 순서대로 삽입한다;
}

if(WorkListQueue가 비어있지 않으면) {
  if(WorkListQueue안에 'Reply' 이벤트가 존재하지 않으면) {
      WorkListQueue안에 있는 가장 앞에 있는 이벤트를 처리한다;
         if(큐에서 인출된 이벤트의 태그가 'Unknown'이거나 'Pop'이면) {
            인증을 위해 인증 서버로 보낸다;
         } else if(큐에서 인출된 이벤트의 태그가 'Forwarding'이면) {
                MailOpThreadMid로 전송된다;
         }
         위에서 처리된 WorkListQueue안에 있던 이벤트는 삭제된다;
  }else {
         /* WorkListQueue안에 'Reply' 이벤트가 존재하면 */
         MailOpThreadMid로 대기하고 있던 reply 이벤트 중 가장 먼저 들어온
         이벤트 하나를 전송한다;
         처리된 WorkListQueue안에 있던 이벤트는 삭제된다;
  } //end else
}

WorkListQueue의 크기를 출력한다;
```

4.1.4.2.2 MailOpThreadMid

동작설명

DE 모델에서 리턴값을 갖는 함수식의 동작을 표현하기 위해서 만든 스타이다.

Pseudo Code

```
if(fromAuth 포트에 새로운 데이터가 들어오면) {
    if(들어온 데이터의 태그가 'Local' 이면) {
          그 이벤트를 MailBufferCache로 보낸다;
    } else if(들어온 데이터의 태그가 'Pop' 이면) {
          그 이벤트를 MailOpThreadBack으로 전송한다;
    } else if(들어온 데이터의 태그가 'Remote' 이면) {
          그 이벤트를 MailOpThreadBack으로 전송한다;
    }
} //end if

if(fromFront 포트에 새로운 데이터가 들어오면) {
   if(들어온 데이터의 태그가 'Forwarding' 이면) {
          그 이벤트를 MailBufferCache로 보낸다;
   } else if(들어온 데이터의 태그가 'Reply' 이면) {
          그 이벤트를 MailOpThreadBack으로 전송한다;
   }
} //end if
```

4.1.4.2.3 MailOpThreadBack

동작 설명

MailOpThreadBack은 인증이 끝난 메일 이벤트(*Local, Remote, Forwarding* 이벤트) 혹은 MailOpThreadFront에서 인증이 필요 없는 메일 이벤트(*reply* 이벤트)에 대해서 자신의 디스크가 있는 노드로 연결시켜 주는 일을 하며, 만일 인증이 자기 노드번호가 아닐 경우는 해당 노드번호를 목적지 포트에 써서 MailHub가 전달해줄 수 있게 한다.

Pseudo Code

```
if(fromBufferCache 포트에 새로운 데이터가 존재하면)
```

```
    토큰을 만들어 스케쥴러로 보낸다;
    ActiveBuffer에 그 이벤트를 삽입하여 스케쥴이 종료되어 돌아오기를
    기다린다;
} //end if

if(fromMid 포트에 새로운 데이터가 존재하면) {
    토큰을 만들어 스케쥴러로 보낸다;
    ActiveBuffer에 그 이벤트를 삽입하여 스케쥴이 종료되어 돌아오기를
    기다린다;
} //end if

if(intoken 포트에 새로운 데이터가 존재하면) {
    ActiveBuffer안에서 들어온 토큰과 일치하는 이벤트를 탐색한다;
    탐색 결과 일치하는 것이 있으면 검색된 이벤트를 ActiveBuffer안에서
    삭제한다;

     if(검색된 이벤트의 태그가 Reply 이면) {
          WaitListQueue에서 그 이벤트를 삭제한다.
     } else if(검색된 이벤트의 태그가 Local 이면) {
          MailBufferCache로 보낸다;
     } else if(검색된 이벤트의 태그가 Forwarding 이면) {
          MailBufferCache로 보낸다;
          sourcePort의 노드번호를 보고 replay event를 돌려보낸다;
     } else if(검색된 이벤트의 태그가 Remote 이면) {
          destPort의 노드번호를 보고 해당 노드로 이벤트를 보낸다;
          Remote 메일 이벤트가 해당 메일박스가 있는 노드에서 처리되는
          동작에 대한 응답을 기다리기 위해 WaitListQueue안에 들어온
          이벤트에 관한 정보를 저장한다;
    } else if(검색된 이벤트의 태그가 Pop 이면) {
        if(destPort가 MY_NODE_NUMBER와 같으면) {
            자신의 노드에 메일박스가 존재하기 때문에 바로 MailBufferCache로
            이벤트를 보낸다;
        } else {
            자신의 노드에 메일박스가 존재하지 않기 때문에 MailHub로 이벤트를
            발생시킨다;
            WaitListQueue에 삽입하여 응답을 기다린다;
        }
    }
} //end if
WaitListQueue의 크기를 출력한다;
```

4.1.4.2.4 MailBufferCache

동작 설명

버퍼캐쉬는 우리들이 흔히 알고 있는 컴퓨터 구조내의 하드웨어 캐쉬가 아닌 운영체제 내부에 완전히 소프트웨어로만 구현돼 있는 것으로, 이중 연결 리스트로 이루어져 있다. 1024바이트를 기본 캐쉬 블럭으로 하고 운영체제에 따라 2048, 4096 바이트 등 다양한 크기의 블럭들을 지원한다. 블럭들은 2개의 리스트로 관리되는데, 캐쉬 블럭으로 사용되지 않고 있는 블럭들은 자유버퍼 리스트(free list)에 연결되어 있으며 블럭이 필요한 경우 블럭들을 할당한다.

할당된 블럭들은 그 블럭을 사용하고 있는 디바이스 번호와 블럭 번호의 조합으로 이뤄진 해쉬 테이블에 의해서 관리되고 있으며, 디스크 읽기 혹은 쓰기 동작이 일어나는 경우 가장 먼저 해쉬 테이블 내의 버퍼 블럭들을 탐색하여 상위 시스템으로 돌려준다. (cache hit) 만일 찾고자 하는 버퍼가 없을 경우(cache miss), 물리적인 디스크에서 블럭을 읽어 자유 버퍼리스트에서 얻은 블럭에 할당한 후 상위 시스템으로 돌려진다. 버퍼리스트에서 해쉬테이블 내의 블럭으로 할당할 경우, 이 때 새롭게 할당할 블럭들이 없거나, 실제 디스크에 쓰여져야 할 블럭의 경우 DMA 신호에 의해 필요한 블럭들의 수와 데이터를 디스크로 전송(디스크 쓰기)후 블럭들을 할당한다. 이러한 디스크 쓰기는 사용자 명령에 의해 명시적으로 일어날 수 있고, 운영체제 환경설정에 의해 설정된 임의의 시간 주기로 일어날 수도 있다. 리눅스에서는 bdflush 커널데몬이 이러한 역할을 수행하며, 사용자 명령으로는 update 명령이 주로 쓰인다.

시뮬레이션 모델 수준에서의 버퍼캐쉬 모델링은 커널 자료구조 수준의 모델링이 아니라 버퍼캐쉬 동작에서 걸리는 시간에 관한 모델 수준이면 적당하다.

Pseudo Code

```
if(inCLK 포트에 새로운 데이터가 존재하면) {
     /* 정해진 시간에 맞추어 자동적으로 flush를 동작하도록 한다. */
     completionTime = arrivalTime;
     flushing();
}

if(event 포트에 새로운 데이터가 존재하면) {
     도착된 이벤트의 크기를 얻는다;
     index = isCached(도착된 이벤트);
     if(index > -1) { /* cache hit */
       버퍼캐쉬내의 LRU 비트를 다시 세팅;
     }

    if(isBufferCacheFull(도착된 이벤트)) {
         /* 버퍼캐쉬가 풀이 아닐 때까지 버퍼를 비운다. */
         while(isBufferCacheFull(incoming_event_size)) flushing();
         DMAinitTime = 10.0;

    } else {
         DMAinitTime = 0.0; /* no DMA initialization time */
         inEvent->TagPayload.Payload.Delay =
                                   memoryWritingTime(incoming_event_size);
    }

    /* no overlapped */
    if(arrivalTime > completionTime)
         completionTime = arrivalTime + DMAinitTime;
    else completionTime = completionTime + DMAinitTime;

    이벤트를 MailOpThreadBack으로 돌려 보낸다;

}//end if(event.dataNew)
```

```
if(data 포트에 새로운 데이터라 존재하면) {
      버퍼캐쉬내의 LRU내의 제일 끝부분에 이벤트를 삽입한다;
      current_buffer_cache_index = current_buffer_cache_index + 1;
} //end if
```

4.1.5 MailHub의 설계 (네트워크 모델링)

동작 설명

MailHub에서는 MailOpThread에서 다른 노드를 목적지로 갖는 메일 이벤트에 대해서는 네트워크 지연시간에 따라 버퍼링을 통해 직렬화를 실시한다. 직렬화가 필요한 이유는 스위칭 허브의 경우, 목적지가 서로 같지 않으면 라우팅할 때 충돌은 일어나지 않는다. 따라서, 다른 목적지로 가는 메일 이벤트에 대해서 자신이 네트워크에서 걸리는 지연 시간이 영향을 받아서는 안되며, 같은 목적지로 두 개 이상의 이벤트가 존재하는 경우에만 MailHub에 들어온 시간에 따라 출력포트에 하나만 쓰는 직렬화를 시켜 내보내야 하기 때문이다. 내부적으로, NetStatus라는 MailTypeHub의 자료구조를 통해 현재 네트워크에서 일어나는 상태를 알 수 있다.

Pseudo Code

