論文
3-2

실시간 압축 전송 아키텍쳐를 이용한 고성능 웹서버 구현
Development of a High Performance Web Server Using A Real-Time Compression Architecture

민병조(Min-Byung Jo)[1] 강명석(Myung-Seok Kang)[2] 우천희(Chun-Hee Woo)[3]
남의석(Eui-Seok Nahm)[4] 김 학 배(Hag-Bae Kim) [5]

국문요약

현재 인터넷은 HTTP 프로토콜을 중심으로 구성되어 있다. 전자상거래, 전자정부, 인터넷 멀티미디어, 홈 네트워킹 등 현재 인터넷으로 가능한 서비스는 점점 증가하고 있지만, 비교적 단순한 메시지의 전송에 기반해 설계된 HTTP가 현재의 이러한 복잡한 서비스와는 적합하지 않는 경우도 있다. 즉, 점점 더 다양해지는 서비스로 인해, 서버 쪽에서는 클라이언트당 점점 더 많은 양의 데이터를 전송하게 되는 것이다. 이러한 웹 서버측의 메시지 전송 부담과 전송 시간, 대역폭의 절감을 위해서, HTTP 프로토콜의 압축전송헤더를 이용한 실시간 압축 전송 아키텍쳐를 개발하였다. 이를 이용하여, 정적 페이지를 실시간으로 압축/전송함으로써, 웹서버측의 대역폭을 절감하였고, 전송속도를 증가시켰다. 또한, 상대적으로 CPU 집중적인 압축연산을 커널 영역에서 수행하여, 압축연산으로 인한 서버측의 CPU 자원사용을 최소화하였다.

영문요약

In these days, such services are popularized as E-commerce, E-government, multimedia services, and home networking applications. Most web traffics generated contemporarily basically use the Hyper Text Transfer Protocol(HTTP). Unfortunately, the HTTP is improper for these applications that comprise significant components of the web traffics.

In this paper, we introduce a real-time contents compression architecture that maximizes the web service performance as well as reduces the response time. This architecture is built into the linux kernel-based web accelerating module. It guarantees not only the freshness of compressed contents but also the minimum time delay using an server-state adaptive algorithm, which can determine whether the server sends the compressed message considering the consumption of server resources when heavy requests reach the web server. Also, We minimize the CPU overhead of the web server by exclusively implementing the compression kernel-thread. The testing results validates that this architecture saves the bandwidth of the web server and that elapsed time improvement is dramatic.

한글키워드(영문키워드): HTTP 압축(HTTP compression), 고성능 웹서버(high performance web server), 웹 가속(web acceleration)

1) 정회원 : 연세대학교 대학원 박사과정
2) 정회원 : 연세대학교 대학원 박사과정
3) 정회원 : 명지전문대학 전기과 부교수
4) 정회원 : 극동대학교 정보통신학부 전임강사
5) 정회원 : 연세대학교 전기공학과 부교수

논문접수 : 2004. 3. 24.
심사완료 : 2004. 4. 2.

1. 서론

HTTP 프로토콜의 경우, "Content-Encoding"과 "Transfer-Encoding" 헤더 옵션을 통해, 압축 전송을 지원한다. 두 헤더는 IETF Internet RFC에 의해 정의되었으며[1], 전자는 전송할 컨텐츠가 클라이언트의 요청이 이루어지기 전에 압축시키는 경우에 사용되며, 후자는 클라이언트의 요청이 도착한 후에, 전송시점에 압축시키는 경우에 사용되도록 정의되었다. 하지만 전자의 경우처럼, 클라이언트의 요청이 이루어지기 전에 단순히 미리 컨텐츠를 압축하는 것은 웹서버 측의 저장공간에 대한 부담이나 컨텐츠의 신선성(freshness) 관리 등과 같은 문제로 인해 효용성이 떨어진다. 이러한 점으로 인해, 현대의 웹사이트의 경우, 두 가지 헤더옵션은 컨텐츠의 전송이 압축전송방식인지의 여부를 판단하는 거의 같은 의미로 사용되게 된다.

근래의 대부분 웹사이트의 경우, 사용자의 요청에 의해 만들어지는 동적 페이지가 점점 더 증가하는 추세이다. 하지만, 대부분의 사용자가 접속하는 메인 페이지나 주요한 페이지의 경우에는 웹서버의 전송속도를 고려해, 정적인 페이지 위주로 설계된다. 가장 방문율이 높은 대한민국의 포털사이트 다섯 개의 메인 페이지 구성을 분석하여, 그를 토대로 웹서비스에서의 압축전송이 어느 정도 효용성을 가질 수 있는지 평가하였다.

