무선 인터넷 프록시 서버 클러스터에서 호스트 부하 정보에 기반한 동적 부하 분산 방안

(A Dynamic Load Balancing Scheme based on Host Load Information in a Wireless Internet Proxy Server Cluster)

곽 후 근 † 정 규 식 ††
(Hukeun Kwak) (Kyusik Chung)

요 약 무선 인터넷 프록시 서버 클러스터에서 부하 분산기는 사용자의 요청을 각 서버로 분산시키는 역할을 한다. 리눅스 가상 서버(LVS: Linux Virtual Server)는 소프트웨어적으로 사용되는 부하 분산기로써 여러 가지 스케줄링 방식들을 지원한다. LVS 스케줄링 방식에는 라운드 로빈 방식, 해쉬 기반 방식, 또는 서버와 부하 분산기 사이에서 서버로 연결된 커넥션 개수를 이용하는 방식이 있다. 일부 향상된 방법에서는 각 서버별로 서버의 최고 성능 범위 안에서 허용된 커넥션 개수의 상한값과 하한값을 사전에 결정하여 이를 스케줄링 시에 적용한다. 그러나, 이러한 스케줄링 방법들에서는 서버의 실시간 부하 정보들이 부하 분산에 반영되지 않는다.

본 논문에서는 서버 부하 정보에 기반한 동적 스케줄링 방식을 제안한다. 제안된 방식에서는 부하 분산기가 서버의 실시간 CPU 부하 정보를 바탕으로 가장 적은 부하를 가지는 서버에 새로운 요청을 할당한다. 16대로 구성된 클러스터링 컴퓨터와 정적 컨텐츠(이미지와 HTML)를 가지고 실험을 수행하였다. 실험 결과 CPU를 많이 사용하는 요청과 호스트의 성능이 다른 경우에 대하여 종래의 스케줄링 방식보다 성능이 향상됨을 확인하였다.

키워드 : 서버 부하 분산, 무선 인터넷, 프록시 서버, 클러스터, 동적 부하 분산

Abstract A server load balancer is used to accept and distribute client requests to one of servers in a wireless internet proxy server cluster. LVS(Linux Virtual Server), a software based server load balancer, can support several load balancing algorithms where client requests are distributed to servers in a round robin way, in a hashing-based way or in a way to assign first to the server with the least number of its concurrent connections to LVS. An improved load balancing algorithm to consider server performance was proposed where they check upper and lower limits of concurrent connection numbers to be allowed within each maximum server performance in advance and apply the static limits to load balancing. However, they do not apply run-time server load information dynamically to load balancing.

In this paper, we propose a dynamic load balancing scheme where the load balancer keeps each server CPU load information at run time and assigns a new client request first to the server with the lowest load. Using a cluster consisting of 16 PCs, we performed experiments with static content(image and HTML). Compared to the existing schemes, experimental results show performance improvement in the cases of client requests requiring CPU-intensive processing and a cluster consisting of servers with difference performance.

Key words : Server load balancing, Wireless internet, Proxy server, Cluster, Dynamic Load Balancing

1. 서 론

무선 인터넷에 대한 관심이 증가하는 가운데 핸드폰, PDA등의 무선 인터넷 단말기의 수요가 늘어나며 보편화 되어가고 있다. 그리고 무선 인터넷 서비스도 기존의 정보검색 위주의 간단한 서비스에서 전자 상거래나

· 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음
† 정 회 원 : 숭실대학교 전자공학과 대학원
gobarian@q.ssu.ac.kr
†† 종신회원 : 숭실대학교 정보통신전자공학과 교수
kchung@q.ssu.ac.kr
논문접수 : 2005년 6월 15일
심사완료 : 2006년 2월 1일

멀티미디어 서비스 등의 복잡한 서비스로 사용자들의 욕구가 상승하고 있다. 이에 따라 무선 인터넷 대역폭의 효율적인 사용과 빠른 이용 시간을 위하여 무선 인터넷 프록시 서버의 필요성이 증대되고 있다. 무선 인터넷 프록시 서버는 무선 사용자를 유선 인터넷 서버에 연결 시켜주는 역할을 한다. 그림 1은 무선 인터넷에 사용되는 무선 인터넷 프록시 서버를 보여주고 있다.

무선 인터넷 프록시 서버는 급증하는 사용자의 요청에 대한 확장성(Scalability)을 보장하기 위해 클러스터링 구조를 가진다. 무선 인터넷 프록시 서버 클러스터는 여러 대의 서버를 하나의 서버처럼 동작하도록 연결함으로써 고성능 및 고가용성의 효과를 가진다. 무선 인터넷 프록시 서버를 클러스터링 하기 위해서는 서버들을 하나로 묶어주고 스케줄링을 해주는 하드웨어적인 스위치[1] 혹은 소프트웨어적인 방법[2]이 필요하다.

본 논문에서는 기존의 스케줄링 방식이 가지는 문제점을 분석하고, 이를 해결할 새로운 스케줄링 방식을 제안한다. 기존 스케줄링 방법에서는 단순한 알고리즘을 사용하거나 (예, Round-Robin, hashing) 또는 네트워크 단에서 사용가능한 정보(예, 서버로 연결된 커넥션 수)를 이용한다. 일부는 서버 성능에 관하여 사전에 확인된 정보를 이용하여 스케줄링시 가중치를 달리 주거나 혹은 서버별 상한값, 하한값을 두고 있다. 지금까지 서버의 부하정보를 실시간으로 반영하는 스케줄링 방식은 소개되지 않았다. 본 논문에서는 서버의 실시간 부하정보를 이용하여 동적으로 스케줄링하는 방식을 제안한다. 본 논문의 구성은 다음과 같다. 제 2장에서는 무선 인터넷 프록시 서버에서 기존의 스케줄링 방식과 그 문제점을 소개한다. 3장에서는 기존 스케줄링 방식의 문제점을 해결하는 새로운 스케줄링 방식을 설명하고, 4장에서는 실험 및 토론을, 5장에서는 결론 및 향후 연구 방향을 제시한다.

2. 기존 연구

2.1 LVS 스케줄링

리눅스 가상 서버(LVS: Linux Virtual Server)[2]는 리눅스를 기반으로 독립된 여러 서버들을 하나의 클러스터로 구성하여 뛰어난 확장성과 가용성을 제공한다. 가상 서버의 구조는 서버 외부에서는 마치 클러스터 서버가 하나의 고성능 서버인 것처럼 보이도록 하고, 실제 서버 내부에서는 사용자의 요청을 스케줄링을 이용하여 처리한다. 그림 2는 리눅스 가상 서버의 전체적인 구조를 보여준다.

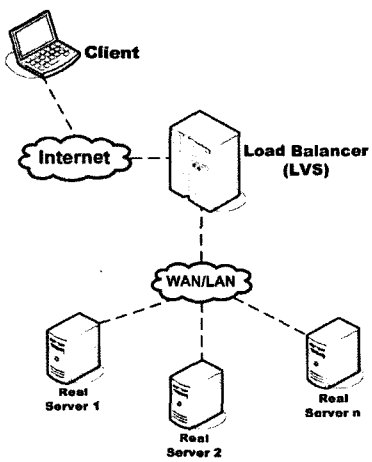


그림 2 리눅스 가상 서버의 구조

LVS는 사용자(Client)로부터 받은 요청을 처리하는 방식(Packet Forwarding Methods)에 따라 세 가지 방식(Network Address Translation, Direct Routing, IP Tunneling)[2]이 존재한다. 이러한 방식은 서버가 처리한 요청을 LVS를 통해 사용자에게 전송하는 방식(Network Address Translation)과 서버가 직접 사용자

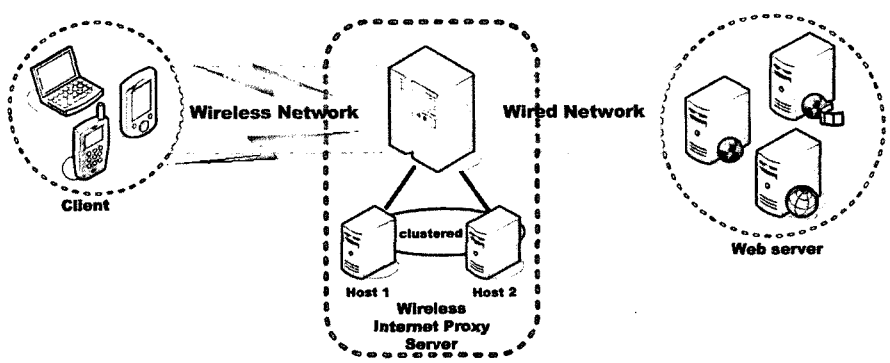


그림 1 무선 인터넷 프록시 서버

에게 보내는 방식(Direct Routing, IP Tunneling)으로 나눌 수 있다.

사용자 요청을 실제 서버들로 스케줄링 하는 방식에 따라 8가지 방식(RR: Round-Robin, WRR: Weighted Round-Robin, LC: Least Connection, WLC: Weighted Least Connection, LBLC: Locality-Based Least Connection, LBLCR: Locality-Based Least Connection with Replication, SH: Source Hash, DH: Destination Hash)[2]이 존재한다. 스케줄링 방식은 실제 서버들로 요청을 분산할 때, 순서대로 분산하는 방식(RR, WRR), 가상 서버와 실제 서버와의 커넥션 개수를 이용해서 분산하는 방식(LC, WLC), 실제 서버들을 몇 개의 단위로 묶어서 분산하는 방식(LBLC, LBLCR)과 해시를 이용하는 방식(SH, DH)으로 나눌 수 있다.