```
count = 0;
for(MailHub에 접속된 노드 수만큼 차례로 반복한다) {
   if(해당노드 포드에 새로운 데이터가 존재하면) {
        HubBuffer[count]에 도착한 이벤트를 삽입한다;
        if(HubBuffer[count]의 태그가 Remote이면) {
             Forwarding으로 변경한다;
         }
         HubBuffer[count]의 지연시간 필드를 초기화한다;
```

```
        count++;
    }
    다음 노드 포트 포인터로 변경한다;
} //end for

for(int i는 count의 수만큼 반복) {
 HubBuffer[i].destPort를 읽는다;

 /***************************************************************
  ** 네트워크 스위치 한 포트에 둘 이상이 동시 혹은 앞의 이벤트에 대한
  ** 전송이 완료되지 않은 상황에서 같은 포트에 다음 이벤트를 출력할 수
  ** 없다. 따라서 SERVICETIME이라는 자료구조 안에 해당 포트에 대해
  ** 이전의 이벤트가 언제 전송 종료가 되었는지를 검사하여, 이벤트를
  ** 직렬화 시켜야 할지를 결정하는 부분이다.
 ***************************************************************/

 if(arrivalTime >= SERVICETIME[outMail.destPort]) {
        completionTime = arrivalTime + getSwitchDelay(이벤트 크기);
 else   completionTime = completionTime +getSwitchDelay(이벤트 크기);

 SERVICETIME[outMail.destPort]에 자신의 이벤트 종료 시각을 기록;
 해당 노드로 메일 이벤트를 발생시킨다;