인터넷조사 전문업체인 랭크서브[2]에 따르면, 2003년 3월 3주째의 대한민국의 인터넷 포털사이트의 방문율(page view) 순위는 다음,야후코리아,한미르,네이버,엠파스 순으로 나타났다. 이러한 대부분 포털 사이트 및 유명 사이트의 경우, 사용자가 처음으로 접속하는 메인 페이지의 구성이 매우 중요하다. 이러한 메인 페이지의 구성은 사이트의 통일성이나 친숙함을 위해, 세부적인 메뉴나 링크들에 비해 구조적으로 많이 변화하지 않는다. 또한 광고나 정보배너들과 같이 세부적인 컨텐츠가 매일매일 변화하는 페이지를 분석한다는 것은 효용가치가 그리 높지 않다. 그러므로, 본 논문에서 각 상용사이트의 페이지를 분석하는 데 있어, 해당 각 사이트의 메인 페이지의 html을 분석하여, 메인 페이지의 구성 컨텐츠를 압축전송 가능한 컨텐츠와 그렇지 못한 컨텐츠로 분류하고, 그 구성도를 조사하였다. 분류 방법은 컨텐츠 구분에서 가장 논리적이라 할 수 있는, 파일 확장자 명으로 분석하였다.

해당 각 사이트의 컨텐츠 구성 분석 결과는 다음 표와 같다.

<표 1> 컨텐츠 구성 분석결과

(%)	다음	야후 코리아	한미르	네이버	엠파스
html	21.56	33.33	17.75	15.00	44.74
htm	0.00	3.45	0.72	0.00	1.75
txt	3.67	0.00	31.52	1.67	0.00
js	10.09	12.64	9.78	10.00	16.67
gif	34.86	33.33	12.32	16.67	16.67
jpg	0.92	0.00	0.00	0.00	0.00
jsp	9.63	11.49	9.42	10.00	10.53
asp	3.67	5.75	5.80	5.83	8.77
cgi	15.14	0.00	6.88	1.67	0.00
tif	0.00	0.00	0.00	0.00	0.88
php	0.46	0.00	5.80	39.17	0.00
압축가능 컨텐츠	35.32	49.43	59.78	26.67	63.16
총 컨텐츠 수	218	87	276	120	114

앞서 언급된 주용 다섯개 사이트의 메인 페이지 분석 결과는 다음과 같다.

<표 2> 메인 페이지 분석 결과

	점유율 상위 5개 사이트 메인 페이지
압축가능 컨텐츠 수	389
전체 컨텐츠 수	815
압축가능 컨텐츠/전체 컨텐츠 비(ratio)	47.73%

<표 1>의 메인 페이지 분석을 통해, 메인 페이지 전체 구성중, 압축과정을 통해 이득을 볼 수 있는 컨텐츠 구성비가 26.67~63.16%를 점유하는 것을 확인하였으며, <표 2>의 통합 데이트 분석의 경우, 약 47.73%의 압축전송가능 컨텐츠가 메인 페이지를 구성하고 있다. 위의 주요 포털 사이트의 인터넷 메인 페이지 분석 결과를 토대로, 웹서버의 정적 컨텐츠 압축전송이 단지 HTTP 스펙상의 기능이 아니라, 실제 상용 사이트에도 유용하다는 것을 보여주고 있다. 즉, 서버 측의 전송 자원(resource)을 절약하며, 전송 컨텐츠양을 줄임으로써, 서버 측의 요청 응답시간을 줄이며, 전체적인 성능향상을 얻는 것이다.

본 논문의 구성은 다음과 같다. 2장에서는 HTTP의 압축지원 기능과 각 구성요소를 나타낸다. 3장에서는 압축전송기능의 구현을 4장에서는 실시간 압축전송 아키텍처와 실제 시스템 구현을 통한 성능평가에 대하여 설명한다. 마지막으로, 5장에서는 결론을 맺고 향후 연구될 사항을 제시한다.

2. HTTP 압축 기능

HTTP의 압축기능은 클라이언트들의 브라우저에서 서버 측에 요청을 보낼 때, "Accept-Encoding" 헤더와 함께 지원되는 압축 알고리즘에 대한 정보를 보냄으로써 시작된다. 인터넷 익스플로러, 넷스케이프 및 오로라 등 주요 브라우저들은 거의 모두 "gzip" 압축 알고리즘[3]을 지원한다. 이 이외에도 "deflate","compress"와 같은 압축기법들도 있지만, 대부분의 브라우저는 "gzip"을 지원한다. 이렇게 클라이언트 측에서 보내진 HTTP 헤더를 통해, "gzip" 과 같은 특정 압축알고리즘을 지원한다는 메시지가 전달되면, 서버 측에서는 전송할 컨텐츠가 압축이 효율적인 컨텐츠인지 여부를 판단하여, 클라이언트에게 전송하게 된다. [4]에서는 HTTP 1.1 프로토콜의 압축전송 기능이 웹서버 성능에 어떤 영향을 미치는지에 대한 연구가 이루어졌다. 아파치[5], MS의 IIS[6]와 같은 대부분의 웹서버 프로그램들도 최근에 들어, 이러한 압축전송기법을 지원하기 시작하였다. 전체 웹서버 중 가장 많이 사용되는 아파치 웹서버 프로그램의 경우, 압축전송 기법을 사용하지만, 실시간 전송기법이 아니라, 미리 압축되어진 컨텐츠의 전송을 지원하는 형태를 취하고 있다. 이는 컨텐츠의 신선성(freshness)을 보장하지 못하며, 압축된 컨텐츠의 저장 및 관리가 지능적이지 못하다. IIS의 경우 5.0 이후에 ISAPI filter에 "gzip" 압축 알고리즘을 지원하여, 사용자가 특정 페이지를 요청하면, 서버는 이를 전송한 뒤에, 이를 임시 폴더에 압축하여 복사한다. 그리고 다음 사용자가 같은 페이지를 요청할 경우, 임시 폴더에 저장된 압축 페이지가 전송된다. 이러한 두 가지 대표적인 웹서버 프로그램의 압축 전송 기능은 다음과 같은 문제점을 가진다.

