

<PROJECT 최종보고서>

Event Loop 을 이용한 WebServer 구현

Team 19

경영학과 박성우

컴퓨터공학부 이승현

컴퓨터공학부 데이비드

TmaxSoft 연구본부 WAS 실 1 팀 한승훈 연구원님

Table of Contents

1. Abstract.....	3
2. Introduction	3
3. Background Study	5
A. 관련 접근방법/기술 장단점 분석	5
B. Concurrent Programming.....	6
C. 프로젝트 개발환경.....	8
4. Goal/Problem & Requirements	8
A. Event Loop의 구현	9
B. HTTP Parser.....	9
C. Event Loop과 Thread Pool 간의 통신	9
D. Cache 구현.....	9
E. Event Loop을 사용하는 기존 서비스와의 비교.....	9
5. Approach	10
A. Read Request	10
B. Parse Request.....	10
C. Find Resource.....	11
D. Read File.....	11
E. Make Response.....	11
F. Write Response.....	11
6. Project Architecture	12
A. Architecture Diagram	12
B. Architecture Description.....	12
7. Implementation Spec.....	13
A. Input/Output Interface	13
B. Inter Module Communication Interface.....	13
C. Modules	14

8. Solution.....	17
A. Implementations Details.....	17
B. Implementations Issues	21
9. Results.....	25
A. Experiments.....	25
B. Result Analysis and Discussion	26
10. Division & Assignment of Work	27
11. Conclusion.....	27

1. Abstract

본 프로젝트의 목적은 event loop 모델을 적용하여 HTTP 요청을 처리하는 web server를 제작하는 것이다. 전통적인 web server의 경우, 각각의 클라이언트마다 서버 내의 worker thread를 하나씩 할당하고, 해당 thread가 하나의 클라이언트를 전담하여 처리한다. 전통적인 web server 구현의 경우 하나의 클라이언트와 하나의 thread를 대응시키는 방법이기 때문에, Connection Oriented Model이라고도 불린다. 하지만 worker thread pool에서 thread를 하나씩 할당하는 전통적인 기법의 경우, thread pool의 크기에 한계가 있기 때문에 동시에 처리할 수 있는 클라이언트의 수에 한계가 있다. 따라서 다량의 클라이언트가 동시에 접속을 시도할 경우 서비스가 마비되는 문제점이 있다. 한편, 수많은 클라이언트가 응답을 기다리고 있는 경우, blocking I/O로 인한 delay가 누적되어 서비스의 품질을 보장할 수 없다는 문제점도 존재한다. Event loop based model은 전통적인 방식과는 다르게 single thread loop를 통해 요청을 처리한다. 그 single thread는 서비스가 제공되는 동안 infinite loop를 돌면서 event queue에 쌓이는 event를 순서대로 수행한다. 이 과정에서 CPU가 대기해야 하는 file I/O 등의 작업들은 별도의 worker thread에게 넘기는 방식을 취한다. 결국 Event loop thread는 쉬지 않고 끊임없이 request를 처리할 수 있게 되어 매우 높은 처리 효율을 보인다. 본 프로젝트는 이러한 single threaded model의 이점을 통해 전통적인 web server 방식의 단점을 개선해보고자 한다.

또한 다량의 클라이언트가 동시에 접속하는 상황을 효율적으로 처리하기 위해 IO multiplexing 기법을 적용한 서버를 제작한다. 싱글 코어에서 웹 서버를 제작하는 데 있어 현재 IO multiplexing은 가장 효과적인 방식이라고 알려져 있으며, 현업에서도 활발히 쓰이고 있는 기술이다. Concurrency를 효율적으로 제공하기 위해 자바의 Selector와 Socketchannel, 또한 concurrent utility 라이브러리를 이용하여 IO multiplexing을 구현하고, 이를 바탕으로 현재 웹 상에서 상용되고 있는 서버에 버금가는 서비스와 속도를 제공할 수 있는 기본적인 서버를 작성해보고자 한다.