} //end for
```

제 5 장 실 험

실험은 우선 실제 메일 서버에서 각 메일 크기별로 100번을 동작시켜 각 부분에서 걸리는 시간을 평균으로 실측한 메일 서버 응답시간을 측정하여 시뮬레이터 모델의 지연 시간을 위한 자료로 사용하였고, 네트워크 지연시간을 위해 TCP socket을 크기와 노드 수를 변경하면서 각각 10000번을 반복 수행하여 걸린 전송시간 차이에 대한 평균을 구하여 실험하였다.

시뮬레이터의 입력 자료로 SPECmail2001을 사용하여, 입력의 정당성을 확보하였다.

5.1 실험 환경

실험 환경은 크게 실제 동작하고 있는 메일 서버의 동작환경과 시뮬레이터를 작성하여 시뮬레이팅한 환경에 대해서 소개하고 있다.

5.1.1 메일 시스템 환경

실측을 위한 클러스터 메일 서버는 1개의 DNS 서버를 갖추고 있고 총 8대의 PC가 100Mbps Fast Ethernet switch로 연결되어 있다. 메일은 1대의 LVS에서 라운드 로빈 스케쥴링으로 SMTP 커넥션을 각 노드도 연결한다.

각 메일 서버는 PentiumIII 550MHz dual CPU(실험시 single 모드로 부팅), 256MB SDRAM, EIDE 20GB HDD을 사용하고 있으며,

운영체제로 Redhat Linux 6.2를 설치했다. Sendmail로는 8.11.1을 사용했으며, sendmail이 local MDA를 'mail.local'로 설정하도록 설정 파일에서 'Mlocal' 관련 부분만을 수정했고, DNS를 통해 도착한 메일을 받도록 'Cw'항목을 수정했다. 그 이외의 설정은 default로 설정했다. 또한 디스크 쓰기에 대한 시간을 줄이기 위해 UDMA 66에 대한 옵션을 세팅했다.

5.1.2 시뮬레이터 환경

시뮬레이터의 구현과 실험환경으로 PentiumIII 550MHz dual CPU 512MB SDRAM, EIDE 8GB에 Linux Reahat 6.2 배포판에서 작업하였다. 시뮬레이터 엔진으로 사용한 PeaCE는 2001년 9월4일자 버전 0.2.1.9를 사용하였으며, 시뮬레이터 스키메틱 작성을 위한 클라이언트 프로그램으로 JDK 1.2.2를 사용했다.

5.2 SPECmail2001

SPECmail2001은 Standard Performance Evaluation Corporation (SPEC)이라고 불리우는 비영리 컴퓨터 벤더들의 모임에서 메일서버 시스템의 성능을 인터넷의 표준 프로토콜인 SMTP와 POP3에 기반한 전자메일 요구에 대한 처리정도를 가지고 평가하기 위한 벤치마크 소프트웨어이다.[12]

5.2.1. 시뮬레이터 입력로드의 정당성

본 연구에서 구현된 시뮬레이터를 검증하기 위해서는 실세계와 거의 비슷하고, 누구나 타당성이 있다고 인정할 만한 메일 부하가 있어야 한다. 만일 그렇지 않고, 임의의 부하량을 가지고 실험한다면, 구현한 시뮬레이터 동작 자체가 매우 작위적이며 특정한 입력파일에 대해서만 검증될 수밖에 없다. 따라서 SPECmail2001의 작업 부하를 사용함으로써 입력의 정당성을 갖는 부하에 대한 가이드라인을 제시하여 메일 서버의 성능 검증의 신뢰를 확보했다.

5.2.2 SPECmail 부하

SPECmail2001에서는 3개의 서로 다른 부하량을 가진 로드를 생성한다. 메일 서버에서 지원하는 총 POP사용자를 설정파일에서 작성하게 되면, 아래와 같이 200 : 1의 분당 메시지 수에 따른 작업 부하가 결정이 되는데, 이 부하량을 가지고 SPECmail 전체 수행 시간동안 각 80%, 100%, 120%의 부하량을 생성, 메일서버의 QoS를 측정한다.

1 SPECmail 2001 messages per minute

=

200 supported POP consumer users

5.2.3 SPECmail QoS (Quality of Service)

SPECmail2001에서 사용하고 있는 QoS는 고품질 메일서버 서비스를 제공하는 기준이 된다. 아래는 4개의 기준을 보이고 있다.

▷ 응답시간(Response Time)

모든 응답시간의 95%는 반드시 5초이내에 응답하여야 한다.

▷ 배달시간(Delivery Time)