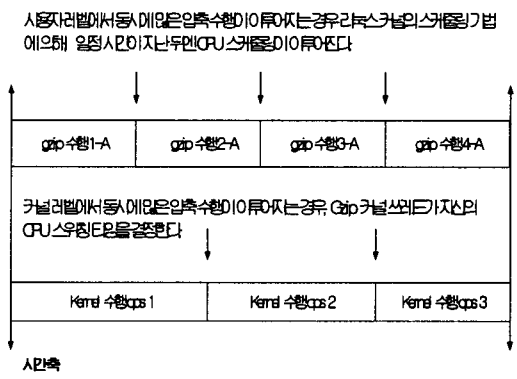
첫째, 압축된 컨텐츠 저장 및 관리가 효율적이지 못하다.
둘째, 아파치의 경우, 압축 컨텐츠의 신선성(freshness) 관리가 효율적이지 못하다.
셋째, 컨텐츠 압축전송이 실시간성을 만족하며 수행되지 않는다.
넷째, 압축 수행의 경우, 비교적 CPU 집중적인 연산인데, 순간적으로 많은 요청이 서버에 도달될 경우, 이로 인한 CPU 오버헤드를 무시하지 못한다.
다섯째, 압축되는 컨텐츠가 주로 디스크에 저장되어서, 메모리 저장방식에 비해 IO 속도 지연이 존재한다.

3. 커널영역에서의 "gzip" 압축

알고리즘 구현

"gzip" 압축알고리즘의 경우, 원본 데이터의 손실이 없는 압축 알고리즘으로, 입력 데이터의 중복된 문자열을 찾아내어, 두 번째 문자열을 첫 번째 문자열에 대한 포인터로 대체함으로써, 전체 데이터를 압축하는 방식으로 구현된다. 이 방식의 경우, 데이터 압축률이 높으며, 압축수행속도 측면에서도 현재까지 가장 빠른 것으로 평가되고 있다. 하지만, 웹서버와 같이 순간적으로 많은 압축 전송 요청이 서버에서 진행될 경우, CPU 집중적인 압축연산으로 인한, 전체 서버의 부하증가로 웹 서비스가 지연되게 된다. 이에 대한 주요 지연이유는 다음과 같다.

첫째, 압축프로그램 연산은 파일 IO에 관련된 많은 시스템 콜을 사용한다.
둘째, 순간적으로 많은 요청이 서버에서 수행될 경우, 각각의 압축 연산에 대한 리눅스 커널 내부의 CPU 스위칭이 많이 일어나게 되며, 이로 인한 오버헤드가 존재한다.
이러한 순간 요청으로 인한 웹서버의 성능문제는 주로 별도의 웹서버를 추가하거나, 웹 캐싱 기법을 이용하여 해결되어 왔다[7,8]. 반면, 추가적인 하드웨어의 도입 없이 웹서비스를 리눅스 커널 영역에서 처리하게 하여, 단일서버당 웹처리속도를 극대화하는 방법도 연구되어왔다[9].
본 논문의 경우 [9]와 같이 이러한 압축연산의 처리속도를 개선하기 위해서, 리눅스 커널 내에서 압축알고리즘을 구현하여, 압축연산시 시스템 콜을 사용하지 않고, IO 연산을 커널 내부에서 직접 수행하여, 이에 관련된 시스템 오버헤드를 없앴고, 압축연산 자체가 커널 내부의 단독 쓰레드(thread)로 동작하여, CPU를 배타적으로 독점할 수 있는 권한을 가지게 되어, CPU 스위칭 횟수가 현저히 줄게 된다. 다음 그림은 이러한 상황에서의 CPU 스위칭 상황을 표현한 것이다.
또한, 수행할 gzip 쓰레드의 수를 사용자가 설정(configure)가능하도록 파라미터(parameter)화하였으며, gzip 쓰레드의 과도한 CPU 독점 문제를 해결하기 위해, 실제 구현 코드(code)에서 전체 웹서비스 속도에 매우 critical한 부분을 제외하고 적절한 실행 코드마다 리눅스 커널의 schedule() API를 사용하여, CPU의 리스케줄링을 일어나게 하였다. 이 순간, 만약 우선순위가 더 높은 다른 커널 쓰레드가 존재한다면 CPU 사용권을 넘겨주게 될 것이며, 그러한 커널 쓰레드가 없다면 다시 gzip 쓰레드로 CPU 사용권이 넘어오게 될 것이다. 결과적으로는 gzip 쓰레드의 독점문제로 딴 커널쓰레드의 사용이 크게 저하되는 문제는 발생되지 않는다.