(1) RR(Round-Robin)과 WRR(Weighted Round-Robin)

LVS는 클라이언트로부터 오는 요청을 순서대로 서로 다른 서버로 보낸다. 이 방식은 모든 서버의 성능이 동일하다고 가정되며, LVS는 별도의 연산 과정 없이 순서대로 서버에 요청을 보내면 된다. 그림 3은 RR 알고리즘의 동작을 보여 주고 있다. WRR은 RR 방식의 일종으로 서버의 처리 용량이 다를 때, 각 서버의 처리 용량에 비례하는 가중치를 두어 요청을 분산한다. 즉, 기본적으로 요청을 분산할 때 RR 방식을 사용하되, 가중치가 큰 서버에는 더 많은 요청을 보내게 된다. 예를 들어, 실제 서버를 A, B, C라고 하고, 이들의 가중치가 각각 4, 3, 2라면, 요청에 대한 서버 할당 순서는 AABABCABC가 된다.

(2) LC(Least Connection)와 WLC(Weighted Least Connection)

LVS는 사용자의 요청이 왔을 때 현재 각 서버에 요청이 연결된 개수를 측정하여 가장 적은 커넥션 개수를 가지는 서버로 요청을 분산한다. 이때, 각 서버는 동일

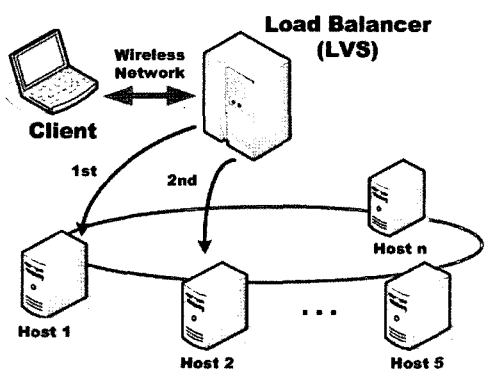


그림 3 RR

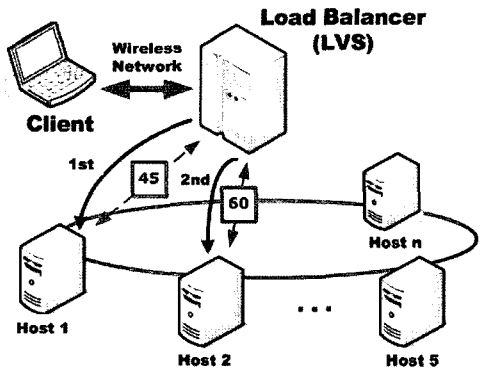


그림 4 LC

한 성능의 처리 용량을 가지고 있음을 가정한다. 그림 4는 LC 알고리즘을 보여주고 있다. WLC는 LC 방식의 일종으로 각 서버의 처리 용량이 다른 경우에 사용된다. 각 서버의 처리 용량에 따라 가중치(Weight)를 할당하고 이 가중치에 따라 요청을 할당한다. 기본적으로 LC 방식으로 동작하되, 가중치가 큰 서버로 더 많은 요청을 할당 하게 된다.

(3) LBLC(Locality-Based Least Connection)와 LBLCR (Locality-Based Least Connection with Replication)

LBLC 알고리즘은 일단 요청이 들어오면 부하가 적은 서버로 요청을 할당한다. 이 수신지(Destination) IP를 가지는 요청은 매번 이 서버로 요청을 보낼 것이다. 그러나 만약 이 서버가 과부하(커넥션 개수가 자신의 가중치 보다 높은 경우)가 되면, 다른 서버들 중에서 자신의 가중치의 1/2보다 적은 부하를 가진 서버를 선택하고 요청을 이 서버에 할당한다. 이 수신지(Destination) IP를 가지고 또 다른 요청이 오면 예전 서버가 아닌 이 서버로 요청을 할당하게 된다. 그림 5는 LBLC 알고리즘을 보여주고 있다. LBLCR은 LBLC의 일종으로 다음과 같은 차이를 가진다. LBLC는 하나의 수신지(Destination) IP에 대해 하나의 서버를 할당하였지만 LBLCR은 서버 그룹(Set)을 할당한다. 같은 수신지 IP는 동일 서버 그룹으로 요청이 할당되고, 서버 그룹 내에서는 가장 커넥션 개수가 적은 서버로 요청이 할당된다. 만약, 서버 그룹 내의 모든 서버가 과부하라면, 서버 그룹을 제외한 다른 서버들 중 가장 커넥션 개수가 적은 서버를 이 서버 그룹에 포함한다. 일정 시간 동안 서버 그룹에 변화가 없다면(부하를 그대로 유지하고 있다면), 서버 그룹 내에서 가장 부하가 높은 서버를 서버 그룹에서 삭제한다.

(4) DH(Destination Hashing)와 SH(Source Hashing)

LVS는 수신지(Destination) IP 주소를 가지고 Static

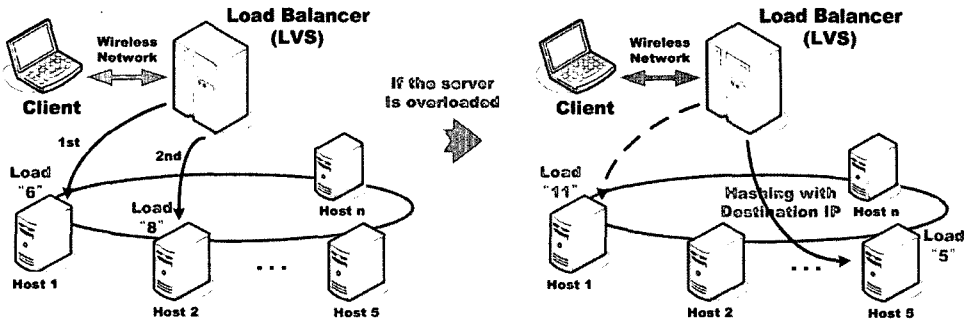


그림 5 LBLC

Hashing 테이블을 통해 매핑된 서버로 요청을 보낸다. Hashing을 이용하게 되면 같은 수신지 IP 주소에 대해 같은 서버로 요청을 보낼 수 있고, 이때 서버에 캐시를 사용하고 있다면 전체적인 응답 시간을 감소시킬 수 있다. 그림 6은 DH 알고리즘을 보여주고 있다. SH는 수신지(Destination) IP 대신 송신지(Source) IP를 사용한다는 것을 제외하고 DH 알고리즘과 동일하다.

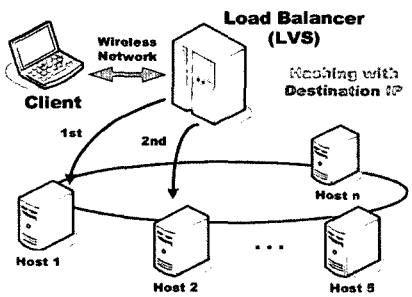


그림 6 DH

2.2 동적 스케줄링(Dynamic Scheduling)

동적 스케줄링[3]은 기존 LVS 스케줄링의 단점을 보완하기 위해 제안된 스케줄링 알고리즘이다. 제안된 알고리즘은 서버(호스트)의 동시 커넥션 개수를 이용하여 부하를 분산하는 방식으로 기존 방식과는 달리 각 서버별로 서버 성능을 고려하여 커넥션 개수의 상한값과 하한값을 사전에 결정하여 이를 스케줄링시 이용한다. 특정 서버의 커넥션 개수가 그 상한값보다 높게 되면 그 서버는 스케줄링에서 제외되고 그 커넥션 개수가 그 하한치보다 낮게 되면 스케줄링에 포함되게 된다. 제안된 알고리즘에서 사용되는 Throttling Mechanism[4]은 상한계(Upper Bound, Unhealthy)와 하한계(Lower Bound, Healthy)를 두고 작업(Task)을 처리하는 방식으로 스케줄링에 이를 적용하게 되면 상한계를 넘어선 서버에게는 더 이상 작업을 할당하지 않고 하한계로 떨어지면 다시 작업을 할당하는 방식으로 동작하게 된다. 스케줄링의 가장 중요한 요소인 상한계와 하한계는 사전 실험을 통해 각 서버가 처리할 수 있는 동시 커넥션 개수 및 사용자의 요청을 처리할 수 없는 시점을 기준으로 설정하였다.

동적 스케줄링 방식은 LVS의 스케줄링 방식 중 WRR의 변형으로 볼 수 있다. 즉, 각 서버가 상한계와 하한계에 도달했을 때 이 정보를 LVS로 보내게 되고, LVS는 각 서버의 상태에 따라 가중치(Weight)를 다시 계산하여 업데이트 한다. 이때 사용자의 요청은 업데이트된 각 서버의 가중치에 따라 분산되는 방식으로 동작한다. 이들의 알고리즘을 정리하면 표 1과 같다. 동적 스케줄

표 1 가중치 테이블 업데이트 알고리즘 (동적 스케줄링)

```
switch signal: // 실제 서버가 보낸 부하 정보가
  case unhealthy: // 과부하라면(unhealthy)
    recalculate weight values without the real server // 실제 서버에 대한 가중치를 다시 계산한다
    Weight_value[real_server] = 0 // 부하 정보를 보낸 실제 서버를 가중치를 0으로 할당한다
    Health_flag[real_server] = FALSE // 부하 정보를 보낸 실제 서버의 Health 플래그를 FALSE로 할당한다
    if (all of Health_flags are FALSE) // 모든 Health 플래그가 FALSE라면
      reset_table() // 디폴트 값으로 가중치 테이블을 초기화한다.
  case healthy: // 과부하가 아니라면(healthy)
    recalculate weight values including the real_server // 실제 서버에 대한 가중치를 다시 계산한다
    Heal_flag[real_server] = TRUE // 부하 정보를 보낸 실제 서버의 Health 플래그를 TRUE로 할당한다
```

링의 알고리즘을 살펴보면 실제 서버들의 가중치를 업데이트할 때 사전에 결정된 실제 서버의 상한계(Unhealthy) 및 하한계(Healthy)의 정보만을 이용할 뿐 실시간으로 변하는 부하 정보를 이용하지 않음을 볼 수 있다.