SMTP를 통해 메일이 전달되는 시간으로, 로컬 사용자에게 배달되는 모든 메시지의 95%는 60초이내에 목적지 메일박스에 배달되야 한다.

▷ 원격 배달(Remote Delivery)

모든 메일 메시지의 95%는 리모트 메일 서버로 보내져야 하며, Sink[3] 서버가 체크한다.

▷ 에러율(Error Rate)

만일 메일서버가 예상치 못한 응답이나 응답시간(60초이내)을 받았을 경우 에러로 간주한다. 에러율은 1%이하이어야 한다.

---

3) 싱크 서버(Sink Server) : 메일 서버로 들어온 메일들의 수신자를 살펴 봤을 때, 자신 노드 아닌 경우를 처리하기 위한 메일 서버

5.3 실험 측정 내용

실험은 먼저 실제 메일 시스템에서 메일을 처리하는 데 관여하는 중요한 동작 부분에 관한 메일 처리 시간을 실측하여 그 결과에서 일반식을 도출하는 방법을 사용하였고, 버퍼캐쉬를 위한 디스크 쓰기 시간 측정은 hdparm-5.6으로 측정했으며, 네트워크 부하 측정을 위해서는 TCP socket을 이용했다.

5.3.1 모델링을 위한 메일 시스템 시간 및 네트워크 성능 측정

측정부분은 다음과 같이 총 7개의 메일 시스템 내부의 시간 측정 위치에서 operation thread의 수와 메일의 크기별로 각각 측정하였고, 시뮬레이션에서는 이들에 대한 평균 그래프에 대한 추계선을 엑셀을 이용해 도출하였다.

▷ Sendmail 접속 연결 처리 시간

Sendmail이 SMTP 연결 접속을 시작하고 MDA를 호출하기 전까지의 시간 (그림 10)

▷ MDA 배달 시간

MDA가 메일 데이터를 받아 Node thread에 전달하기 직전까지의 시간 (그림 11)

▷ Node Thread 인식 시간

Node thread가 MDA로부터 첫 번째 데이터 패킷이 도착했음을 인식하기까지의 시간 (그림 12, 13)

▷ Node Thread 데이터 읽기 시간

Node thread가 도착된 데이터를 읽는데 걸리는 시간 (그림14, 15)

▷ Work List Queue 삽입 시간

Node thread가 work list queue에 데이터를 삽입하는데 걸리는 시간 (그림 16, 17)

▷ Work List Queue 대기 시간

데이터가 work list queue에서 인출되기를 대기하고 있는 시간으로 operation thread의 개수 1, 2, 3, 50등으로 변경하면서 측정하였다. (그림 18, 19)

▷ Operation Thread 처리 시간

Operation thread가 데이터를 받아 처리하는 데 걸리는 시간으로 인증 및 버퍼캐쉬에서의 캐쉬 쓰기 시간 등이 포함되어 있다. (그림 20)

▷ 네트워크 데이터 전송 시간

각 노드에 서버 프로그램을 띄운 후 서로간에 TCP 클라이언트 접속을 통해 정해진 크기의 데이터를 보내고 돌아오는데 걸리는 시간을 측정하여, 그 값에 반을 전송시간으로 채택했다. 실험은 1번의 평균값을 얻기 위해 10000번을 수행했다. (그림 21)

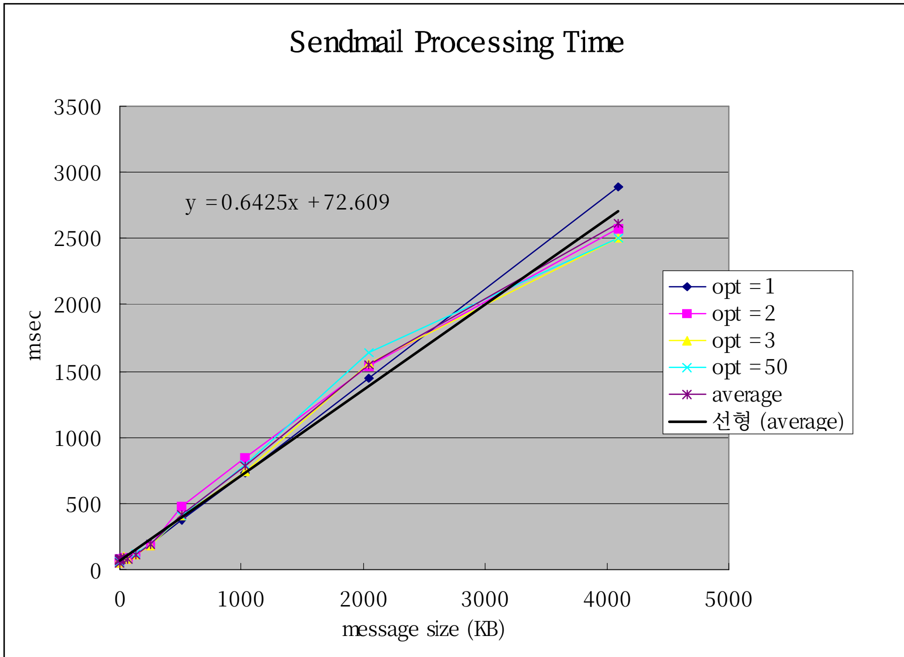


그림 10. senmdail 처리 시간

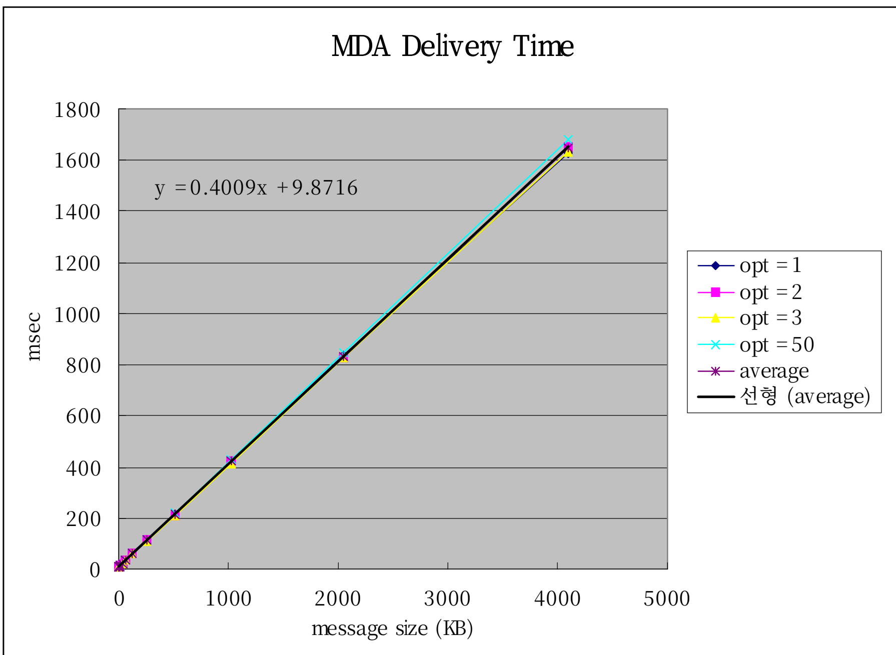


그림 11. MDA 메일 전달 시간

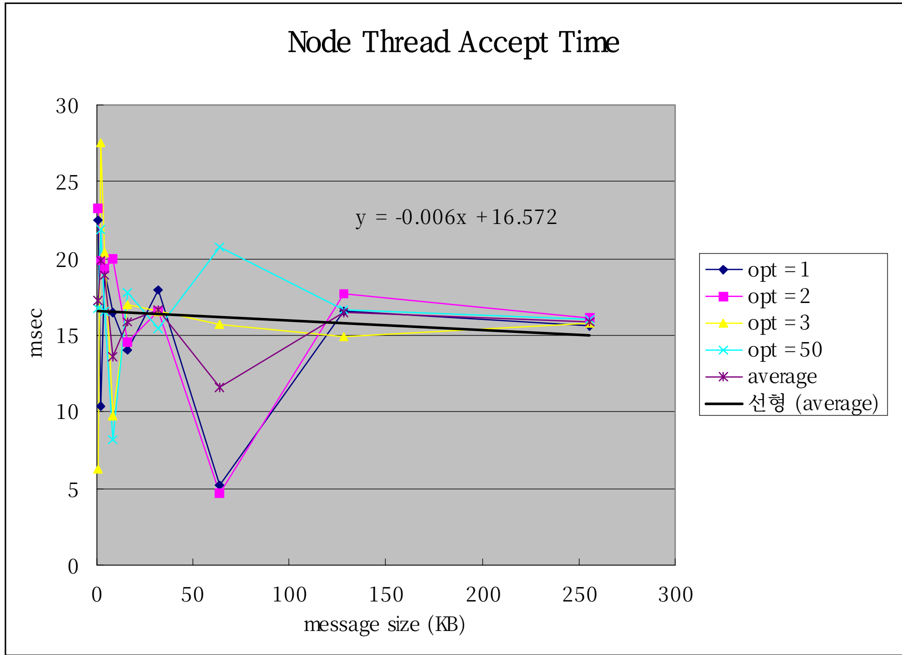


그림 12. node thread accept time (1KB ~ 256KB)

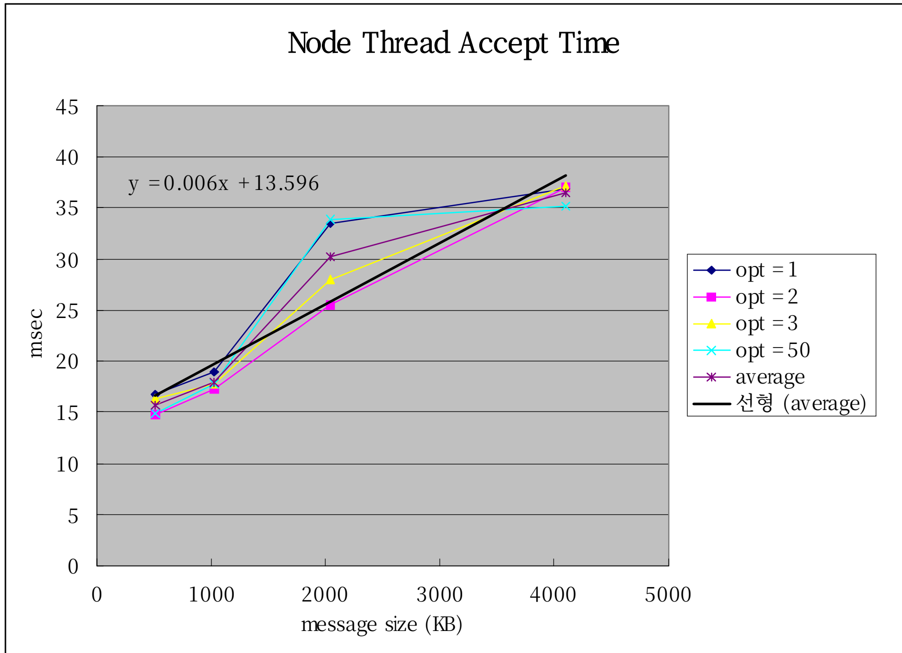


그림 13. node thread accept time (512KB ~ 4096KB)

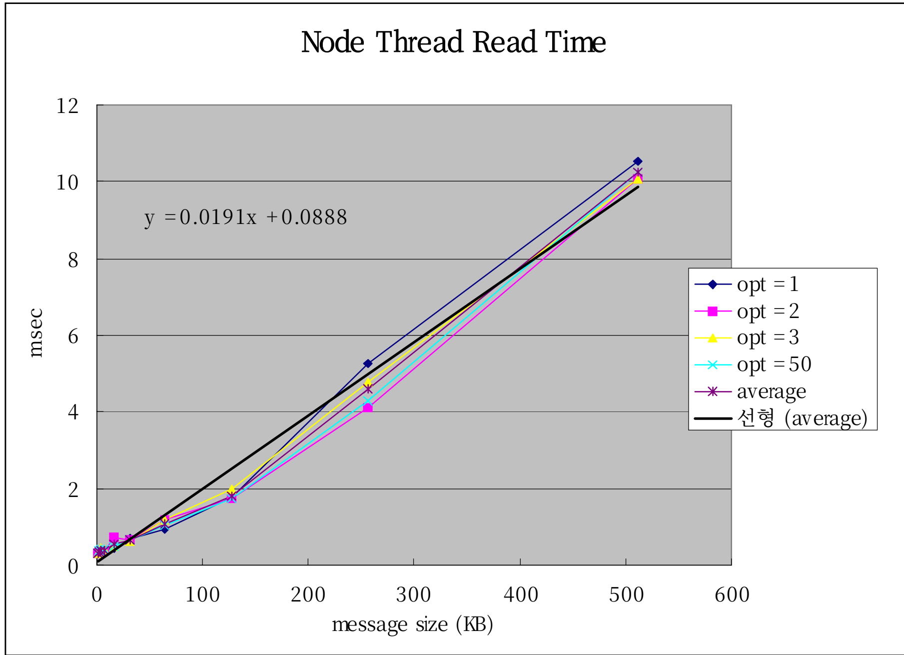


그림 14. node thread read time (1KB ~ 512KB)

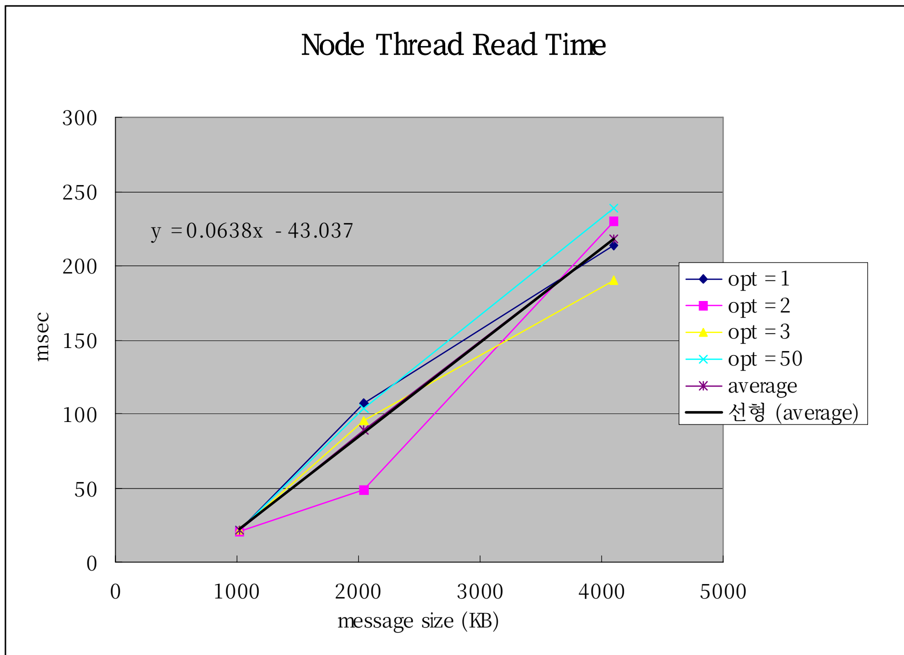


그림 15. node thread read time (1024KB ~ 4096KB)

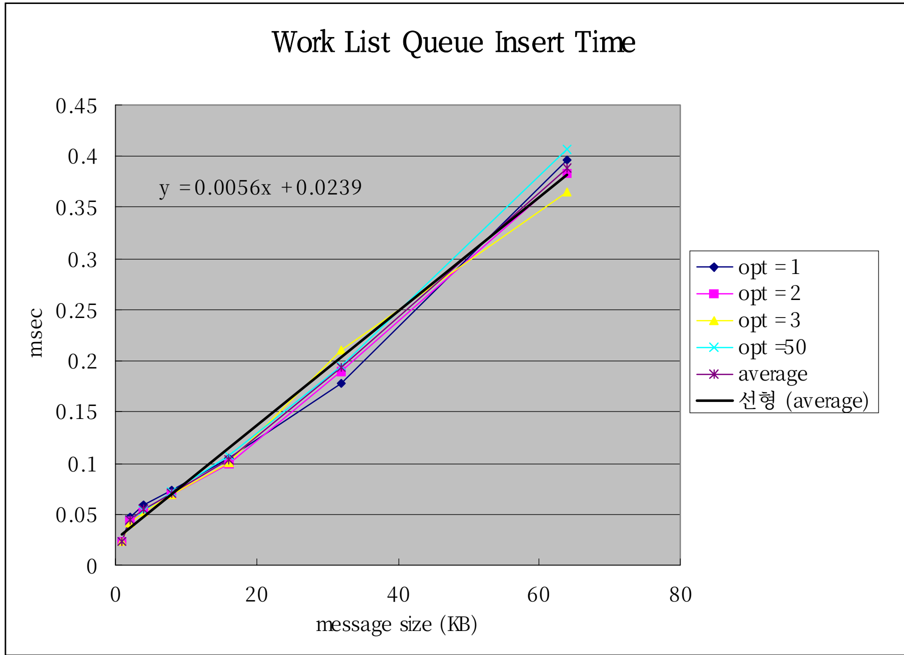


그림 16. work list queue insert time (1KB ~ 64KB)