(그림 1) CPU 스위칭 방식의 차이

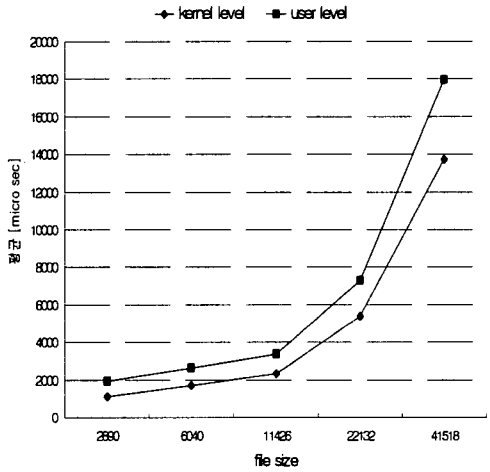
결과적으로, 사용자레벨에서의 너무 많은 프로세스의 동작으로 인한 CPU 스위칭 오버헤드가 없다. 이로써, 사용자 레벨에서의 압축보다 처리속도에 이득을 얻을 수 있다. 이를 증명하기 위해, 본 논문에서는 "gzip" 알고리즘을 사용자 영역과 커널 영역에서 각각 구현하여, 그 수행시간을 테스팅하였다.

압축 수행(operation)을 리눅스 커널에서 구현한 경우와 일반적인 사용자 영역에서 구현한 경우에 대한 실험을 시행하였다. 그리하여, 압축 수행을 사용자 영역에서 시행하였을 경우와 커널레벨에서 실행하였을 경우의 수행 시간을 측정하여 상호 비교함으로써 커널영역에서 압축한 경우의 시간이득을 측정하였다.
사용된 압축 알고리즘은 사용자 영역 및 커널 영역 모두 위에서 언급한 LZ77[10]의 알고리즘

을 사용하였고, 이는 가장 많이 알려진 "gzip"의 기본 알고리즘이다. 이 알고리즘은 입력 데이터에서 중복되는 문자열을 찾아내서, 두 번째 같은 문자열을 첫 번째 문자열과의 거리와 길이로 표현된 포인터로 대체한다. 중복되는 두 문자열의 거리(distance)는 32K 바이트(bytes)로 한정되며, 길이(length)는 258 바이트(bytes)로 제한된다. 이렇게 처리된 글자, 일치 길이(length)는 하나의 호프만 트리(Huffman tree)로 압축되며, 일치 거리(distance) 또한 다른 트리로 압축된다. 이러한 트리들은 각 블락의 시작 지점에 저장된다. 이러한 방식을 통해, 중복된 문자열은 해시 테이블을 이용하여, 찾아진다. 이 방식의 경우, 데이터 압축률이 높으며, 내부 연산량이 작기 때문에 압축수행속도 측면에서는 현재까지 가장 빠른 것으로 평가된다.

객관적인 속도 측정을 위해, 같은 컨텐츠를 가지고 있는 동일한 파일에 대해, 커널 영역과 사용자 영역 모두 각각 약 1000번의 압축 동작을 수행하였으며, 그 평균 수행시간으로 압축수행속도를 비교하였다.

각 압축수행의 시간 측정은 두 가지 모두 gettimeofday 시스템 콜(system call)을 이용하여, 압축수행 직전의 시간과 압축수행 직후의 시간차를 계산하였고, 다른 시스템 상의 IO 오퍼레이션이 실험에 영향을 주지 못하게 하기 위해, 네트워크나 주요 로그 데몬 등이 동작하지 못하게 한 상태에서 실험하였다. 테스팅 파일의 경우, 2890, 6040, 11426, 22132, 41518 바이트(bytes)이며, 모두 ZDNet의 WebBench[11]에서 사용되는 테스팅 부하(load) 파일을 사용하였다. 다음은 커널 영역 및 사용자영역에서의 압축 수행시간을 비교한 테스팅 결과이다.



(그림 2) 사용자/커널 영역에서의 압축수행 시간 비교

구체적인 압축수행시간 및 비교는 다음 표와 같다.