2.3 접근 방식

본 절에서는 기존 LVS 스케줄링 방법의 문제점과 동적 스케줄링의 문제점을 정리하고, 3장에서는 이를 해결하는 새로운 스케줄링 알고리즘을 제안한다.

(1) LVS 스케줄링의 단점

LVS는 8가지 스케줄링 알고리즘을 가지고 있으나 서버의 처리 능력 혹은 사용자의 요청 컨텐츠가 다른 경우 이를 반영하지 못하는 단점을 가진다. LVS 스케줄링 알고리즘이 가지는 단점을 정리하면 표 2와 같다.

(2) 동적 스케줄링의 단점

동적 스케줄링은 기존의 LVS 스케줄링 방식의 단점을 보완하기 위해 상한계와 하한계를 두어 서버의 처리 능력과 사용자의 서로 다른 요청을 스케줄링에 반영하였으나 다음과 같은 단점을 가진다.

- 스케줄링시 서버의 성능은 반영되나 서버의 실시간 부하 정보는 반영되지 않는다.
- 사전 실험을 통하여 상한계와 하한계를 설정해주어야 한다.
- 상한계와 하한계는 사전 실험으로 고정되어 있어 서버의 처리 능력이 바뀔 때마다 이를 다시 수정해주어야 한다.
- 사용자가 서로 다른 컨텐츠를 요청하는 경우 상한계와 하한계가 고정되어 있어 이를 반영하지 못한다.

(3) 본 연구의 접근 방식

본 논문에서는 LVS 스케줄링의 단점과 동적 스케줄링의 단점을 보완하는 새로운 알고리즘을 제안한다. 제안된 알고리즘은 호스트의 실시간 부하 정보를 이용하여 동적으로 스케줄링하는데 이는 사전 실험 없이 서버의 처리 능력과 사용자의 요청 컨텐츠가 다른 경우를 능동적으로 반영하는 방식이다.

3. 호스트 부하 정보에 기반한 동적 부하 분산 방안

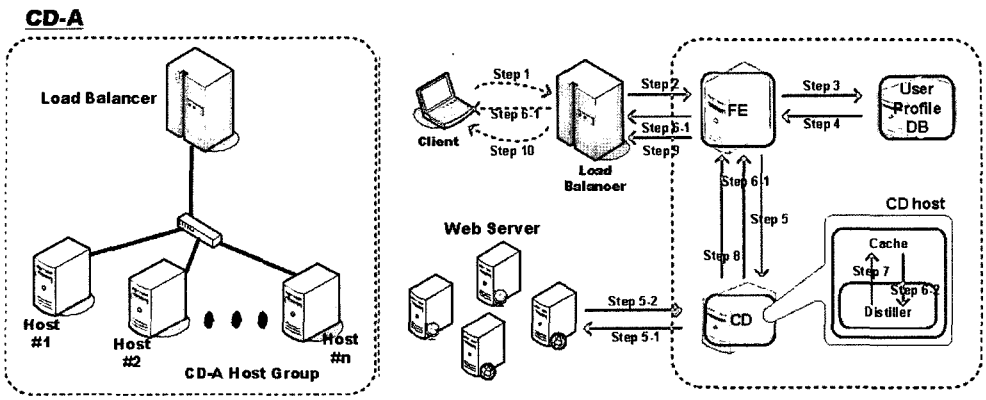
3.1 무선 인터넷 프록시 서버

무선 인터넷 프록시 서버는 무선 인터넷의 낮은 대역폭 해결하기 위해 캐싱(Caching)[5]과 압축(Distillation)[6]을 사용하며, 대용량 트래픽에 대한 확장성(Scalability)이 고려되어야 한다. TranSend[7]는 대용량 트래픽에 대한 확장성을 고려하여 클러스터링으로 구현된 무선 프록시 서버이다. 본 논문에서는 무선 인터넷 프록시 서버 클러스터인 TranSend를 확장성과 구조적인 관점에서 개선한 CD-A(CD & All-in-one)구조[8]를 사용하였다. CD-A 구조는 부하 분산을 위한 LVS와 CD-A 호스트들로 구성되며 호스트의 각 모듈로서 FE와 CD(Cache & Distiller)가 구성된다. LVS는 클라이언트의 요청을 받아서 각 호스트들에게 전달하는 역할을 담당하며, FE는 호스트 내에서 클라이언트의 요청에 대한 외부 인터페이스를 담당한다. CD는 클라이언트의 요청을 처리하는 Cache와 데이타에 대한 압축을 수행하는 Distiller의 역할을 함께 수행한다.

그림 7은 CD-A 무선 인터넷 프록시 서버의 구조와 동작 과정을 보여준다. LVS가 스케줄링 알고리즘을 통하여 클라이언트의 요청을 각 호스트에 전달하게 되면, 호스트 내의 FE는 그 요청을 CD에 전달한다. CD는 해당 데이타가 Cache에 존재하면 FE에 전달하고, 존재하지 않으면 웹 서버로부터 데이타를 요청하여 얻어온 후 압축과정을 거쳐 FE에 전달하고 FE는 그 데이타를 클라이언트에 보내는 방법으로 동작한다.

표 2 LVS 스케줄링 알고리즘의 단점

스케줄링	단 점
RR	서버의 처리 능력이 동일하거나 사용자의 요청 컨텐츠가 동일할 때 유용하나 만약 서버의 처리 능력이나 요청 컨텐츠가 다른 경우 이를 반영하지 못한다.
WRR	서버의 처리 능력에 따라 가중치를 줄 수 있다는 장점을 가진다. 그러나 가중치는 사용자의 요청 컨텐츠에 따라서 유동적으로 바뀔 수 있으나 WRR은 이를 반영하지 못한다.
LC	서버 커넥션 개수를 이용하여 부하 분산을 하여 서버의 처리 능력의 일부를 부하 분산에 반영하나 사용자의 요청 컨텐츠가 다른 경우 이를 반영하지 못한다.
WLC	서버의 처리 능력에 따라 가중치 및 커넥션 개수를 사용한다는 장점을 가진다. 그러나 가중치는 사용자의 요청 컨텐츠에 따라서 유동적으로 바뀔 수 있지만 WLC는 이를 반영하지 못한다.
LBLC	DH와 WLC를 결합한 방식으로 동작함으로 이들의 단점을 그대로 가진다.
LBLCR	서버들이 그룹화 되어 있다는 점을 제외하고 DH와 WLC를 결합한 방식으로 동작함으로 이들의 단점을 그대로 가진다.
DH	사용자가 요청한 주소를 기반으로 서버를 선택함으로, 서버의 처리 능력 혹은 사용자의 요청이 다른 경우 이를 반영하지 못한다.
SH	사용자의 주소를 기반으로 서버를 선택함으로, 서버의 처리 능력 혹은 사용자의 요청이 다른 경우 이를 반영하지 못한다.



(a) 구조 (b) 동작 과정

그림 7 CD-A 무선 인터넷 프록시 서버

3.2 구조

제안된 스케줄링 알고리즘의 구조는 2가지의 새로운 모듈을 필요로 한다. 하나는 호스트의 부하 정보를 부하 분산기로 보내는 모듈로서 각 호스트에 설치되어 동작한다. 나머지 하나는 부하 분산기에서 동작하는 모듈로 각 호스트로부터의 부하 정보를 받아 가중치를 다시 계산하여 업데이트한다.

그림 8은 제안된 스케줄링 알고리즘의 전체적인 구조를 보여준다. 호스트는 주기적으로 자신의 부하 정보를 부하 분산기에 보내고, 부하 분산기를 각 호스트의 부하 정보를 이용하여 가중치를 다시 계산한다. 클라이언트가 부하 분산기로 요청을 보내면, 부하 분산기를 실시간으로 업데이트된 가중치 테이블에 따라 요청을 분산한다.

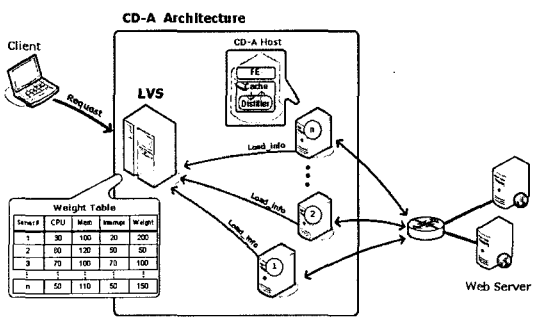


그림 8 호스트 부하 정보에 기반한 동적 부하 분산 방안 (DS-HLI)

3.3 동작 과정

호스트 부하 정보에 기반한 동적 부하 분산 방안의 구체적인 동작 과정은 다음과 같다.

단계 1 : 서버(호스트)들은 LVS에게 주기적으로 자신의 부하 정보를 보낸다.

단계 2 : LVS는 서버(호스트)들로부터 받은 부하 정보를 기반으로 각각의 서버들에 대한 가중치(Weight) 테이블을 갱신한다.

단계 3 : 사용자(Client)가 무선 인터넷 프록시 서버 클러스터(CD-A)로 컨텐츠를 요청한다.