2. Introduction

시장 조사 업체 이마케터가 발표한 자료에 따르면, 현재 전세계 인구의 48.2%에 해당하는 36억 명이 한 달에 한 번 이상 인터넷을 사용한다. 인터넷 사용 인구의 성장은 인도와 인도네시아 같은 신흥 국가에서 급격한 성장세에 힘입어 증가율이 꾸준히 커지고 있다. 향후 모바일 광대역 연결이 활성화됨에 따라 앞으로 인터넷 사용자가 더욱 급격하게 늘어날 것으

로 전망된다.¹ 또한 시스코의 2015-2020 글로벌 전망 보고서 발표에 따르면 2020년에 세계 인터넷 사용자 수는 40억 명에 달할 것이며, IP 트래픽은 현재보다 3배 이상 증가할 것으로 예상된다². 이와 같이 인터넷 사용자가 지금보다 더욱 증가할 것으로 예상되는 가운데, 그 추세를 따라가기 위해 웹 서버도 발전을 거듭하고 있다. 웹 서비스를 이용하는 클라이언트가 지금처럼 많지 않았을 때 설계된 전통적인 모델에서는 동시 처리 가능한 클라이언트의 수에 한계가 있어 오늘날의 웹 서비스 이용자들을 견뎌내지 못한다. 이를 대비하여 클라이언트가 많아져도 그를 견딜 수 있는 웹 서버를 구축하기 위한 노력이 이어지고 있다. 일례로 러시아의 프로그래머 이고르 시쇼브는 Apache HTTPd를 이용하다 많은 수의 클라이언트가 들어오면 문제가 생기는 것을 인지하고, 이를 극복하기 위해 Nginx와 같은 새로운 웹 서버 프로그램을 개발하였다. 역사가 오래되지 않아서 점유율 면에서는 기존의 웹 서버에게 많이 밀리지만, 신규 서비스를 중심으로 점유율 상승에 가속이 붙는 중이다. 이미 전통적인 형태의 웹 서버 모델이 많은 한계를 보이고 있으므로 기존 사용자도 새로운 모델의 웹 서버로 이전하고 있는 추세이다. 새로운 모델의 웹 서버는 요청 당 스레드 혹은 프로세스 기반의 구조 대신 비동기 이벤트 기반의 구조로 작동한다. 현재 큰 인기를 얻고 있는 Nginx, Nodejs 등이 해당 구조로 이루어진 웹 서버 프로그램이며, 본 프로젝트에서도 해당 모델을 채택한 웹 엔진을 개발할 것이다. 본 프로젝트를 수행하기 위해 서버, 클라이언트의 네트워크 구조와 event driven program model, thread pool의 설계 능력이 필요하며, Node.js와의 성능 비교에서 80% 이상의 성과를 내는 것을 목표로 한다.

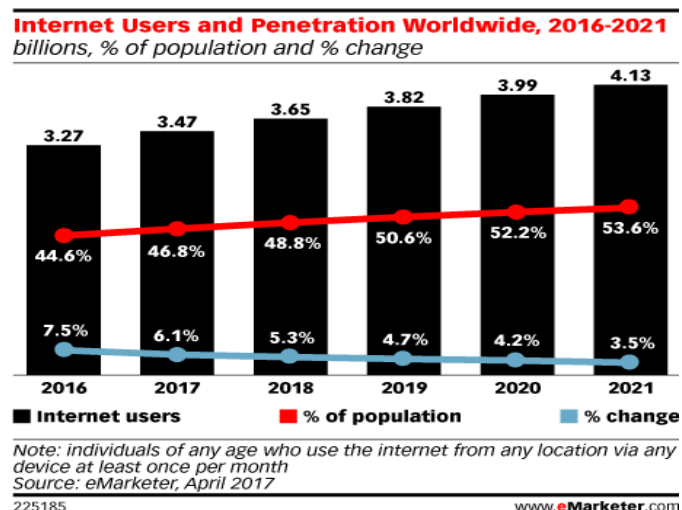


Figure 1 인터넷 사용자 현황

¹ <Internet Users and Penetration in Urban vs. Rural India>, eMarketer, 2018.02.20