<표 3> 압축수행 비교결과

테스트 파일 사이즈 (byte)	커널에서의 압축수행시간 (micro sec)	사용자영역에서의 압축수행시간 (micro sec)	상대 압축성능이득 (%)
2890	1109.2	1952.7	43.2
6040	1694.2	2674.0	36.6
11426	2320.3	3367.3	31.1
22132	5423.5	7298.9	25.7
41518	13738.1	17953.7	23.5

위의 결과와 같이, 압축 연산을 사용자 영역과 커널영역에서 비교한 결과, 파일 사이즈에 따라 다르지만, 커널영역에서 압축연산을 수행한 결과가 약 23~43% 정도 빠른 것을 확인할 수 있다. 그러므로, 앞에서 언급한 것과 같은 이유로 커널에서 수행하는 단일 압축 수행이 사용자 영역에 비해 더 빠른 것을 알 수 있다.

4. 실시간 압축전송 아키텍처

기존 웹서버 프로그램의 압축전송 기능은 압

축전송 기능이 실시간적이 아니며, 압축대상 컨텐츠의 관리가 지능적이지 못해, 신선성(freshness)을 보장할 수 없으며, 또한 순간 트래픽이 아주 클 경우, 압축 수행으로 인한 부하로, 기존 웹서비스의 처리속도가 지연될 가능성을 포함한다는 것이다. 또한 주로 압축 컨텐츠를 디스크에 저장함으로써, 컨텐츠에 대한 IO가 디스크 IO속도에 의해 결정된다.

4.1. 지능적 압축전송 알고리즘

본 논문에서는 이러한 단점들을 고려하여, 압축전송 아키텍처를 설계하였다.

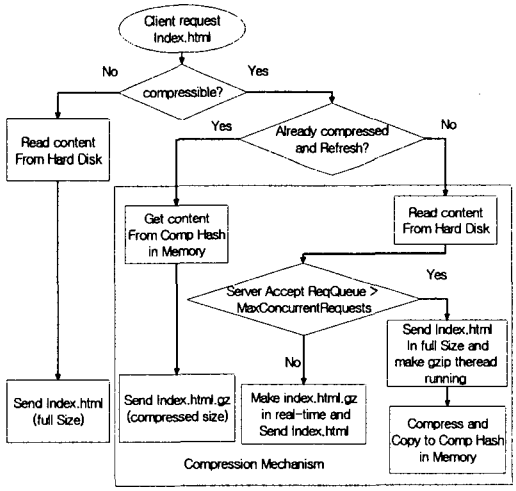
첫째, 무조건적인 실시간 압축 전송이 아니라, 서버의 자원상황을 고려해, 적응(adaptive)적인 실시간 전송여부를 판단하게 구현되었다. 즉, 기존 웹서버와 같이 웹 서비스 루틴 도중에 압축을 하는 경우, 컨텐츠 사이즈 및 서버측 자원상황에 따라 오히려 서비스 처리속도가 늦어지는 시간지연이 존재하는 경우가 발생하게 된다. 그리하여, 본 아키텍처에서는 기존 컨텐츠 서비스 루틴에서 압축을 하는 것이 아니라, 서버의 idle 시간에 동작하는 압축 쓰레드를 두어, 이 쓰레드에서 압축을 해야하는 컨텐츠를 처리하게 하였다. 이는 사용자 요청에 따른 처음으로 압축가능 컨텐츠를 전송할 경우, 서버측의 상황을 고려하여, 서버 측의 Accept Queue에 대기중인 순간 요청수(request) 양이 사용자가 설정한 허용가능최대요청수를 넘을 경우, 실시간 압축전송기능을 사용하지 않고, 압축 전용 쓰레드에 압축하라는 명령만 내려두고, 이를 서버의 idle 시간에 처리하였다. 결과적으로 서버의 트래픽으로 인해 부하가 많은 상황에서도 첫 번째 컨텐츠의 서비스 전송 시간지연을 최소화할 수 있다.

둘째, 컨텐츠를 압축할 때, 디스크가 아니라, 메모리 상에 해쉬형태로 저장하여, 디스크에 보다 IO 속도를 높이고, 컨텐츠 탐색을 빠르게 처리하도록 하였다. 또한, 압축되어 저장될 컨텐츠의 최대허용가능 메모리 사용량을 파라미터화하여, 런타임중에 기설정된 최대허용가능 메모리 사용량를 넘어갈 경우, 순간적으로 압축수행을 하지 않는다. 또한 실제 압축수행 컨텐츠 서비스 시간간격을 사용자가 파라미터(parameter)화하여, 이 시간간격내에 압축컨텐츠의 재사용이 일어나지 않을 경우, 압축되었던 메모리를 해제하는 방식으로 메모리 사용량이 무한정 증가되는 문제점을 피하였다.

셋째, 압축된 컨텐츠의 신선성을 보장하기 위해, 압축된 컨텐츠를 메모리에 저장할 때, 이 컨텐츠의 last access time 정보를 같이 저장하고, 서비스하기 전에 이 정보를 실제 파일의 last access time 값과 비교하여, 컨텐츠의 변경을 지능적으로 처리하도록 구현하였다.

넷째, 압축컨텐츠의 MIME 타입 및 파일별 허용압축최대크기 제한을 사용자가 유연하게 설정(configure)할 수 있다.