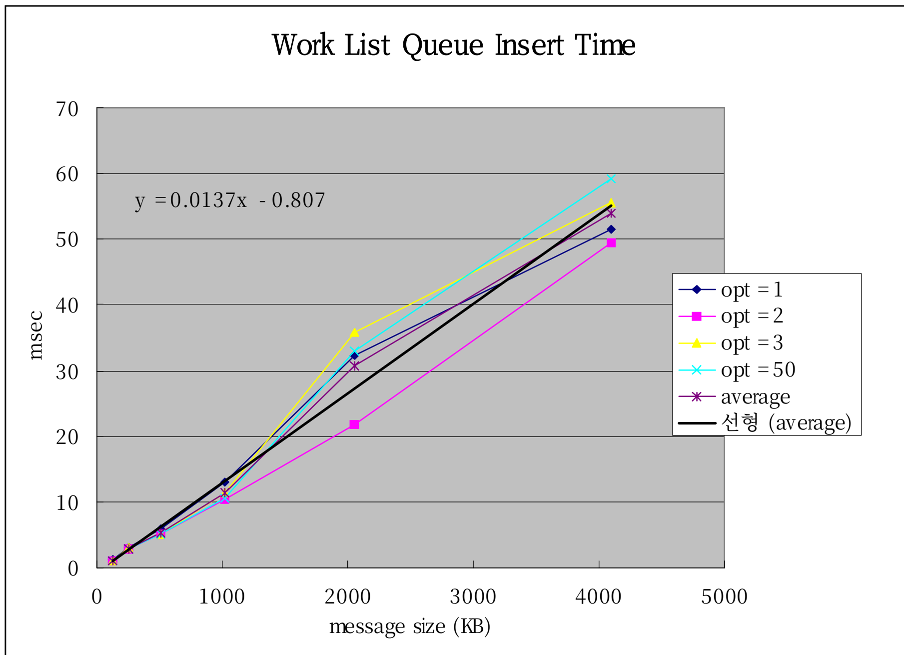


그림 17. work list queue insert time (128KB ~ 4096KB)

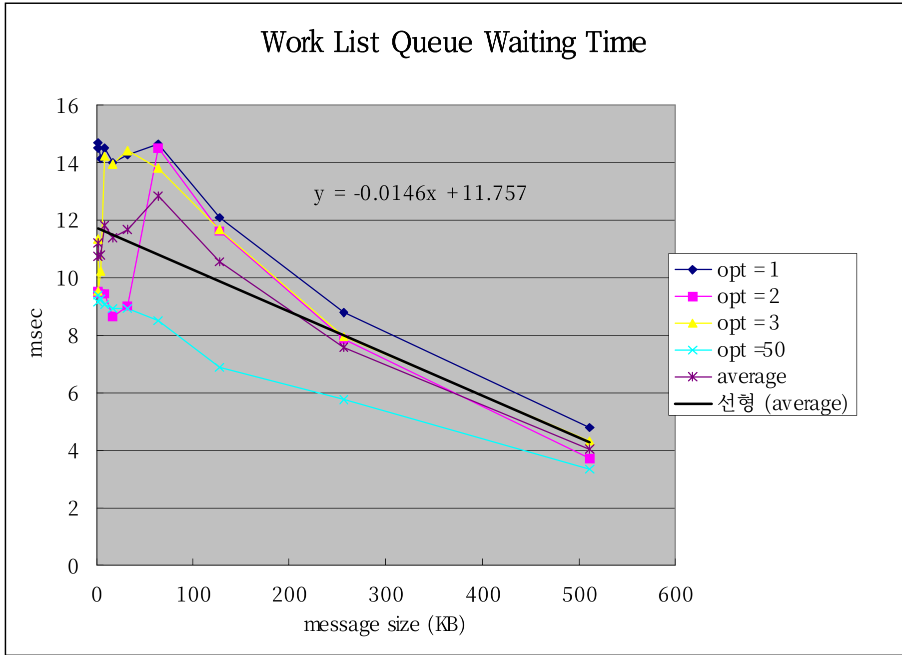


그림 18. work list queue waiting time (1KB ~ 512KB)

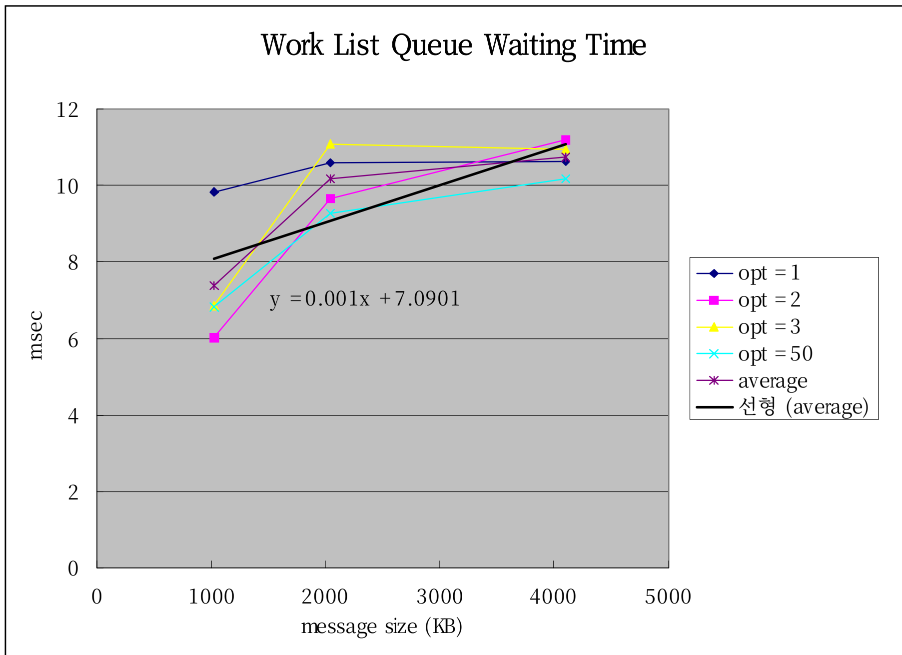


그림 19. work list queue waiting time (1024KB ~ 4096KB)

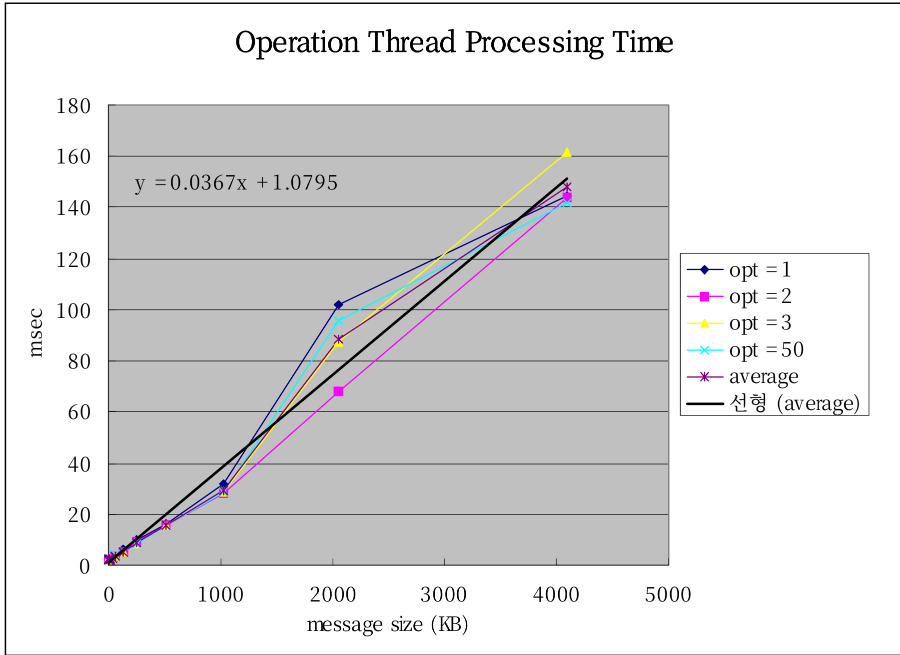


그림 20. operation thread processing time

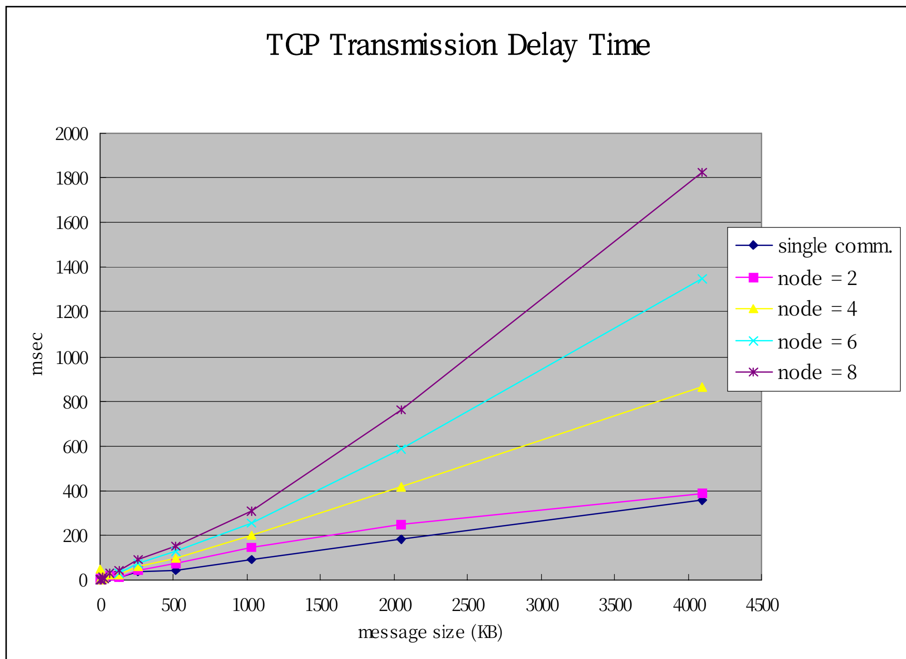


그림 21. 노드별 TCP 데이터 전송 시간

5.3.2 시뮬레이터 실험 결과

시뮬레이션 실험은 운영체제 스케쥴 정책으로 라운드 로빈과 우선순위 기반 정책으로 실험하였고, 입력 부하로써 SPECmail2001의 출력부하 5천, 만, 2만, 4만, 5만, 6만, 8만, 16만, 32만, 64만, 128만명 총 11개의 부하를 사용했으며, 클러스터 노드로는 1개, 2개, 4개, 6개를 사용했다. 네트워크 비중 실험으로 100Mbps를 사용했으며, 비교실험으로 100Mbps에서 나온 시간을 반으로 줄였을 때 throughput과 QoS가 어떤 영향을 미치는지에 대해서 실험하였다.

5.3.2.1 단일 메일 응답시간 비교

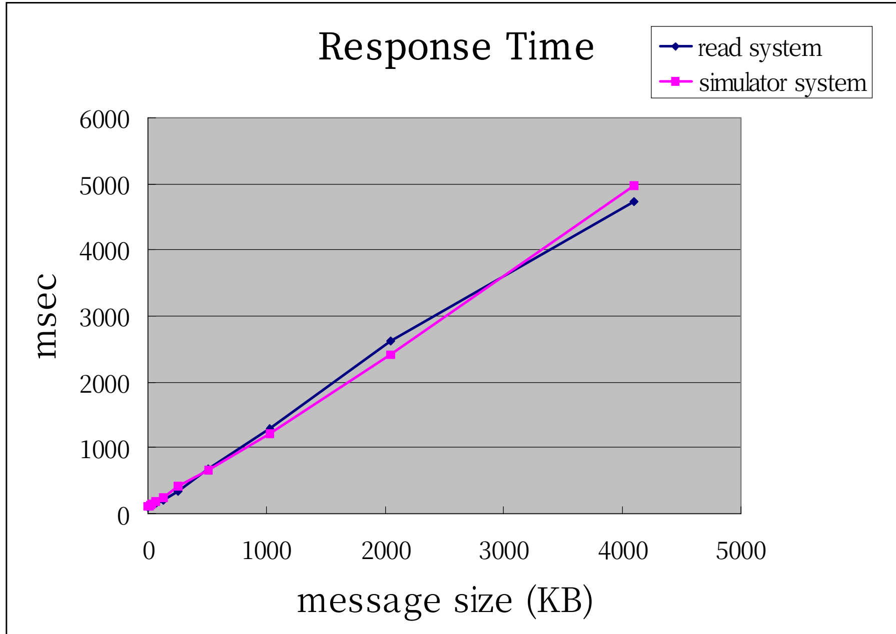
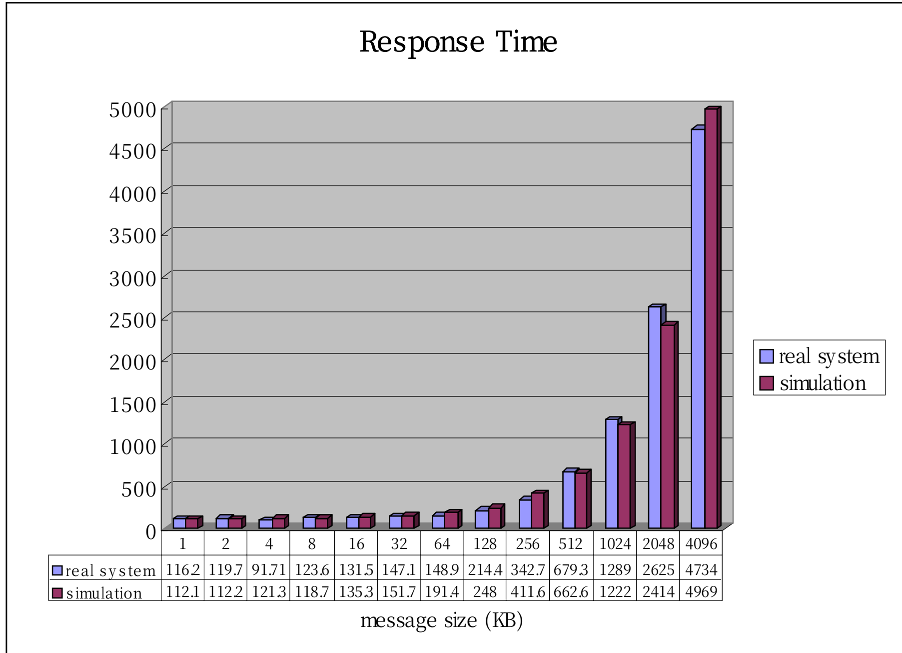


그림 22. 시뮬레이터와 실제 시스템간의 응답시간 비교 (I)

각 메일 크기에 대해서 한 노드에 메일을 보내어 얻어진 실제 시스템 값과 시뮬레이터 시스템과의 응답시간 비교 그래프와 표로써 시뮬레이터가 실제 시스템을 얼마나 잘 반영하고 있는 지를 보여주는데 의미가 있다.

결과를 보면, 실측값과 시뮬레이터의 값이 큰 차이를 보이지 않고 있다. 메일 서버 대 시뮬레이터의 결과를 비교하면 각 메일 크기에 대해서 2048KB일 경우가 최대 108.72%, 4KB일 경우가 최소 75.0%, 전체 메일 크기에 대해서는 평균 95.64%의 정확성을 보였고, 시간 증가의 양상은 메일 크기가 증가하면 응답시간은 지수 함수적으로 증가함을 알 수 있다. 그림22 그래프는 X축과 Y축이 모두 지수 스케일이기 때문에 직선으로 보여졌다.



	1	2	4	8	16	32	64	128	256	512	1024	2048	4096
real system	116.2	119.7	91.71	123.6	131.5	147.1	148.9	214.4	342.7	679.3	1289	2625	4734
simulation	112.1	112.2	121.3	118.7	135.3	151.7	191.4	248	411.6	662.6	1222	2414	4969

그림 23. 시뮬레이터와 실제 시스템간의 응답시간 비교 (II)

5.3.2.2 Throughput 측정

Throughput 측정은 컴퓨터 노드를 1, 2, 4, 6대까지 변경시키면서 SPECmail의 작업부하를 측정한 결과이다. 작업부하는 5000명에 대한 부하부터 시작하여, 128만명까지 시스템을 충분히 포화시킬 수 있을 정도의 부하량을 부여하였다.

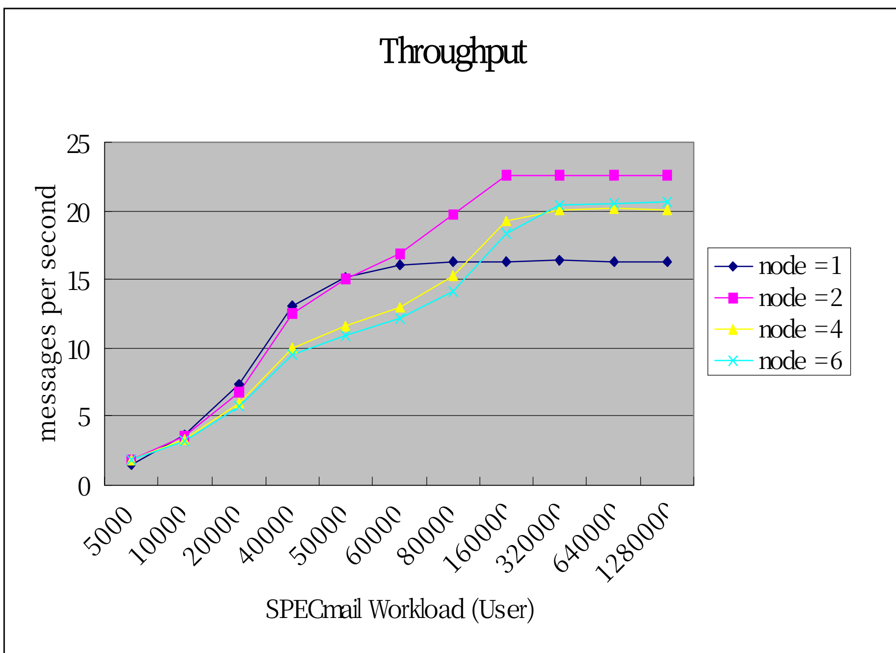


그림 24. 노드수가 1, 2, 4, 6일 때의 throughput 결과 (I)

실험 결과는 노드가 1개일 경우 최대 throughput이 초당 16.35개를 처리할 수 있고, 노드가 2개일 경우 최대 22.69개를 초당 처리할 수 있다. 노드가 4개와 6개인 경우는 각각 초당 20.15, 20.60개로 측정이 되었다. 노드수가 2개일 때 최대 throughput이 되는 이유는 네트워크의 오버헤드가 노드수가 4,6일 때보다 적게 작용했기 때문이다.

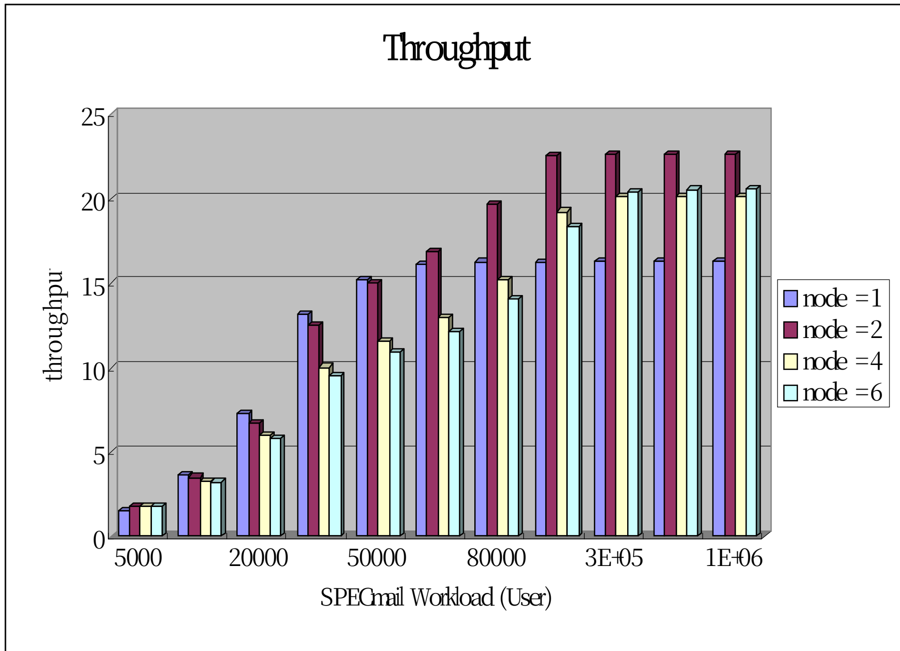