단계 4 : 사용자 요청을 받은 LVS(외부 인터페이스)는 현재 각각의 서버들에 대한 가중치 테이블을 검사하여 가장 많은 가중치를 가지는 서버로 요청을 분산한다(WLC).

그림 9는 무선 인터넷 프록시 서버 내에서 전체적인 사용자 요청 처리 순서를 보여주며 각 단계를 요약하면 다음과 같다.

단계 1 : 서버(호스트)들은 LVS에게 주기적으로 자신의 부하 정보를 보내고 LVS는 서버(호스트)들로부터 받은 부하 정보를 기반으로 각각의 서버들에 대한 가중치(Weight) 테이블을 갱신한다.

단계 2 : 사용자(Client)가 무선 인터넷 프록시 서버 클러스터(CD-A)로 컨텐츠를 요청한다.

단계 3 : 사용자 요청을 받은 LVS(외부 인터페이스)는 현재 각각의 서버들에 대한 가중치 테이블을 검사하여 가장 많은 가중치를 가지는 서버로 요청을 분산한다.

단계 4 : 사용자 요청을 받은 서버 내 FE는 요청한 컨텐츠를 캐시에 요청한다.

단계 5 : 캐시에 요청된 컨텐츠가 있고 압축된 형태라면 바로 FE로 응답하고, 압축된 형태가 아니라면 압축기로 보낸다. 만약, 요청한 데이타가 없다면 외부 웹 서버로 컨텐츠를 요청하고 이를 캐시에 저장 후 압축기로 보낸다.

단계 6 : 캐시는 압축된 데이타를 저장하고 이를 FE로 보낸다.

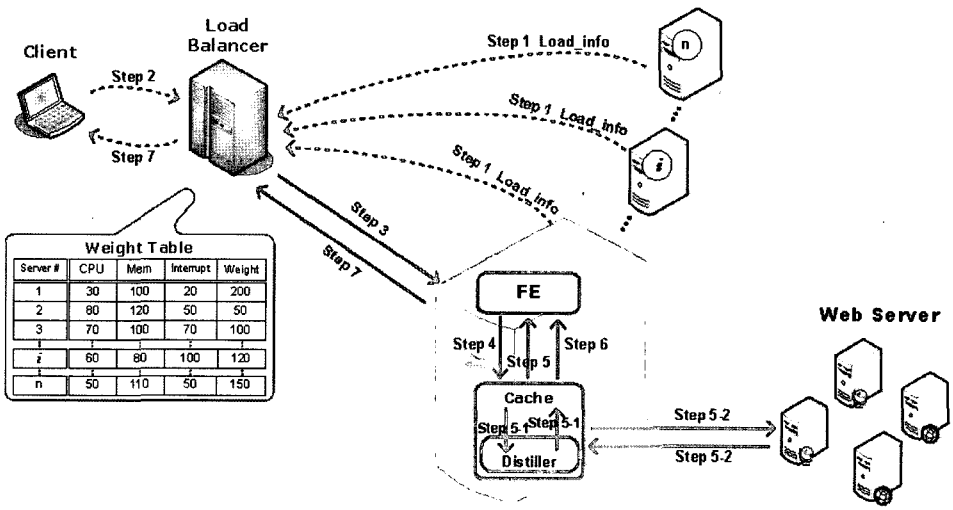


그림 9 무선 인터넷 프록시 서버 내에서 전체적인 사용자 요청 처리 순서

단계 7 : FE는 압축된 데이타를 LVS로 보내고, LVS는 FE로부터 받은 데이타를 사용자에게 요청에 대한 응답으로 보낸다.

3.4 비교

표 3에서는 LVS의 8가지 스케줄링, 동적 스케줄링(DS) 및 제안된 방식(DS-HLI)을 가중치와 서버 선택 방법 관점에서 비교하였다. 기존 스케줄링에서는 서버의 처리 능력을 반영하여 가중치를 적용할 수 있으나, 한번 설정을 하면 부하 분산 중에는 설정값을 바꿀 수 없다는 단점을 가진다. 즉, 요청을 처리하는 서버들의 처리 능력이 실시간으로 변하는 것을 반영할 수 없다. 반면, 제안된 스케줄링 방법에서는 서버(호스트)의 부하 정보를 반영하여 가중치를 실시간으로 업데이트함으로써 서버들 사이로 부하를 균일하게 분산할 수 있다.

4. 실험 및 토론

4.1 실험 환경

표 4는 실험에 사용된 하드웨어와 소프트웨어를 나타낸다. 무선 인터넷 프록시 서버 클러스터는 PC 16대로 구성된 CD-A 구조를 사용하였으며, 클러스터내의 부하 분산기로는 LVS를 사용하였다. 사용자가 AB(Apache Bench)[11]라는 프로그램을 수행하여 무선 인터넷 프록

표 3 기존 스케줄링 vs. 제안된 스케줄링

스케줄링	가중치	서버 선택 방법
RR	X	순차적으로 선택
WRR	O (고정)	서버 처리 용량에 따른 가중치를 고려하여 순차적으로 선택
LC	X	서버 커넥션 개수가 가장 적은 서버를 선택
WLC	O (고정)	서버 처리 용량에 따른 가중치를 고려하여 서버 커넥션 개수가 가장 적은 서버를 선택
LBLC	O (고정)	DH와 WLC를 이용하여 서버 선택
LBLCR	O (고정)	DH와 WLC를 이용하여 서버 선택
DH	X	사용자 요청 주소의 해쉬값을 이용하여 선택
SH	X	사용자 주소의 해쉬값을 이용하여 선택
DS	O (고정)	WRR의 변형으로 서버의 커넥션 개수를 이용하여 선택
DS-HLI	O (가변)	실시간 부하 정보를 이용하여 가중치를 지속적으로 업데이트. 가중치를 이용한 서버 선택시에는 WLC를 사용

표 4 실험용 하드웨어 & 소프트웨어

		하드웨어		소프트웨어	개수
		CPU (Hz)	RAM (MB)		
사용자		P-IV 2.26 G	256	AB[9]	1
LVS		P-IV 2.4 G	512	NAT[2]	1
서버	캐시	P-II 400 M	256	Squid[10]	16
	압축기			JPEG-6b[11]	

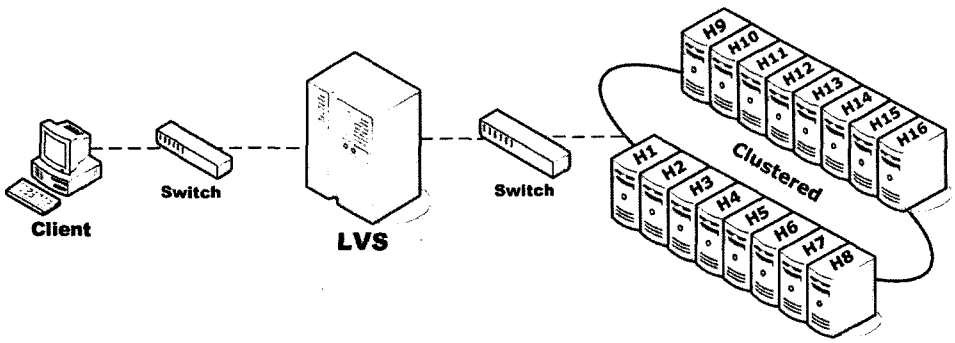


그림 10 실험에 사용된 구성도

시 서버에 컨텐츠(이미지)를 요청하는 방식으로 실험을 수행하였다. LVS는 NAT 방식을 사용하여 서버가 처리한 요청을 LVS를 통해 사용자에게 전송하도록 하였으며, 서버 내 캐시와 압축기는 오픈 소스인 *Squid*와 JPEG-6b 라이브러리를 각각 사용하였다. 표 4에서 사용자와 LVS가 서버보다 하드웨어 성능이 좋은 이유는 스케줄링 실험을 할 때 사용자와 LVS에서는 병목 현상이 발생하지 않는 상황에서 무선 인터넷 프록시 서버 클러스터 내 서버들 사이의 스케줄링을 확인하고자 했기 때문이다.

그림 10은 실험에 사용된 구성도를 보여준다.

4.2 실험 방법

표 5는 실험에 사용된 변수들을 정리한 것이다. 사용자의 요청 개수는 약 200초 동안 프록시 서버가 처리할 수 있는 최대 개수를 사용하였다. 요청 시간을 200초 이상으로 하면 전체적인 실험 결과에는 영향을 미치지 않는다는 것을 확인하였고, 전체적인 실험 시간을 고려하여 200초로 제한하였다. 사용자의 요청 컨텐츠는 웹에서 가장 많은 사용 빈도를 가지는 JPEG 이미지[12]와 HTML을 사용하였으며 요청 크기는 300 Bytes에서 100 Kbytes 사이의 이미지 및 1 Bytes에서 1 MBytes 사이의 HTML을 사용하였다. 요청의 다양화를 위해 Variation Kbytes를 사용하였고, 같은 이름(vari.jpg)으로 1 Kbytes에서 10 Kbytes까지 10개의 서로 다른 크기의 이미지를 저장하고 사용자가 이를 요청하도록 하였다[13]. 이렇게 요청을 하게 되면 같은 동일 서버 내에서 동일 이름으로 서로 다른 크기의 이미지를 요청하는 효과를 가지게 된다.