² <Cisco Global Cloud Index 2015-2020>, GCI Forecast, 2016.06.13

3. Background Study

A. 관련 접근방법/기술 장단점 분석

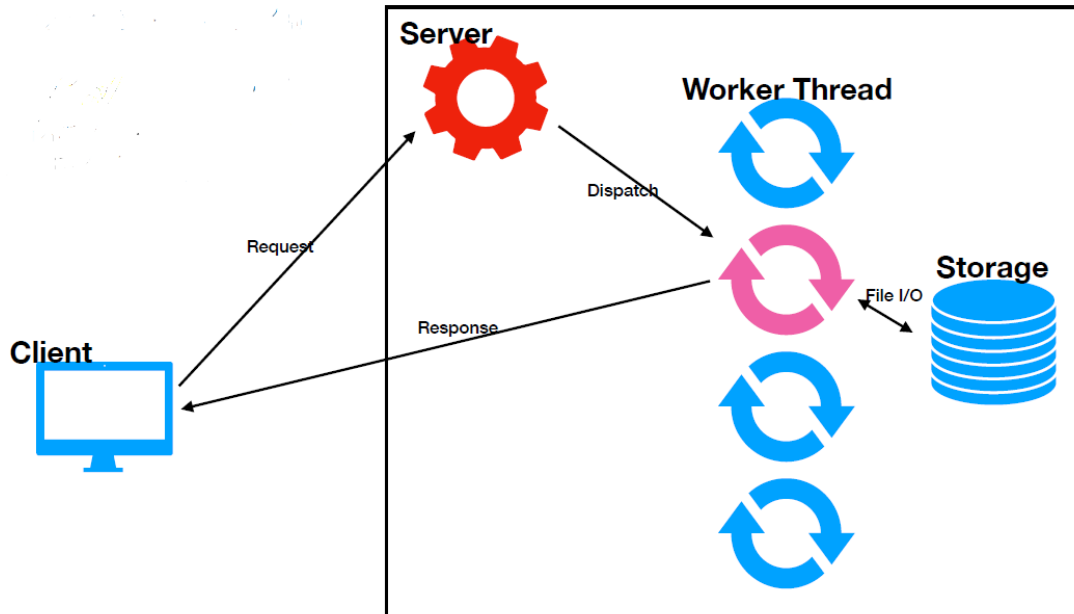


Figure 2 전통적인 Web Server의 Architecture

전통적인 web server는 Figure 1.에서와 같이 Connection Oriented Model의 구조를 가진다. 클라이언트가 들어올 때마다 서버에서 새로운 thread를 생성하고, 해당 thread에서 클라이언트가 작업을 종료할 때까지 모든 요청을 처리한다. File I/O 등의 CPU가 대기해야 하는 상황에서도 해당 thread는 그 요청이 끝날 때까지 대기해야 한다. 인터넷 사용자가 많아지고 동시에 처리해야 하는 클라이언트의 수가 증가함에 따라 C10K 문제가 화두로 떠올랐다. C10K 문제란 concurrently handling ten thousand connections의 약어로, 클라이언트 10,000명의 동시 접속을 현상을 의미한다. 수많은 클라이언트가 동시 접속하면 서버에서는 그 수에 맞게 worker thread를 생성해야 하는데, thread가 무한정 증가할 수는 없기 때문에 요청을 처리하는 데 한계를 보인다. 또한, worker thread가 제대로 생성되었다고 하더라도 수많은 클라이언트가 동시에 I/O 요청을 보낸다면 서버의 CPU가 하는 일 없이 대기만 하는 상황이 벌어질 수도 있다. 본 프로젝트에서 제작하는 웹 엔진은 event loop architecture를 가지고 있다. Event loop architecture는 single thread 기반으로 하나의 event loop thread가 서비스를 이용하는 모든 클라이언트를 담당한다. 하나의 thread가 여러 개의 요청을 처리하는 구조로 되어 있기 때문에 C10K 문제를 처리하기도 용이하다. File I/O 등의 작업과 같이 CPU가 일을 하지 못하고 대기해야 하는 종류의 작업은 비동기로 처리하여, 다른 작업을 수행하다가 I/O 처리가 끝나면 이벤트를 받아서 나머지 작업을 처리하는 방식이다. 이로 인해서, CPU가 I/O