그림 25. 노드수가 1, 2, 4, 6일 때의 throughput 결과 (II)

5.3.2.3 QoS 측정

그림26은 throughput 실험의 결과로써 SPECmail에서 정한 QoS를 얼마나 만족하는지를 보이는지를 나타내는 그래프이다.
SPECmail에서는 QoS 95%를 최하 기준으로 제시하고 있기 때문에 그 기준에 맞춰 본다면, 노드가 1개 일 때 3만 정도(초당 2.5통)의 부하를 견딜 수 있다는 결과를 보여주고 있다.

노드가 2개일 경우, 최대 5만5천명(초당 4.58개), 노드가 4개일 경우 약 12만명(초당 10통), 노드가 6개일 경우 15만명(초당 12.5통) 정도를 안정적으로 처리할 수 있음을 보여주고 있다. 특징적인 것은 노드가 1개에서 2개로 늘어났을 때의 QoS의 향상과 2개에서 4개, 4

개에서 6개로 증가할 때의 QoS 향상이 크지 않는 이유는 노드 수가 늘어날수록 네트워크의 영향이 커지기 때문이라는 것을 확인할 수 있었다.

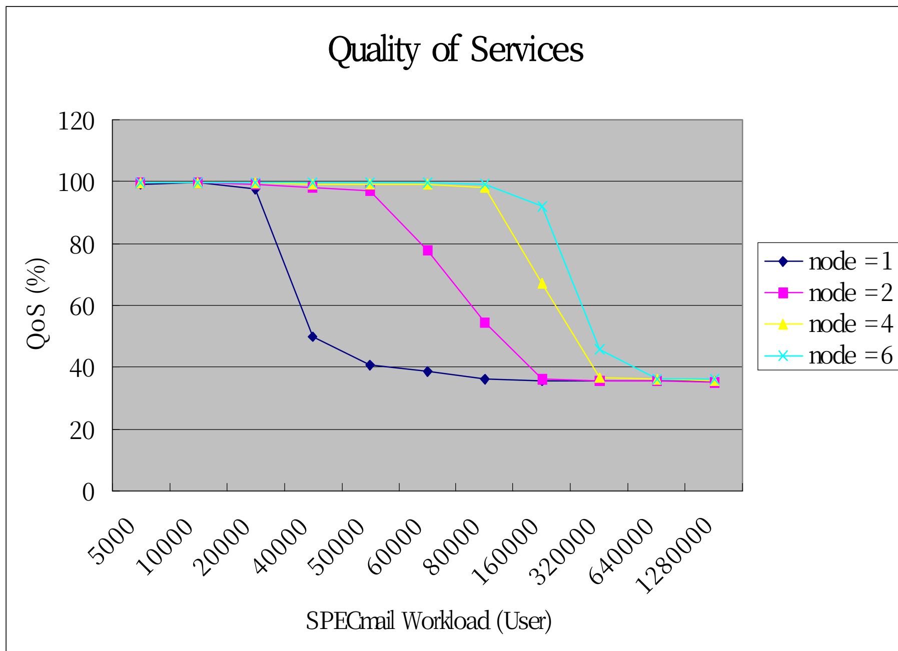


그림 26. 노드수가 1, 2, 4, 6일 때의 QoS 결과 (I)

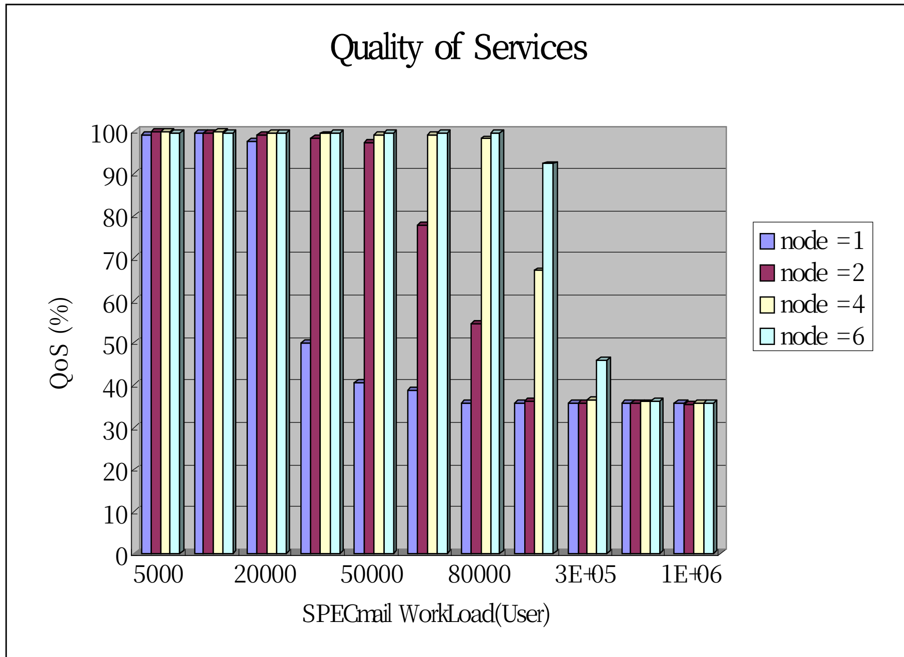


그림 27. 노드수가 1, 2, 4, 6일 때의 QoS의 결과 (II)

5.3.2.4 스케쥴링 정책의 변경 실험

실험은 라운드 로빈 방식과 우선순위 방식을 갖는 스케쥴러로 수행시켰을 때의 throughput과 QoS를 측정한 실험이다. 입력 부하로는 위 실험에서 모든 노드를 포화시킬 수 있는 64만명에 대한 입력을 사용하였고, 노드 수를 바꿔가면서 라운드 로빈과 우선순위 방식의 시뮬레이터를 실험하였다. 우선순위는 전체 응답시간 중 가장 많은 시간을 차지하고 있는 sendmail process, MDA, node thread, operation thread의 순으로 우선순위를 할당하였다.

실험 결과는 두 방식 모두 throughput과 QoS에 별 영향을 미치지 않는 것으로 측정되었다. 이유를 예를 들어 살펴보면, 메일 시스템

에서 특정 한 부분, 즉 sendmail을 스케쥴러가 먼저 처리하면 그만큼 다른 실행되는 부분, 즉 MDA나 node thread의 큐 대기시간을 증가시켜 결국 전체적인 throughput 면에서는 별 이득을 보지 못하는 것으로 분석됐다.

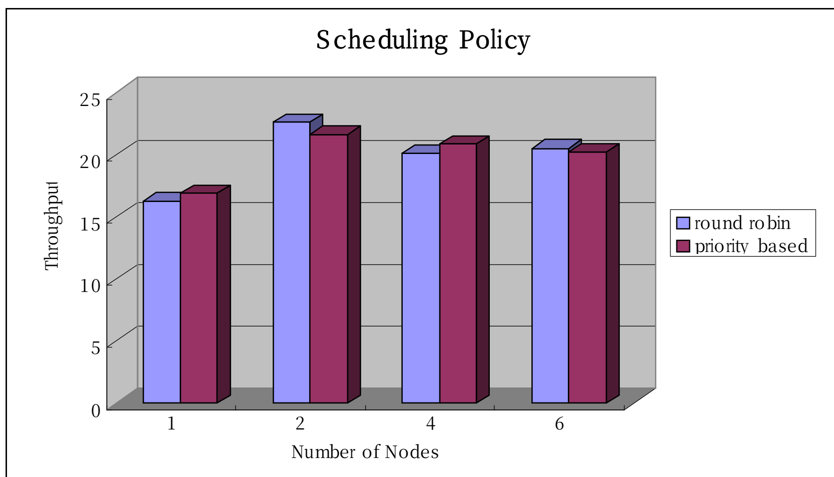


그림 28. 스케쥴링 정책 변경에 따른 throughput 비교

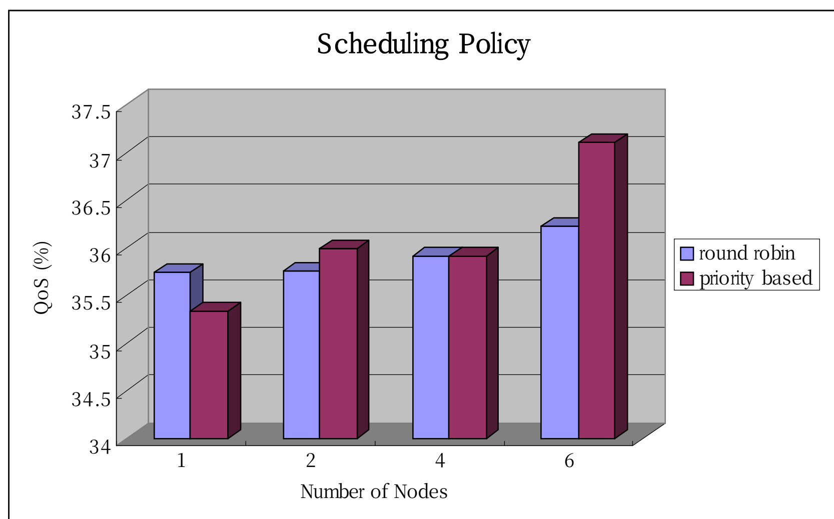


그림 29. 스케쥴링 정책에 따른 QoS 비교

5.3.2.5 네트워크의 영향 실험

5.3.2.4에서 메일 시스템에서는 스케쥴링과 같은 소프트웨어 알고리즘의 변경이 시스템 성능에 별 영향을 미치지 않음을 알 수 있었다. 이번 실험은 노드수를 4개, 스케쥴링 방법을 라운드 로빈 방법으로 고정시킨 후 네트워크의 성능을 변경했을 때 어느 정도의 시스템에 영향을 발생시키는 지를 알아보는 실험이다.

네트워크의 성능으로는 실측 자료인 100Mbps에서 걸린 전송시간과 그 전송시간을 반으로 줄였을 때와의 비교이다. 이 실험 자체는 입력 전송시간을 반으로 줄이는 것이 매우 작위적 것이기 때문에 결과값 자체보다는 시스템의 특성을 알아보는 정도로 삼아 5.3.2.4에서 보인 소프트웨어 알고리즘 변화 결과와 비교하기 위한 하드웨어 성능 변화가 미치는 영향을 살펴보는데 있다.

실험 결과를 QoS에서는 별 영향을 미치지 않았지만, throughput 면에서 보면 상당한 차이가 나는 것을 알 수 있다. 최대 throughput 도 100Mbps에서는 초당 20.1개였는데 100Mbps/2 에서는 최대 33.21개로 약 60.5% 증가한 모습을 보였다. 이런 경향은 노드 수가 커지면 커질수록 네트워크의 영향을 많이 받음을 알 수 있고, 메일 서버와 같은 CPU intensive한 시스템에서는 스케쥴링의 고려보다는 얼마나 빠른 컴퓨터를 사용하는가에 따라 그 처리량이 늘어남을 알 수 있다. QoS 측면에서도 작은 부하량일 경우는 영향이 없지만 큰 부하량 일수록 QoS에 최대 약 55.5% 정도의 성능 향상이 측정되었다.