사용자가 설정할 수 있는 압축기내에서의 압축 정도는 중간 레벨로 하였고, 웹 서버는 캐시 서버 자체에 둠으로써 스케줄링 알고리즘의 성능 평가에 초점을 맞추도록 하였다. 스케줄링 알고리리즘은 LVS의 3가지 방법(RR, LC, DH)과 제안된 방법(DS-HLI)을 비교하였다. WRR, WLC는 실험에 사용된 서버의 성능이 같고 그 결과가 RR, LC와 동일하여 실험을 수행하지 않았고, LBLC와 LBLCR은 DH의 변형이고 요청이 균일하게 분포함으로 그 결과가 DH와 동일하여 실험에서 제외하였다. 또한 SH는 소스 주소가 변한다는 것을 제외하고 DH와 동일함으로 실험을 수행하지 않았다. DS 알고리즘은 상한계와 하한계를 이용하는 WRR의 변형으로, 실험에 사용된 서버의 성능이 동일함으로 실험을 수행하지 않았다.

실험에 사용한 사용자의 요청 방식을 정리하면 표 6과 같다. 가중치는 실제 웹 요청 분포와 유사하게 작은 크기의 JPEG과 HTML에는 큰 가중치를 할당하고, 큰 크기의 JPEG과 HTML에는 작은 가중치를 할당하였다. 각 크기에 따라 할당된 가중치를 정리하면 표 7과 같다. 호스트 별 성능을 다르게 하기 위해 부하 발생기를 짝수 번째 호스트(2, 4, ..., 16)에 사용하였다. 부하 발생기는 1-10초 사이에서 랜덤하게 CPU를 사용하고, 1-10초 사이에서 랜덤하게 CPU를 사용하지 않도록 하였다.

각 호스트에서 자신의 부하 정보를 업데이트 하는 주기는 20 초로 하였다. 호스트 16대를 기준으로 모든 요

표 5 실험에 사용된 변수

사용자의 요청 개수	◦약 200초 동안 프록시 서버가 처리할 수 있는 최대 개수
요청 컨텐츠	◦JPEG, HTML
요청 크기	◦JPEG : 300 Bytes, 1 K, 10 K, 100 Kbytes, Variation(1-10 Kbytes) ◦HTML : 1, 100 Bytes, 1 K, 10 K, 100 Kbytes, 1 Mbytes
사용자 정보(Preference)	◦압축 정도(이미지 해상도) = 중간
웹 서버	◦캐시 서버 자체에 둠 (스케줄링 알고리즘의 성능 평가에 초점을 맞춤)
스케줄링 알고리즘	◦4가지 : RR, LC, DH, DS-HLI

표 6 실험에 사용된 변수

요청 방식	컨텐츠 크기	가중치 (각 크기별 분포)	호스트가 처리하는 컨텐츠	호스트 별 성능
방식1-JPEG	동일	동일	동일	동일
방식2-JPEG	틀림	동일	동일	동일
방식3-JPEG	틀림	틀림	틀림	동일
방식4-JPEG	틀림	틀림	틀림	틀림
방식5-HTML	틀림	틀림	틀림	동일
방식6-JPEG-HTML	틀림	틀림	틀림	동일
방식7-HTML	동일	동일	동일	동일
방식8-HTML	동일	동일	동일	틀림

표 7 각 크기에 따라 할당된 가중치

JPEG	HTML	가중치(%)
300 Bytes	100 Bytes	35
1 Kbytes	1 Kbytes	50
Variation (1-10) Kbytes	10 Kbytes	14
10 Kbytes	100 Kbytes	0.9
100 Kbytes	1 Mbytes	0.1

청 크기에서 주기를 0.1초에서 30초까지 바꾸어 가며 테스트를 하였고 그 결과는 표 8과 같다. 각 주기간의 성능 비교는 LVS 스케줄링 알고리즘 중 각 이미지 요청에서 성능이 가장 좋은 알고리즘을 기준으로 하였다. 동일 컨텐츠(300, 1K, 10K, 100K, Variation bytes)를 요청하는 경우와 다른 컨텐츠(Diff bytes)를 요청하는 경우에서 각 주기의 결과를 LC와 비교하여 가장 높은 성능 향상률을 가지는 업데이트 주기를 실험에 사용하였고, 표 8(b)를 보면 제안된 스케줄링 방법은 20초에서 가장 좋은 성능을 나타냄을 알 수 있다.

업데이트 주기 외에 고려할 변수는 호스트에서 부하 정보를 보낼 때 현재의 부하 정보를 보낼지 혹은 평균값을 보낼 지에 대해서 생각해 볼 수 있다. 표 9는 20초의 업데이트 주기를 가지고 현재 부하를 보내는 경우와 평균 부하를 보내는 경우를 고려한 실험 결과이다. 업데이트 주기와 마찬가지로 각 이미지에 대한 요청에서 평균값과 현재값을 사용했을 때의 결과를 LC와 비교하여 가장 높은 성능을 가지는 값으로 실험을 수행하였다. 표 9의 결과를 보면 평균값을 사용한 경우(156.28%)가 현재값을 사용한 경우(118.81%)에 비해 성능 향상 폭이 큼을 알 수 있다.

무선 인터넷 프록시 서버 클러스터(CD-A)를 이용한 스케줄링 알고리즘 실험 방법을 정리하면 다음과 같다. 실험은 아래의 6 단계를 4개의 스케줄링 알고리즘에 반복 적용하였다.

1. 무선 인터넷 프록시 서버(CD-A) 1대를 구성한다.
2. 실험에 사용할 스케줄링 알고리즘을 세팅한다.
3. AB(Apache Bench)를 이용하여 서버로 JPEG 이미지를 약 200초 동안 요청한다.
4. 초당 처리된 요청 개수를 측정한다.

표 8 업데이트 주기 (호스트 16대 기준)

(a) 업데이트 주기별 초당 처리된 요청수

업데이트 주기	300 bytes	1 Kbytes	10 Kbytes	100 Kbytes	Variation	Diff bytes
0.1	3102	2361	542	38.21	980	1293
1	2784	2213	451	38.22	744	1186
5	3981	2341	391	38.37	664	1197
10	4196	2975	570	38.37	679	1438
20	4661	3330	428	38.29	992	1456
30	4591	3448	398	38.33	974	1444

(b) 업데이트 주기별 향상률

(기준 : RR - Variation, LC - 300, 1 K, 100 K, Diff bytes, DH - 10 Kbytes)

업데이트 주기	300 bytes	1 Kbytes	10 Kbytes	100 Kbytes	Variation	Diff bytes	Total
0.1	-33.12	-33.13	-6.74	-1.07	-4.52	125.99	7.90
1	-39.97	-37.32	-22.41	-1.04	-27.48	107.34	-3.48
5	-14.17	-33.70	-32.68	-0.66	-35.28	109.24	-1.21
10	-9.53	-15.75	-1.83	-0.65	-33.87	151.39	14.96
20	0.50	-5.70	-26.26	-0.87	-3.35	154.49	19.80
30	-1.02	-2.34	-31.47	-0.75	-5.07	152.49	18.64

표 9 부하 정보 비교 (현재와 평균)

요청 이미지	평균값 (Req/s)	현재값 (Req/s)	LC (Req/s)	평균값 vs. LC (%)	현재값 vs. LC (%)
300 Bytes	4516	4661	4638	-2.62	0.50
1 Kbytes	3454	3330	3531	-2.20	-5.70
10 Kbytes	579	428	581	-0.39	-26.26
100 Kbytes	38.13	38.29	38.62	-1.28	-0.87
Variation	983	992	1026	-4.17	-3.35
Diff Bytes	1527	1456	572	166.95	154.49

5. 서버를 1대 추가한다.
6. 실험에 사용된 서버의 수(16대)만큼 3~5 과정을 반복한다.

4.3 실험 결과

(1) 방식1-JPEG

표 10과 그림 11은 사용자가 방식1-JPEG으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 동일 컨텐츠의 요청은 같은 크기의 이미지를 요청하는 방식으로 실험하였다. 서버를 추가할 때 마다 초당 처리된 요청수가 선형적으로(Linear) 증가하는 것을 볼 수 있고 같은 처리 능력을 가지는 서버들에게 동일 컨텐츠를 요청하는 경우임으로 실험에 사용된 모든 스케줄링 방법이 비슷한 성능을 보임을 알 수 있다.

표 10 호스트 개수에 따른 초당 처리된 요청 수

(a) 300 Bytes

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	316	596	895	1188	1446	1732	2025	2285	2598	2890	3212	3474	3750	4034	4315	4603
LC	314	599	888	1155	1489	1778	2042	2279	2628	2944	3204	3550	3796	4115	4413	4638
DH	315	593	889	1175	1471	1769	2062	2316	2601	2908	3178	3505	3770	4113	4388	4580
DS-HLI	319	559	825	1119	1388	1649	1916	2298	2512	2805	3149	3342	3720	4045	4272	4516

(b) 1 Kbytes

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	233	444	664	882	1090	1282	1516	1732	1921	2137	2353	2577	2799	3026	3244	3458
LC	229	443	662	886	1099	1301	1508	1709	1978	2178	2348	2617	2849	3092	3305	3531
DH	228	439	662	874	1095	1318	1539	1729	1948	2174	2388	2628	2826	3098	3292	3454
DS-HLI	236	443	658	877	1052	1302	1492	1649	1943	2149	2369	2557	2824	3006	3256	3454