응답을 기다리는 시간이 필요 없고, 대부분의 연산 작업에 사용되기 때문에 높은 효율성을 가질 수 있다. 또한, event loop과 I/O를 분리하여 context switching 비용을 줄일 수도 있다. 한편, 하나의 event가 CPU를 오랫동안 잠식하고 있다면, event queue에 보관된 다른 event 처리 시간에 영향을 미칠 수 있으므로, 전체 시스템의 성능이 저하되지 않도록 조치하는 것이 필요하다.

B. Concurrent Programming

클라이언트의 접속은 서버의 상황에 관계없이 발생한다. Iterative한 서버는 한 클라이언트의 접속에 대한 요청을 모두 처리한 다음에야 다음 클라이언트의 요청을 받는다. 하지만 이러한 웹 서버는 무수히 많이 접속하는 클라이언트의 모든 요청을 처리하기에는 속도 면에서 상당히 뒤쳐지는 모습을 보인다. 그렇기 때문에 웹 서버는 접속하는 클라이언트의 모든 요청들을 동시에 처리할 수 있는 구조를 필요로 한다.

Concurrent Programming을 구현할 때 동시에 요청을 처리함으로써 이루어질 수 있는 문제 상황을 모두 고려해야 한다. 먼저 두 가지 이상의 요청이 동시에 수행되면서 Race가 발생할 수 있다. Race는 각 요청의 수행 순서에 따라 결과가 바뀌는 현상을 의미한다. 프로그래머는 각 요청 중 어떤 것이 먼저 수행될 지 알 수 없고, 운영체제의 scheduling decision에 따라 그 순서가 달라지기 때문에 이 과정에서 Race가 발생하지 않도록 조심해야 한다. Deadlock을 방지하는 것도 중요하게 고려할 점이다. Deadlock이란 두 개 이상의 작업이 서로 상대방의 작업이 끝나기만을 기다리고 있는 상황으로 인해 결과적으로 아무것도 완료되지 못하는 상태를 의미한다. 이는 클라이언트의 요청을 더 이상 수행할 수 없게 만들기 때문에 서버의 운영에도 치명적인 영향을 미치게 된다. 이와 반대로 Livelock이란 두 개 이상의 작업이 계속해서 그들의 상태를 바꾸며 작업을 하지만 전체적으로 진행이 되지 않는 상황을 의미하는데, 이는 서버의 CPU는 계속 소비하면서 클라이언트의 요청을 제대로 수행하지 못하게 만든다. 또 다른 이슈로 starvation, fairness 등의 문제가 있을 수 있는데, 이는 일부 클라이언트의 요청만 계속 처리하지 못하는 공정성의 문제를 의미한다. 이와 같은 문제들로 인해 프로그래머는 적절한 scheduling decision이 내려질 수 있게 프로그램을 작성해야 하며, Shared variable들에 대해 Mutual Exclusion이 잘 이뤄질 수 있도록, 또한 프로그램이 여러 번 돌아가도 같은 결과를 낼 수 있도록 thread safe한 서버를 작성해야 한다.

Concurrent한 Flow를 생성하는 방법으로 3가지 방안을 고려할 수 있다. 첫번째는 클라이언트의 요청이 들어올 때마다 Process를 생성하는 것이다. 서버에서 클라이언트를 Accept할 때마다 Fork를 하여 새로운 Child 프로세스를 만들고, 해당 프로세스에서 클라이언트의 요청을 처리한다. 각 클라이언트는 독립된 프로세스에 의해 처리되며, 그들 간에 공유되는 상태