그림 3은 본 실시간 압축 전송 메커니즘 알고리즘을 간략하게 다이어그램으로 표현한 것이다.



(그림 3) 실시간 압축전송 알고리즘도

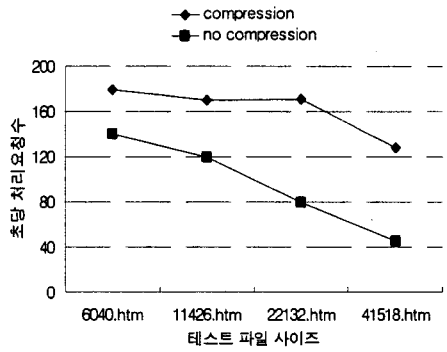
4.2. 실시간 압축전송기능이 구현된 커널 웹서버의 성능 평가

실시간 압축성능의 성능을 평가하기 위해서, 커널 레벨의 압축 전송 아키텍처를 사용하였을 경우와 그렇지 않은 경우, 웹서버의 응답시간, 처리속도를 평가하였다. 네트워크 상의 간섭성을 최소로 하기 위해, 단일 내부(private) 네트워크로 구성하여, 테스팅을 받는 웹서버와, 서버에게 HTTP 트래픽을 요청하는 부하(load) 발생(generate) PC들로 구성하였다. 또한 압축

전송과 그렇지 않은 경우의 객관적인 비교를 위해, 두 경우 모두 Apache Benchmark 툴[12]을 이용하여 웹서버에 부하를 거는 2대의 컴퓨터를 사용해 서버부하를 발생하였으며, 응답시간을 체크하는 PC를 따로 두어, 두 경우의 웹서버의 응답시간 및 초당처리요청수를 체크하였다. 압축전송의 경우는 본 연구 결과물인 커널모듈을 로딩하여 시간을 체크하였으며, 비압축전송의 경우는 같은 서버에서, 레드햇 9.0 환경에서 기본적으로 사용되는 Apache 1.3.27을 사용하여 테스트하였다. 또한 모든 서버 및 부하 PC의 경우, 레드햇 9.0을 사용한 리눅스 플랫폼으로 실험하였다. 압축전송의 속도를 테스팅할 실제 컨텐츠의 경우, 객관성을 확보하기 위해, WebBench 테스팅툴에서 사용하는 테스팅 페이지 중에서 컨텐츠 사이즈별로 구분하여 사용하였다. 또한, 그림 4,5,6의 6040.htm, 11426.htm, 22132.htm, 41518.htm의 경우 파일 이름과 파일 사이즈가 동일하며, 이는 압축하지 않았을 경우, 6040.htm의 파일사이즈가 6040 바이트(bytes) 임을 의미한다. 테스팅은 각 페이지별로, 압축 기능을 사용한 경우와 그렇지 않은 경우로 나뉘어져서, 각각 100,000 번의 테스팅에 대한 평균값으로 계산하였다. 결과적으로 다음 압축전송과 비압축전송시의 테스팅 파일 사이즈별의 실험결과를 도출하였다.

4.2.1 초당 처리 요청수(request per second)

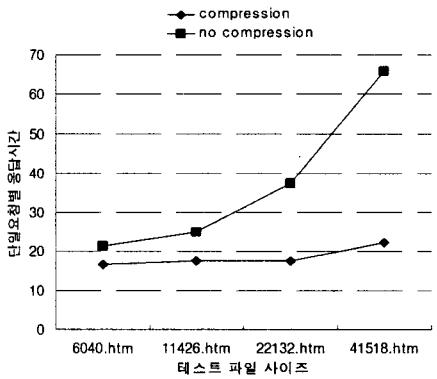
그림 4는 실시간 압축전송 아키텍쳐를 사용한 경우와 그렇지 않은 경우 웹서버의 초당처리 요청수를 비교한 것이다. 파일 사이즈가 6040, 11426, 22132, 41518 bytes 인 경우를 각각 테스팅하였으며, 압축전송 아키텍쳐를 이용한 경우 단위 시간당 더 많은 요청을 처리한다.



(그림 4) 테스트 파일 사이즈별 서버측의 초당 처리 요청수

4.2.2 요청 수행시간(time per request)

그림 5는 실시간 압축전송 아키텍쳐를 사용한 경우와 그렇지 않은 경우 웹서버의 단일 요청당 응답시간을 비교한 것이다. 파일 사이즈가 6040, 11426, 22132, 41518 bytes 인 경우를 각각 테스팅하였으며, 압축전송의 경우 단일 요청당 응답시간이 더 빠르며, 파일 사이즈가 커질수록 응답시간의 차이가 커진다.