(c) 10 Kbytes

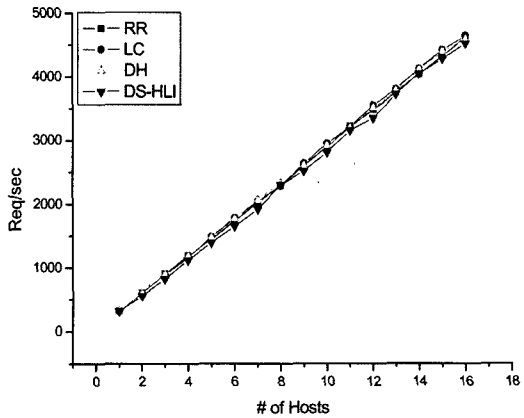
Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	36	71	110	143	179	215	251	287	322	358	393	429	465	501	539	574
LC	37	72	109	142	178	219	252	292	327	364	401	432	468	505	542	574
DH	36	71	108	144	180	211	250	290	325	361	398	433	471	505	547	581
DS-HLI	37	71	107	144	173	203	248	288	316	351	386	425	459	501	538	579

(d) 100 Kbytes

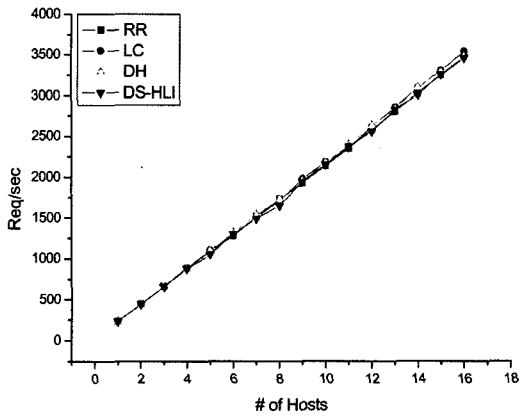
Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	2.27	4.78	7.16	9.57	11.94	14.29	16.76	19.15	21.58	23.97	26.37	28.78	31.16	33.59	35.99	38.46
LC	2.40	4.81	7.15	9.65	12.05	14.37	16.83	19.31	21.60	24.05	26.48	28.92	31.30	33.75	36.18	38.62
DH	2.42	4.80	7.23	9.62	12.01	14.42	16.83	19.26	21.69	24.07	26.50	28.89	31.38	33.76	36.18	38.58
DS-HLI	2.52	4.81	7.15	9.70	11.91	14.43	16.53	19.05	21.46	23.68	26.23	28.72	31.04	33.37	35.83	38.13

(e) Variation

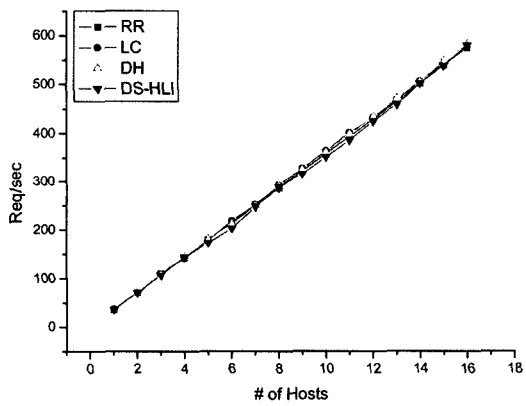
Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	66	129	193	258	323	386	451	513	578	646	709	772	834	898	961	1026
LC	65	127	192	256	321	382	446	512	580	634	699	770	827	900	958	1025
DH	65	126	192	253	319	384	447	511	571	635	698	763	830	888	959	1017
DS-HLI	65	127	190	250	320	374	446	496	555	629	684	754	822	860	931	983



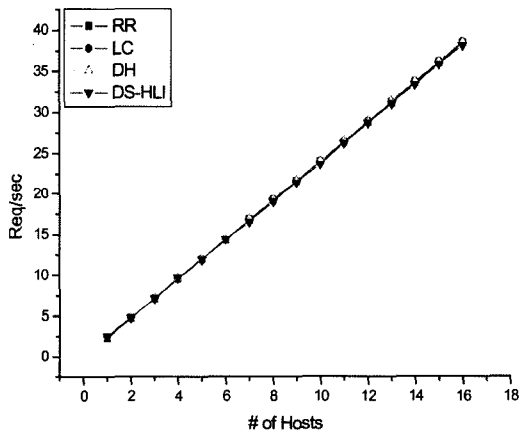
(a) 300 Bytes



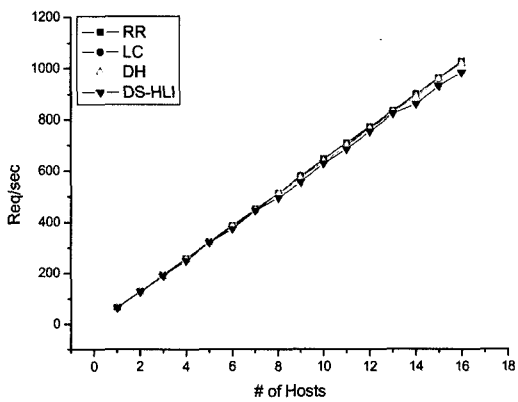
(b) 1 Bytes



(c) 10 Kbytes



(d) 100 Kbytes



(e) Variation

그림 11 호스트 개수에 따른 초당 처리된 요청 수

(2) 방식2-JPEG

실험에 사용된 사용자(Apache Bench)는 동일 이름의 컨텐츠만을 요청할 수 있음으로 사용자가 다른 컨텐츠를 요청한 경우를 실험하기 위해 다음과 같은 방법을 사용하였다. 1번 호스트에는 300 bytes를, 2-5번 호스트에는 1K, 10K, 100K, Variation bytes를 diff.jpg로 이미지를 만들고, 사용자가 diff.jpg를 요청하도록 하였다. 나머지 호스트들도 같은 작업을 반복하였다. 사용자가

표 11 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	317	462	116	16	19	24	15	15	20	35	36	38	42	41	44	47
LC	320	522	463	210	252	395	614	484	289	386	483	720	707	514	532	572
DH	317	469	114	16	20	25	15	15	20	35	35	40	44	42	43	47
DS-HLI	320	524	483	477	500	640	845	822	732	793	1080	1339	1295	1292	1395	1527

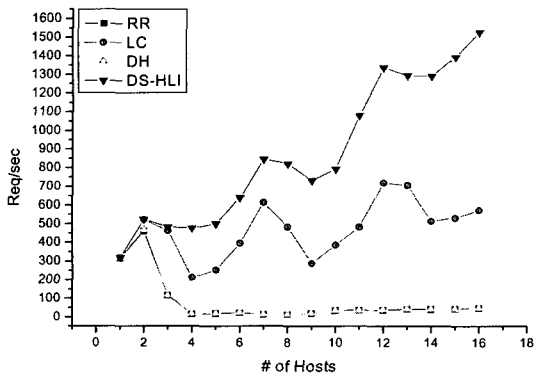


그림 12 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

diff.jpg를 요청하면 각 호스트마다 diff.jpg의 크기가 틀림으로 서로 다른 컨텐츠를 요청한 경우로 고려될 수 있다. 표 11과 그림 12는 사용자가 방식2-JPEG으로 서로 다른 컨텐츠를 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다.

사용자가 서로 다른 컨텐츠를 요청한 경우에는 RR, DH가 가장 낮은 성능을 보이고, 제안된 방법이 가장 좋은 성능을 보임을 알 수 있다.

- RR, DH : 사용자의 요청 컨텐츠에 상관없이 요청을 분산함으로 최종 초당 처리된 요청수는 100 Kbytes를 처리하는 서버에 종속됨을 알 수 있다. 왜냐하면, 100 Kbytes를 처리하는 호스트의 부하 상황에 상관없이 계속 100 Kbytes를 처리하는 서버로 동일하게 요청이 분포되고, 100 Kbytes를 처리하는 호스트가 사용자 요청에 응답해야 모든 요청이 끝나기 때문이다.
- LC : 100 Kbytes를 처리하는 서버로 초당 처리된 요청수가 종속되지만, 100 Kbytes로 요청이 몰려 있어 (다른 이미지를 처리하는 호스트에 비해 서버 커넥션 개수가 상대적으로 많음) 상대적으로 다른 서버를 선택할 기회가 많음으로 RR, DH 방법보다 좋은 성능을 보인다.
- DS-HLI : 사용자의 서로 다른 컨텐츠 요청에 따른 호스트 부하 정보를 이용하여 스케줄링을 함으로 100 Kbytes를 처리하는 서버에 초당 처리된 요청수가 종속되지 않고(호스트의 부하 정보에 따라 가중치가 업데이트 됨으로 100 Kbytes를 처리하는 호스트로 요청이 적게 분포됨) 모든 방식 중에서 가장 좋은 성능을 가짐을 알 수 있다.

(3) 방식3-JPEG

표 12와 그림 13은 사용자가 방식3-JPEG으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 방식3-JPEG는 호스트의 성능이 동일한 상황에서 가중치를 가지고 이미지를 랜덤하게 요청하는 방식이다. 이미지에 대한 요청을 처리하는 것은 CPU 자원을 많이 소모하게 됨으로 제안된 방식(CPU 이용률이 가장 적은 호스트로 요청을 분산)이 기존 방식에 비해 약간의 성능 향상을 가짐을 알 수 있다.