는 없다고 볼 수 있다. 이 방식은 세 가지 방식 중 가장 쉽고 간단한 방법이지만, 다른 방식들에 비해 Overhead가 큰 편이다. 두번째는 클라이언트의 요청이 들어올 때마다 Thread를 생성하는 것이다. 서버에서 클라이언트를 Accept할 때마다 Worker thread를 생성하고, 해당 Thread에서 클라이언트의 요청을 전담하여 처리한다. 이는 앞에서 언급한 전통적인 Web Sever의 구조와 같은 방식이다. 이 방식은 Process를 새로 만드는 방식보다 Overhead가 적다. Process를 생성, 삭제하는 것보다 Thread를 생성, 삭제하는 것이 훨씬 효율적이며, Context Switch 면에 있어서도 Thread가 Process보다 더 높은 속도를 보인다. 또한, Thread는 같은 주소 공간을 가지고 있기 때문에 서로 간에 데이터 공유가 쉽다. 하지만 이러한 장점이 큰 단점으로 나타나기도 한다. 프로그래머가 의도하지 않은 Data Sharing이 일어날 수도 있으며, 이는 서버가 의도한 결과를 완전히 왜곡하여 처리할 수도 있는 상황을 만든다. 이러한 상황을 해결하기 위한 디버깅도 굉장히 어렵다. 그렇기 때문에 Thread를 생성하는 방법은 이전 방법에 비해 효율적인 모습을 보이거나 프로그래머가 더욱 주의를 기울여야 한다. 마지막으로 IO Multiplexing을 통해 여러 클라이언트의 요청을 한 번에 처리하는 것이다. 서버가 active한 connection의 set을 가지고 있으며, 이러한 Set을 순회하면서 pending된 것들을 하나씩 처리하는 구조를 가진다. 단 하나의 Thread로 모든 클라이언트를 처리하기 때문에 Context Switch나 새로운 작업 생성, 삭제하는 데 드는 Overhead가 없다. 하지만 이 방식은 Process나 Thread를 기반으로 한 Design 방식에 비해 상당히 복잡하여 구현이 어렵다. 하나의 Thread로 처리하다 보니 Concurrency가 공정하게 돌아가지 못할 수도 있으며, Multicore의 이점을 활용하지 못한다. Multicore Computer가 상용화되면서 IO multiplexing을 활용하던 사람들도 점차 Thread Model로 회귀하는 모습을 보이기도 하였다. 하지만 코어 개수만큼의 Selector를 사용하여 IO multiplexing을 활용할 수 있도록 개선이 이루어지면서 다시 IO Multiplexing이 각광을 받고 있다.

C. 프로젝트 개발환경

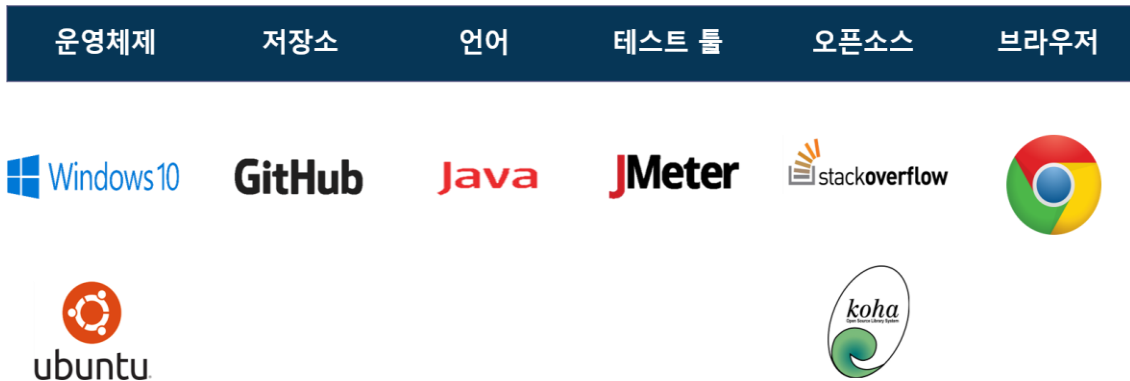


Figure 3 프로젝트 개발환경

3인 1조로 구성된 팀으로서 일의 진행 방향에 대한 회의도 중요하지만 코드 작성에 있어서의 협업 또한 중요한 요소가 될 것이다. 각자의 코드를 작성하고 이를 손쉽게 공유하며, 버전 관리에도 용이하도록 원격 저장소로 Github를 사용한다. 언어는 회사의 권유로 Java를 선택하였다. 후에 Node.js와의 성능 비교를 이용할 때 JMeter를 사용한다. 기본적인 요청과 그에 대한 응답은 모두 Chrome 브라우저에서 이루어진다. 서버는 기본적인 구현이 완료될 때까지 localhost를 이용하며, 구현이 완료되면 AWS의 EC2에 올릴 계획이다.