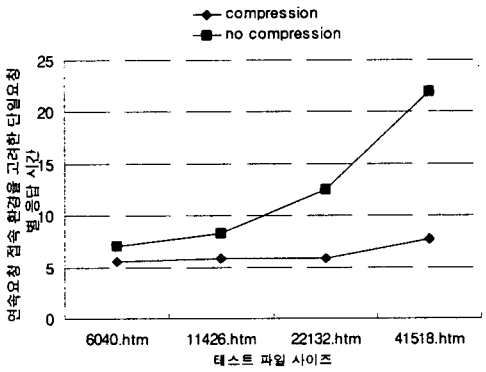


(그림 5) 테스트 파일 사이즈별 서버측의 단일요청별 응답시간

4.2.3 동시요청을 고려한 요청 수행시간(time per request)

그림 6는 실시간 압축전송 아키텍쳐를 사용한 경우와 그렇지 않은 경우 웹서버의 단일 요청당 응답시간을 비교한 것이다. 4.2.3과 다른 점

은 최신 브라우저들의 연속요청(pipeline request) 기능을 반영하여, 웹서버에 요청하는 응답시간 분석 프로그램을 순차적이 아니라, 멀티 쓰레드를 사용하여 concurrent하게 구현한 것이다. 파일 사이즈가 6040, 11426, 22132, 41518 bytes 인 경우를 각각 테스팅하였으며, 압축전송의 경우 단일 요청당 응답시간이 더 빠르며, 파일 사이즈가 커질수록 응답시간의 차이가 커진다.



(그림 6) 테스트 파일 사이즈별 연속요청 접속 환경을 고려한 단일요청별 응답 시간

그림 4,5,6과 같은 압축전송시의 웹서버 성능평가 결과를 요약하면 다음 표와 같다.

<표 4> 압축전송기능이 수행된 경우 웹서버의 성능평가

Apache Benchmark 비압축전송/압축전송의 경우	6040 바이트 비압축 / 압축시	11426 바이트 비압축 / 압축시	22132 바이트 비압축 / 압축시	41518 바이트 비압축 / 압축시
총테스팅 요청수	10^5	10^5	10^5	10^5
실패 요청수	0 0	1 0	1 0	0 1
테스팅 파일 사이즈(byte)	6040 2401	11426 2937	22132 4836	41518 11446
초당 처리 요청수	140.92 179.11	120.03 169.94	79.89 170.96	45.7 128.67
단일요청당 응답시간(ms)	21.34 16.79	25.03 17.71	37.58 17.63	65.9 22.33
클라이언트의 연속요청기능을 이용한 경우의 단일요청당 응답시간(ms)	7.11 5.59	8.34 5.91	12.53 5.88	21.97 7.77

테스팅 파일인 6040, 11426, 22132, 41518.htm의 경우, 압축을 하고난 뒤 각각 2401, 2937, 4836, 11446 bytes의 크기를 가졌으며, 총 10^5의 요청실험 중 비압축실험의 경우, 11426.htm, 22132.htm의 경우 1번의 응답실패가 존재하였고 압축전송실험의 경우 41518.htm의 경우 1번의 응답실패가 존재하였다. 실시간 압축전송 아키텍쳐를 사용하였을 경우의. 비압축전송시의 상대적인 성능비를 요약하면 다음과 같다.

<표 5> 압축전송기능의 사용시 비압축전송시에 대한 상대 성능 이득

테스팅 파일별 상대성능이득	초당처리 요청수	단일요청당 응답시간	단일요청당응답 시간(동시접속 환경시)
6040.htm	21.32%	27.10%	21.38%
11426.htm	29.37%	41.33%	29.14%
22132.htm	53.27%	113.16%	53.07%
41518.htm	64.48%	195.12%	64.63%

위의 표에서와 같이, 초당 처리 요청수의 경우, 압축전송을 사용한 경우가 약 21~64% 정도 더 높은 성능을 보였으며, 전송 파일 사이즈가 커짐에 따라 비례하여 더 큰 성능 차이를 보여주었다. 단일요청당 평균 응답시간의 경우, 클라이언트의 연속요청환경을 고려하였는지 여부에 따라, 다른 성능 차이를 보여주었는데, 약 27~195%의 성능 차이를 보여주었다. 역시 전송 파일사이즈가 커짐에 따라 더 많은 응답시간의 차이를 보였으며, 압축전송시 더 높은 성능 이득을 보여주었다.