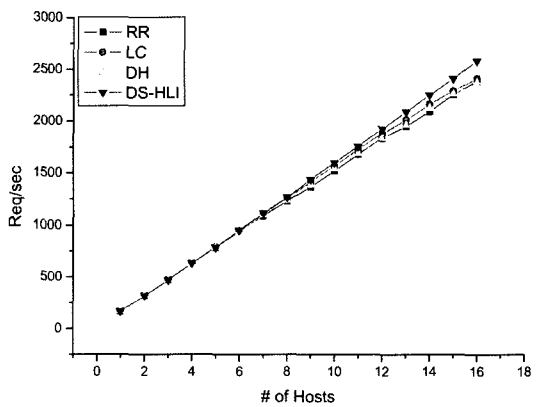


그림 13 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

표 12 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	167	319	473	630	782	941	1088	1230	1369	1520	1682	1838	1950	2097	2255	2398
LC	166	319	474	629	792	950	1110	1258	1417	1570	1730	1880	2013	2163	2294	2416
DH	167	321	470	624	782	933	1099	1233	1387	1543	1689	1836	1964	2126	2265	2378
DS-HLI	165	314	469	632	782	956	1114	1265	1438	1602	1758	1925	2093	2252	2415	2586

(4) 방식4-JPEG

표 13과 그림 14는 사용자가 방식4-JPEG으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 방식4-JPEG는 호스트의 성능이 틀린 상황에서 가중치를 가지고 이미지를 랜덤하게 요청하는 방식이다. RR과 DH는 호스트의 현재 처리 능력에 상관없이 요청을 분산함으로써 가장 낮은 성능을 보임을 알 수 있다. LC는 연결 개수를 이용하여 호스트의 현재 처리 능력을 약간 반영하였고, 제안된 방식은 호스트의 실시간 부하 정보를 이용함으로 기존 알고리즘에 비해 가장 좋은 결과를 보임을 알 수 있다.

(5) 방식5-HTML

표 14와 그림 15는 사용자가 방식5-HTML으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 방식5-HTML은 호스트의 성능이 동일한 상황에서 가중치를 가지고 HTML을 랜덤하게 요청하는 방식이다. HTML은 이미지에 비해 상대적으로 CPU를 적게 사용하고 Network I/O를 많이 사용하기 때문에(초당 요청수가 이미지에 비해 높다) 기존방식과 제안된 알고리즘이 비슷한 성능을 보임을 알 수 있다.

(6) 방식6-JPEG-HTML

표 15와 그림 16은 사용자가 방식6-JPEG-HTML으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 방식6-JPEG-HTML은 호스트의 성능이 동일한 상황에서 가중치를 가지고 이

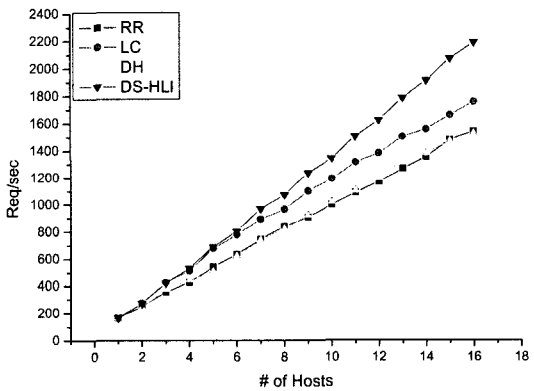


그림 14 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

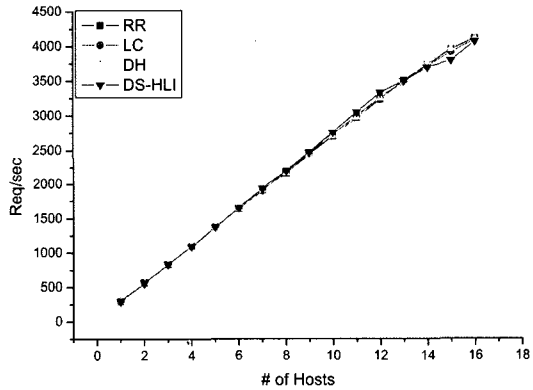


그림 15 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

표 13 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	164	249	354	426	541	634	746	838	906	999	1087	1166	1266	1352	1479	1541
LC	162	272	424	513	677	780	890	964	1100	1189	1311	1380	1500	1555	1659	1756
DH	165	240	373	441	505	620	738	821	920	1017	1101	1191	1328	1379	1469	1524
DS-HLI	166	269	422	530	691	803	967	1073	1235	1342	1502	1622	1784	1911	2072	2190

표 14 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	298	564	834	1102	1373	1640	1913	2157	2444	2696	2967	3232	3502	3727	3948	4107
LC	295	564	834	1100	1368	1640	1901	2186	2457	2729	2991	3259	3488	3710	3912	4082
DH	294	563	838	1103	1367	1639	1909	2160	2428	2696	2963	3246	3461	3757	3954	4110
DS-HLI	294	560	837	1094	1374	1653	1935	2201	2470	2758	3050	3328	3498	3689	3792	4065

표 15 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	203	398	594	789	984	1175	1371	1558	1746	1931	2093	2286	2472	2644	2828	3000
LC	204	397	599	798	990	1189	1389	1588	1778	1949	2138	2323	2486	2665	2834	3011
DH	203	397	593	787	983	1190	1380	1559	1744	1936	2106	2307	2452	2664	2838	2970
DS-HLI	202	395	598	794	995	1195	1403	1592	1804	2001	2276	2457	2594	2794	3037	3236

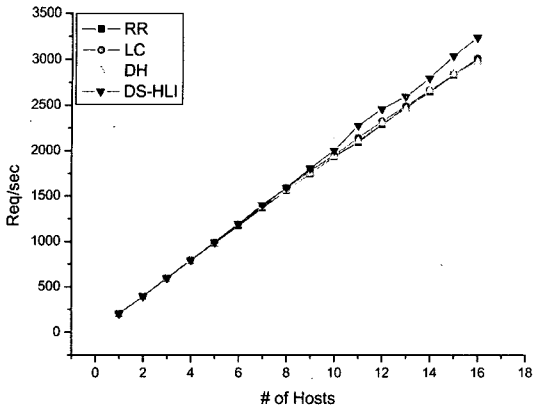


그림 16 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

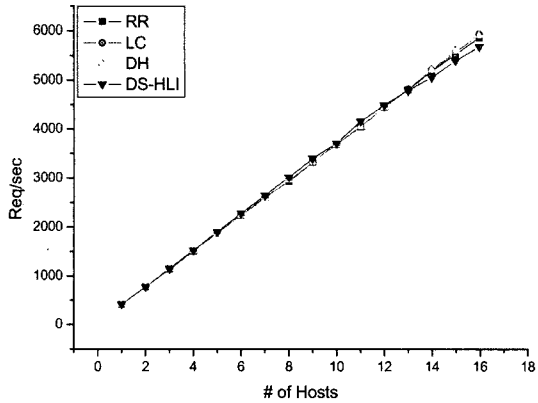


그림 17 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

미지와 HTML을 동시에 랜덤하게 요청하는 방식이다. CPU와 Network I/O를 균일하게 사용하는 경우로써 CPU 이용률에 따라 적절하게 요청을 분산하는 제안된 알고리즘이 가장 좋은 성능을 보임을 알 수 있다.

(7) 방식7-HTML

표 16과 그림 17은 사용자가 방식7-HTML으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 방식7-HTML은 호스트의 성능이 동일한 상황에서 1 바이트의 HTML만을 요청하는 방식이다. 이러한 요청 방식은 CPU 보다는 Network I/O를 더 많이 사용하는 요청을 나타내기 때문에 제안된 방식과 기존 방식이 비슷한 성능을 보임을 알 수 있다.

(8) 방식8-HTML

표 17과 그림 18은 사용자가 방식8-HTML으로 요청한 경우 각 스케줄링별 호스트 개수에 따른 초당 처리된 요청수를 보여준다. 방식8-HTML은 호스트의 성능이 틀린 상황에서 1 바이트의 HTML만을 요청하는 방식이다. CPU 보다 Network I/O를 더 많이 사용하는 요청에서도 호스트의 성능이 틀린 경우 호스트의 실시간 부하 정보를 이용하는 제안된 방식이 기존 알고리즘에 비해 가장 좋은 결과를 보임을 알 수 있다.

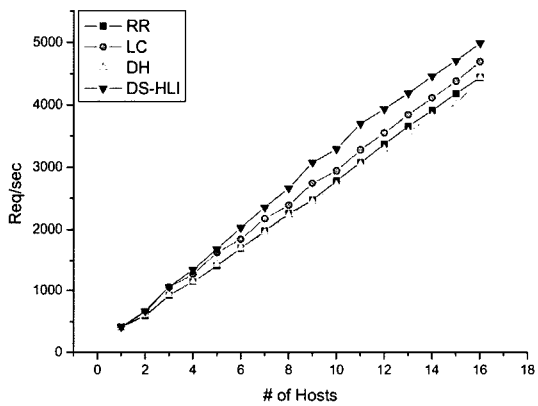


그림 18 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

(9) 기존 스케줄링 vs. 제안된 스케줄링

표 18은 기존 스케줄링 알고리즘에 대한 제안된 알고

표 16 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	408	772	1140	1504	1877	2234	2607	2926	3324	3691	4050	4447	4809	5179	5523	5865
LC	410	773	1145	1504	1873	2235	2607	2929	3325	3682	4056	4448	4820	5200	5559	5930
DH	411	774	1145	1505	1865	2232	2577	2960	3322	3683	4044	4447	4718	5213	5599	5915
DS-HLI	414	769	1148	1528	1898	2276	2642	3014	3395	3709	4161	4487	4797	5053	5390	5680

표 17 스케줄링별 호스트 개수에 따른 초당 처리된 요청 수

Host	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
RR	415	596	922	1151	1416	1698	1973	2255	2472	2782	3077	3368	3655	3915	4187	4444
LC	412	654	1054	1278	1632	1843	2171	2389	2742	2942	3281	3550	3843	4116	4382	4695
DH	414	614	937	1150	1438	1703	1947	2231	2450	2716	3049	3289	3565	3822	4012	4441
DS-HLI	415	672	1065	1356	1687	2027	2355	2662	3076	3293	3697	3935	4194	4463	4713	4994

표 18 제안된 스케줄링 알고리즘의 성능 향상률

요청 방식	성능 향상률 (%)
방식1-JPEG	-1.58
방식2-JPEG	1840.96
방식3-JPEG	2.61
방식4-JPEG	23.46
방식5-HTML	0.33
방식6-JPEG-HTML	3.14
방식7-HTML	0.46
방식8-HTML	13.50

리즘의 성능 향상률을 보여준다. 성능 향상률은 3개의 기존 알고리즘과 제안된 알고리즘과의 성능 향상률을 각각 구한 후에 이를 평균한 것이다. 제안된 알고리즘은 호스트의 성능이 동일하고 랜덤하게 컨텐츠를 요청하는 경우에는 기존 알고리즘과 비슷한 성능을 가진다. 그러나 요청 컨텐츠가 CPU를 많이 사용하는 경우 또는 호스트의 성능이 틀린 경우에는 제안된 방식이 기존 방식에 비해 성능이 높은 것을 볼 수 있다.