4. Goal/Problem & Requirements

A. Event Loop의 구현

첫째로, Event loop architecture의 기본인 event loop을 제대로 구현해야 한다. 간혹 비정상적인 클라이언트의 요청으로 인하여 전체 web server의 동작에 악영향을 미치는 경우가 있다. 이에 대해 read buffer를 확장하거나 request timeout 등의 exception handling을 따로 처리해야 한다. 뿐만 아니라 다수의 클라이언트 연결을 동시에 처리하기 위해 필요한 selector를 이용해야 할 것이다. 클라이언트의 요청에서 I/O 작업이 있다면 이를 non-blocking 방식으로 처리하도록 구현해야 한다. Selector를 사용하여 event 간의 multiplexing과 buffering에 대해서 이해하고 있어야 하며, 그것들이 모두 event loop에 구현되어야 한다. 또한, 해당 event loop은 기본적인 network event를 처리할 수 있어야 한다. Remote 클라이언트의 연결을 받아들이고 요청하는 파일을 읽고 쓰는 정도의 기본적인 구현이 되어 있어야 한다.

B. HTTP Parser

이번 프로젝트에서 사용하는 웹 브라우저는 하나의 브라우저로 한정하며, Chrome을 이용한다. Chrome이 보내는 기본적인 HTTP request에 대해 해석할 수 있는 기능이 있어야 한다. Web browser가 보낸 HTTP request의 header를 파싱하고, method, URI, protocol에 대한 정보를 추출할 줄 알아야 한다. 이에 더하여 connection header의 상태를 해석하고, 그 상태에 따라 연결을 끊지 않고 계속해서 요청을 처리할 수 있도록 하거나 요청을 처리한 후에 연결을 끊는 동작을 구현해야 한다. HTTP request를 처리하고 난 후에는 web browser가 해석할 수 있는 HTTP response를 구성해야 한다. 이를 위해서 해석을 마친 request에 대해 올바른 HTTP response의 형태가 무엇인지 이해하고 예외 상황들을 고려해야 한다.

C. Event Loop과 Thread Pool 간의 통신

Event loop과 thread pool의 역할을 구분하고, 필요한 경우에는 양자간에 통신을 하는 방법에 대해서 이해하고 구현해야 한다. Event loop 내에서 처리하기에 적절하지 않은 task에 대해서 이해하고 이를 thread pool에 맡겨 처리하며, 처리된 결과가 새로운 event로서 event loop 내에서 처리되도록 해야 한다.

D. Cache 구현

Web browser가 요청한 파일에 대해서 매번 디스크에 접근을 하여 읽게 된다면 빠른 속도를 내지 못한다. 이를 해결하기 위해 web cache를 구현한다. 현실적인 관점에서 우리가 사용할 수 있는 memory의 양이 무한하지 않기 때문에 모든 file을 다 memory에 올릴 수 없다. 따라서 cache의 memory 사용량을 제한할 수 있어야 한다. Memory의 사용량을 제한한다고 하면 결국 새로운 cache entry가 추가될 때 기존의 cache entry 하나를 제거해야 하는데, 이 때 제거할 entry를 고르는 방식도 LRU, FIFO를 비롯하여 여러가지가 있을 수 있으므로 각각에 대해 모두 테스트를 진행하도록 한다. 그 외에 파일의 크기가 너무 클 경우 cache에 올리는 것 자체가 비효율적일 수 있으므로 cache에 들어갈 수 있는 파일의 크기를 제한하는 방법도 생각해본다.