6. 결론

점점 더 확산되는 인터넷 사용에 대해서, 웹서버 프로그램 자체의 성능을 높이려고 하는 방법은 주로 하드웨어적인 관점에서 진행되어 왔다. 이러한 방식은 주로 웹서비스를 처리하는 서버 수를 증대하거나 개별적인 CPU, 네트워크 장비의 성능향상으로 진행되어 왔다. 반면 본 논문에서는 이러한 하드웨어적인 성능향상 관점과는 달리, 기존 웹서버 프로그램의 요청 처리방식을 분석하고, 새로운 요청 처리방식을 도입하였다. 즉, HTTP의 압축전송 기능을 웹서버 단의 커널 내에서 구현하였다. 이는 상대적으로 CPU에 집중적 연산을 하는 "gzip" 압축 알고리즘 수행을 커널 내부에서 구현하여, 사용자 영역에서 구현한 것보다 약 23 ~40% 정도 더 빠른 연산을 수행함을 증명하였다. 또한 이러한 압축 알고리즘의 구현 이외에도, 지능적인 압축 전송 기법을 도입하여, 트래픽 양에 따른 적응적(adaptive) 아키텍처를 구현하였다. 마지막으로, 본 실시간 전송 아키텍처를 실제적으로 구현하여, 대표적인 웹서버 프로그램인 아파치와 응답시간 및 초당 처리 요청수를 비교하였으며, 기존 방식보다 약 21~195 % 더 낳은 성능을 보여주었다. 또한 압축기능을 사용하는 컨텐츠의 신선성(freshness)을 보장할 수 있도록 하였고, 백엔드 전용 압축 커널쓰레드를 사용해 압축함으로써, 압축수행으로 인한 전송시간 지연을 최소하도록 하였다. 또한 이러한 정적 컨텐츠 위주의 웹서비스개선 방법과 더불어, 향후 동적 페이지와 정적 컨텐츠가 혼합된 상용 사이트의 객관적인 웹컨텐츠 속도 분석 및 이에 대한 웹서비스 처리속도 개선방법에 대한 연구도 수행되어야 한다.

참 고 문 헌

[1] R.Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. BernersLee, "Hypertext transfer protocol-HTTP/1.1", IETF RFC 2068, January 1997.

[2] http://www.rankserv.com

[3] J. L. Gailly. Gzip program and documentation, 1993. Source code available from ftp://prep.ai.mit.edu/pub/gnu/gzip-* .tar.

[4] H. Nielsen , J. Gettys , A. Baird-Smith , E. Prud'hommeaux , H. Lie , C. Lilley, "Network performance effects of HTTP/1.1, CSS1, and PNG" ACM SIGCOMM Computer Comm. Review, Proc. of the ACM SIGCOMM '97 conf. on Applications, technologies, architectures, and protocols for computer communication, Vol. 27, pp. 155-166, 1997.

[5] http://www.apache.org

[6] http://www.microsoft.com/iis

[7] E. Levy, A. Iyengar, J. Song, and D. Dias, "Design and Performance of a Web Server Accelerator", IEEE INFOCOM'99, March 1999.

[8] J. Song, E. Levy, A Iyengar, and D. Dias, "Design Alternatives for Scalable Web Server Accelerators", IEEE ISPASS, pp. 184-192, 2000.

[9] 박종규,민병조,임한나,박장훈,장휘,김학배, "커널 쓰레드 웹가속기(SCALA-AX) 개발", 정보처리학회논문지 A 제 9-A권, 제 3호, pp. 327-332, 2002.

[10] A. Lempel, J. Ziv, "A Universal Algorithm for Sequential Data Compression," IEEE Trans. Information Theory, pp. 337-343, 1977.

[11] zdnet, Inc. Webbench Benchmark Information. http://www.zdnet.com/etestinglabs/sto-ries/ benchmarks/0,8829,2326243,00.html 1998.

[12] http://httpd.apache.org/docs/programs /ab.html

민병조:Min Byung Jo

1998년 연세대학교 전기공학과 졸업(학사)
2001년 연세대학교 대학원 전기컴퓨터공학과(공학석사)
2001년 - 현재 연세대학교 대학원 전기전자공학과 박사과정
관심분야: 인터넷 웹서버 기술, 임베디드 시스템, 정보보안

강명석:Kang Myung Seok

2001년 원광대학교 컴퓨터공학과 졸업(학사)
2003년 원광대학교 컴퓨터공학과 졸업(공학석사)
2003년 - 현재 연세대학교 대학원 전기전자공학과 박사과정
관심분야: RTOS, Fault-Tolerant, 지능형 홈 네트워크

남 의 석 Nahm Eui Seok

1991년 연세대학교 전기공학과과 졸업.
1993년 연세대학교 석사
1998년 연세대학교 박사
2003년~현재 극동대학교 정보통신학부 전임강사.
관심분야는 시스템 제어 및 응용, 지능형 모델링.

우 천 희 : Chunhee Woo

1985년 연세대학교 전기공학과 졸업.
1993년 연세대학교 석사
2000년 연세대학교 박사
1995년~현재 명지전문대학 전기과 부교수.
관심분야는 전기 응용시스템, 배전계통, 제어시스템 및 응용, 모델링.

김 학 배 : Hagbae Kim

1988년 서울대학교 전자공학과 졸업.
1990년 미국 미시간대학교 전기 및 컴퓨터 공학과 석사
1994년 미국 미시간대학교 전기 및 컴퓨터 공학과 박사
1996년~현재 연세대학교 전기전자공학과 부교수.
관심분야는 실시간 시스템, 인터넷 웹서버 기술, 디지털 시스템 고장포용 및 신뢰도 평가분야.