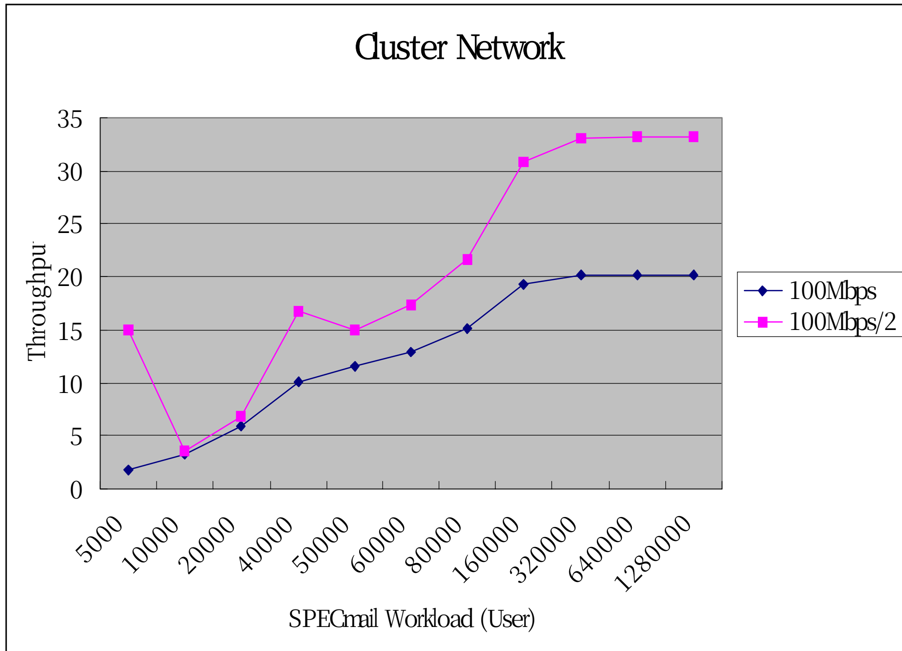


그림 30. 네트워크의 성능 변화와 throughput 변화

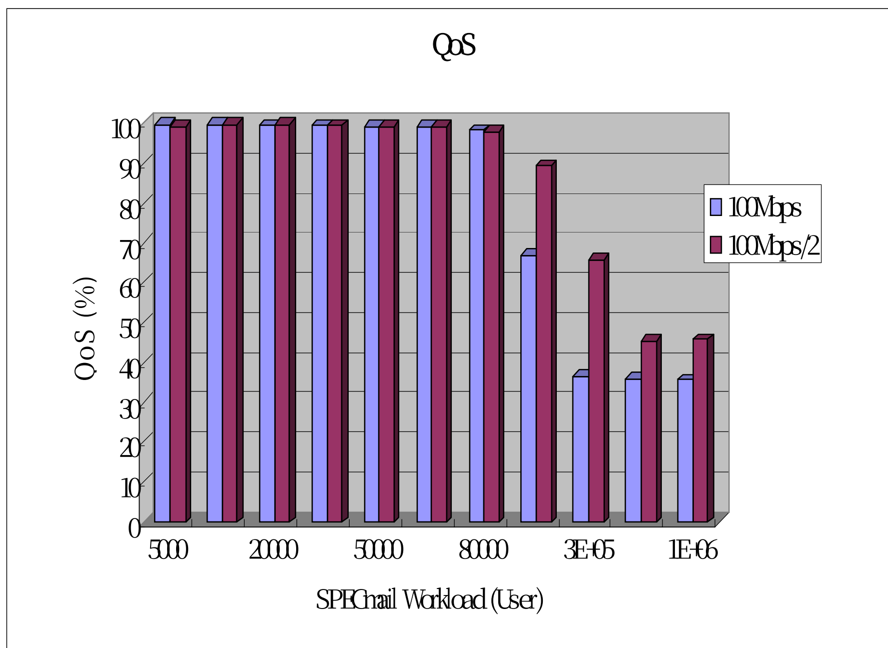


그림 31. 네트워크의 성능 변화와 QoS의 변화

제 6 장 결론 및 향후의 과제

메일 시뮬레이션 결과 throughput과 QoS에 영향을 주는 요인으로 라운드 로빈 방식과 우선 순위 기반의 스케쥴링 정책의 변경과 네트워크 변경을 SPECmail workload와 노드 수를 변경하면서 실험하였다. 실험 결과 소프트웨어 적인 알고리즘의 변화보다는 네트워크 성능 개선과 같은 하드웨어의 성능을 높히는 방법이 더 효과적인 것으로 나타났고, 현 메일 시스템은 SPECmail의 기준으로써 노드1개 일 때 최대 3만명을 안정적으로 처리할 수 있으며, 노드가 2개일 경우 5만5천명, 노드가 6만일 경우 15만명(초당 12.5통) 정도를 안정적으로 처리할 수 있음을 보여주고 있고, 하드웨어 조건이 한정돼 있다면, 노드를 2개이상 사용할 때가 네트워크의 오버헤드를 가장 줄일 수 있는 노드 구성수인 것으로 측정되었다. (그림 24 참고)

향후의 과제로써는 현 시뮬레이터는 스케쥴러 안에서 스래드에 대한 context switching 동작은 모델하고 있지만, switching에 대한 시간 지연 오버헤드는 고려하고 있지 않기 때문에 스래드 개수에 따른 메일 서버의 성능 측정의 신뢰를 할 수 없어 향후 해결해야 할 과제이며 로드 밸런싱의 모델 표현 확장도 향후에 이루어져야할 과제이다. 향후 구현하고자하는 로드 밸런싱은 특정한 시간에 어느 사용자를 어디로 옮기는 방법을 제안할 예정이다. 이 제안은 각 사용자들과 시스템의 정보들을 파라미터로 해서 가장 적은 비용을 가지는 방법을 찾는데 있다.

참 고 문 헌

[1] http://cesdis.gstc.nasa.gov/beowulf

[2] Jonathan B. Postel, "RFC821 SIMPEL MAIL TRANSFER PROTOCOL"

[3] J. Myers and M. Rose, "RFC1939 POST OFFICE PROTOCOL"

[4] David Wood, "Programming Internet Email", O'REILLY

[5] Terry Gray, "Message Access Paradigms and Protocols" Director, Networks & Distributed Computing University of Washington http://www.imap.org/imap.vs.pop.html
Terry Gray, "Comparing Two Approaches to Remote Mailbox Access : IMAP vs. POP", Director, Networks & Distributed Computing University of Washington
http://www.imap.org/imap.vs.pop.brief.html

[6] Gregory F. Pfister, "In Search of Clusters Second Edition", Prentice Hall PTR pp. 71～84

[7] Wan-teh Chang, Soonhoi Ha, Edward A. Lee, "Heterogeneous Simulation - Mixing Discrete-Event Models with Dataflow"

[8] "Discrete Event Modeling in PtolemyII", Lukito Muliadi, eecs.UC,Berkeley

[9] http://ptolemy.eece.berkeley.edu

[10] Bryan Costales with Eric Allman, "sendmail", O'REILLY

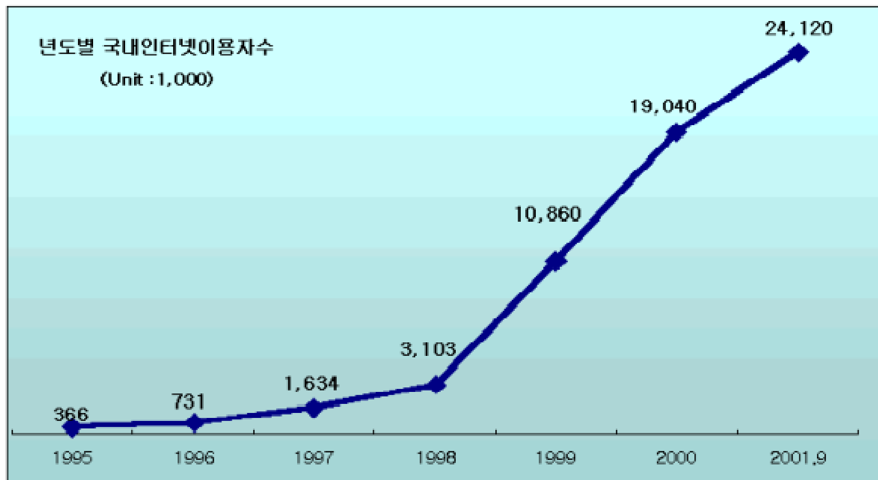
[11] "The Almagest", chapter 12, vol 2. UC. Berkeley

[12] "SPECmail2001 Mail Server Benchmark Architecture White