본 논문에서 제안하는 스케줄링 방법은 정적 컨텐츠(예, HTML 페이지 또는 이미지)와 동적 컨텐츠(예, 사용자의 요청에 따라 CGI 프로그램을 수행하여 페이지를 생성하여 결과를 전송)가 섞여 있는 응용에도 효과적이다. 반면에 동시 커넥션 개수 기반의 기존 스케줄링 방법들은 정적 컨텐츠와 동적 컨텐츠가 섞여 있는 응용에는 효과적이지 않다. 왜냐하면 동적 컨텐츠가 주로 할당된 서버에서는 정적 컨텐츠가 주로 할당된 서버와 비교할 때 서버의 동시 커넥션 개수가 낮더라도 서버의 CPU 부하가 높을 수 있으므로 동시 커넥션 개수만을 고려 하여 스케줄링을 할 경우 서버들간의 부하분산이 제대로 수행되지 않을 수 있기 때문이다.

4.4 토론

제안된 방식의 장점은 사용자가 서로 다른 컨텐츠를 요청할 때 기존 방식에 비해 성능이 향상된다는 점이다. 이는 제안된 방식이 스케줄링을 할 때 호스트의 부하 정보를 이용한 것으로 설명할 수 있다. 기존 방식은 서버의 처리 능력 혹은 사용자의 요청 컨텐츠에 무관하게 동작하나 제안된 방식은 서버의 CPU 부하 정보를 이용하여 스케줄링 함으로써 서버의 처리 능력 혹은 사용자의 요청 컨텐츠가 변하더라도 이를 능동적으로(사전 실험 없이) 반영하도록 하였다.

제안된 방식은 서버의 부하 정보를 부하 분산기로 보내는 모듈을 각 서버에 설치해야 하고 부하 분산기에서는 각 서버의 부하 정보를 받아 각 서버의 가중치를 업데이트하는 모듈을 설치해야하는 단점을 가진다. 기존 방식은 서버의 부하 정보를 고려하지 않음으로 부하 분산기에서의 설정만으로 부하 분산(스케줄링)이 이루어지나 제안된 방법은 서버들의 부하 정보를 얻기 위해 해당 모듈을 각 서버에 설치하고 부하 분산기에서는 이들의 정보를 받아 각 서버의 가중치를 업데이트하는 모듈을 설치해야 한다.

무선 인터넷 프록시 서버의 주요 기능은 압축과 캐싱이다. 압축은 유선 웹 서버로부터 받은 이미지 등을 무선 클라이언트 기기에 맞게 변환하는 것으로써 많은 CPU 작업을 필요로 한다. 캐싱의 경우 유선 웹 서버의 데이타를 무선 인터넷 프록시 서버에서 저장하여 이를 클라이언트가 사용하게 하는 것으로써 저장된 파일이 작을 때는 많은 Interrupt(네트웍 I/O) 작업이 필요하지만, 저장된 파일이 클 경우에는 많은 CPU 작업을 필요로 한다. 이러한 상황을 고려했을 때, 호스트 부하 정보에서 CPU를 고려하여 부하 분산에 적용하는 것은 의미를 갖는다.

호스트의 부하를 나타내는 최적의 방법은 사용 가능한 모든 부하의 합을 가지고 부하를 나타내는 것이다. 즉, 가능한 동적 부하들의 합은 다음과 같이 표현될 수 있다.

동적 부하 = (A x 커넥션 개수) + (B x CPU 이용률) + (C x Memory 사용률) + (D x Interrupt 개수) + ... + (Z x 부하를 나타내는 정보)

여기서, 모든 가중치의 합은 A + B + C + D + ... + Z = 1이다.

만약, 현재 수행하고 있는 작업이 CPU를 많이 사용하는 작업이라면 B의 가중치를 높게 설정하고, Memory를 많이 사용하는 작업이라면 C의 가중치를 높게 설정하면 된다. 가중치는 항상 고정되어 있는 것이 아니라, 처리하는 작업의 상황에 따라 가변적으로 변해야 한다.

무선 인터넷 프록시 서버의 경우 처리하는 작업의 특성상 CPU를 많이 사용함으로써 CPU의 가중치를 1로 두고 실험을 수행하였다. 만약, 다양한 부하를 사용하는 작업이 발생한다면 이들의 가중치를 반영한 최적의 동적 부하로의 확장도 가능하다.

5. 결 론

본 논문에서는 대용량 트래픽을 처리할 수 있는 무선 인터넷 프록시 서버 클러스터에서 사용되는 스케줄링 방법들의 문제점을 지적하고, 이를 개선한 스케줄링 방법을 제안하였다. 기존 방법이 서버의 처리 능력 혹은 사용자의 요청 컨텐츠에 무관한 방법인 반면 제안된 방법은 서버의 부하 정보를 이용하여 서버의 처리 능력 및 사용자의 요청 컨텐츠 정보를 반영하도록 하였다. 또한 실험을 통해 제안된 방법이 기존 스케줄링 방법에 비해 높은 성능 향상을 가짐을 확인하였다.

향후 연구 방향을 요약하면 다음과 같다:

- 다양한 호스트 부하 정보의 적용 : CPU 부하 정보 외에도 호스트의 부하를 반영하는 메모리, 네트웍 I/O, 시스템 인터럽트 등을 스케줄링 시에 고려한다.
- 사용자의 다양한 컨텐츠 요청 반영 : 정적 컨텐츠(이미지, HTML 등)뿐만 아니라 동적 컨텐츠(스크립트, CGI 등)를 요청하는 경우를 고려한다.

참 고 문 헌

[1] T. Schroeder, S. Goddard and B. Ramamurthy, "Scalable Web Server Clustering Technologies," IEEE Network, Vol.14, No.3, pp.38-45, 2000.

[2] LVS(Linux Vitual Server), http://www.linuxvirtualserver.org.

[3] S. Hwang and N. Jung, "Dynamic Scheduling of Web Server Cluster," Proceedings of the 9th International Conference on Parallel and Distributed Systems, IEEE, 2002.

[4] C. A. Ruggiero and J. Sargeant, "Control of Parallelism in the Manchester Dataflow Machine," In Functional Programming Language and Computer Architecture, LNCS 274, Springer-Verlag, pp. 1-15, 1987.

[5] A. Feldmann, R. Caceres, F. Douglis, G. Glass and M. Rabinovich, "Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments," In Proceedings of the INFOCOM Conference, 1999.

[6] A. Savant, N. Memon and T. Suel, "On the Scalability of an Image Transcoding Proxy Server," In IEEE International Conference on Image Processing, Barcelona, Spain, 2003.

[7] A. Fox, "A Framework for Separating Server Scalability and Availability from Internet Application Functionality," Ph. D. Dissertation, U. C. Berkeley, 1998.

[8] 곽후근, 정규식, "무선 인터넷 프록시 서버 클러스터 성능 개선", 한국정보과학회논문지 : 정보통신, Vol. 32, No. 3, pp. 415-426, 2005. 6.

[9] AB(Apache Bench), http://www.apache.org.

[10] Squid Web Proxy Cache, http://www.squid-cache.org.

[11] T. Lane, P. Gladstone and et. al., "The Independent JPEG Group's JPEG Software Release 6b.," ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gz.

[12] T. Kelly and J. Mogul, "Aliasing on the World Wide Web: Prevalence and Performance Implications," Proceedings of the 11th International World Wide Web Conference, pp. 281-292, 2002.

[13] S. Chandra, A. Gehani, C. Ellis and A. Vahdat, "Transcoding Characteristics of Web Images," Proceedings of the SPIE Multimedia Computing and Networking Conference, 2001.

곽 후 근

1996년 호서대학교 전자공학과 졸업(학사). 1998년 숭실대학교 전자공학과 대학원 졸업(석사). 2006년 숭실대학교 전자공학과 대학원 졸업(박사). 1998년 8월~2000년 7월 (주) 3R 주임 연구원. 2003년 6월~현재 (주) 펌킨넷 코리아 선임 연구원. 관심분야는 Internet Computing and Security

정 규 식

1979년 서울대학교 전자공학과(공학사) 1981년 한국과학기술원 전산학과(이학석사). 1981년~1984년 금성사(현 LG전자) 중앙연구소 선임연구원. 1986년 미국 University of Southern California(컴퓨터공학석사). 1990년 미국 University of Southern California(컴퓨터공학박사). 1990년~1993년 (주) 코리아써키드 기술자문. 1993년 12월~1994년 3월 미국 IBM Wastson 연구소 방문 연구원. 1998년 2월~1999년 2월 미국 IBM Almaden 연구소 방문 연구원. 1990년 9월~현재 숭실대학교 정보통신전자공학부, 교수. 관심분야는 Internet Computing and Security