Paper", SPEC
http://www.spec.org/osg/mail2001/docs/whitepaper.html
[13] Sendmail, Home page, http://www.sendmail.net
[14] Nick Christenson. Tim Bosserman, David Beckemeyer, EarthLink Network, Inc. A High Scalable Electronic Mail Service Using Open Systems, Proceedings of the USENIX Symposium on Internet Technologies and System, Monterey, California, December 1977
[15] 1997년 12월 3일자, 1999년 8월 20일자 전자신문
[16] 한국인터넷정보센터 http://www.nic.or.kr
[17] fujimoto, "Parallel Discrete Event Simulation"

The Number of Internet Users in Korea (By year)

Date : 2001.9.30
Source : KRNIC

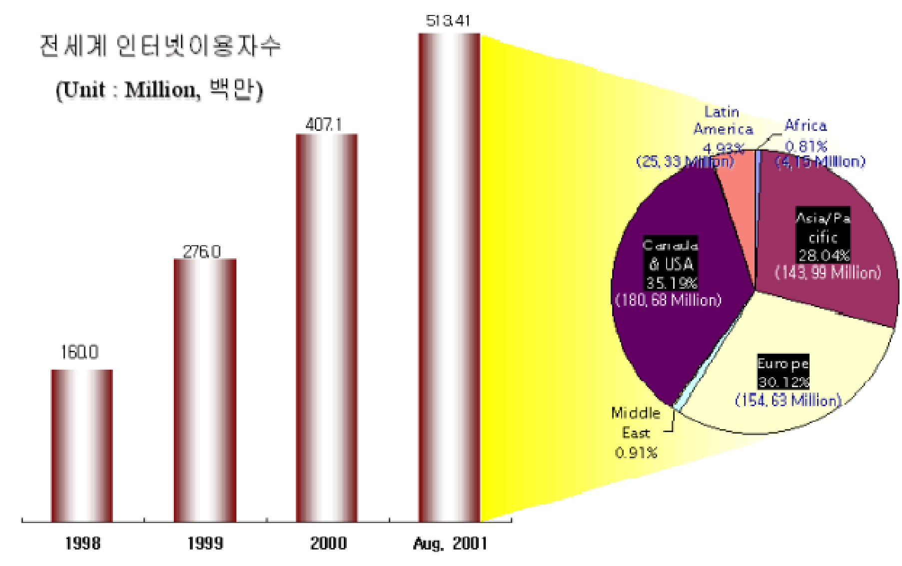


Year	1994	1995	1996	1997	1998	1999	2000	2001.9
Users	138	366	731	1,634	3,103	10,860	19,040	24,120

별첨1. 국내 인터넷 사용자의 연도별 추이

The Number of Internet Users all over the world

Date : 2001.9.30
Source : Nua (www.nua.ie)



별첨 2. 해외 인터넷 사용자의 연도별 추이

Abstract

As Internet user has been increased and the motivation of the production and the consumption has been converged into Internet mail, we need a set of severs that able to process mail message with heavy size steadily. In result the mail server clustering technique is used for an alternative plan in general. Because in 90's, moreover, turn around time of system is faster and system complexity is better complicated then before, we need that necessary of mail server simulator can imitate a behavior of real mail system. The mail server simulator have one or more strengths that not only shortness of system development time but also develop a system with a newly applied algorithm without the change of operating system or reconfiguration physically.

In this paper analysis that a guide lines of performance and characteristics of cluster based real e-mail server system through result of designed simulator using the discrete event(DE) domains in the tool, is called SNU PeaCE, compare the result of simulator with real system's one and predict maximum user of cluster system.

Keywords : Cluster(ing), Mail Server, Simulator(-tion)