E. Event Loop을 사용하는 기존 서비스와의 비교

Event loop을 사용하는 대표적인 서비스인 Node.js와의 비교를 통해 우리가 만든 웹 서버의 성능을 평가한다. 텍스트 형태의 HTML 페이지를 JMeter를 통하여 반복 호출하는 테스트를 수행하며, 해당 서비스 성능의 80% 이상의 성과를 낼 수 있도록 한다.

5. Approach

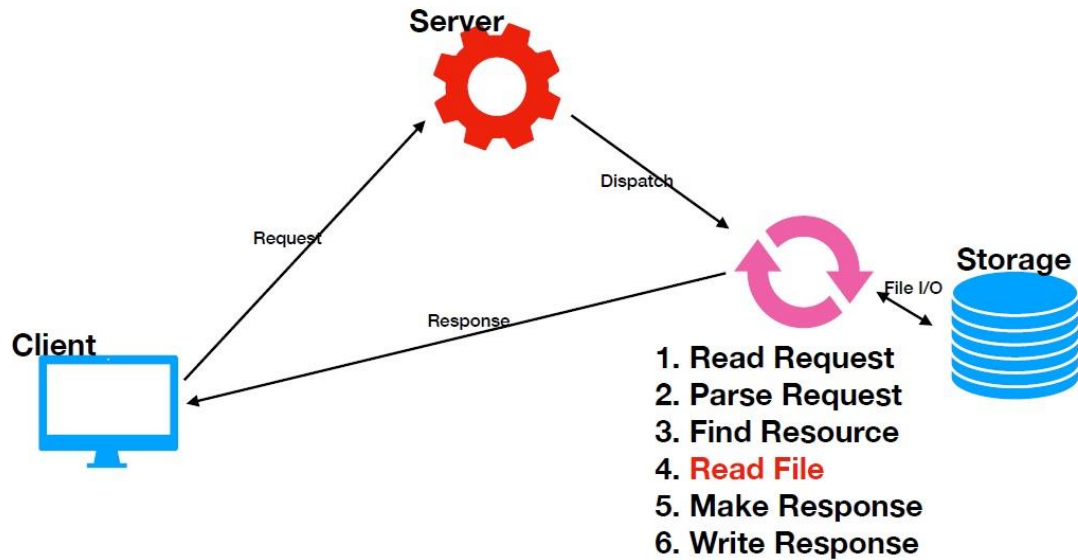


Figure 4. Approach

본 프로젝트에서 구현하는 웹 서버의 작업을 크게 6가지의 단계로 나눌 수 있을 것이다. Connection Oriented Model에서는 클라이언트를 전담하는 thread가 배정되어 있기 때문에 여섯 가지 일을 단순히 순서대로 수행만 하면 된다. 이와 달리 event loop model에서는 하나의 event loop thread가 모든 클라이언트를 담당하기 때문에 각각의 job은 연속적으로 수행될 수 없을 수도 있으며, event에 의해 적절히 통제되어야 한다.

A. Read Request

첫째로, 클라이언트가 Web browser에서 보낸 HTTP request를 읽어야 한다. 서버에서 소켓을 열어둔 채로 클라이언트를 기다리며, 요청이 왔을 때 이를 String 형태로 받아들인다.

B. Parse Request

클라이언트의 요청에 의해 String 형태로 받아들인 HTTP request를 해석하는 단계이다. Request가 어떤 HTTP 명령을 내리는지, 어떤 파일을 요청하고 있는지, request의 header에 포함된 정보는 어떤 것들이 있는지 판단한다. 이때, 클라이언트가 비정상적인 request를 보냈다면 뒤의 단계를 거치지 않고, 바로 exception handling을 통해 오류를 출력할 수 있게 한다.

C. Find Resource

Parsing에 성공한 HTTP request에서 어떤 파일을 요구했는지 확인했다면, 해당 파일이 존재하는지 확인한다. 파일이 존재한다면 파일을 읽는 단계로 진행하고, 존재하지 않는 파일이라면 오류를 출력할 수 있게 한다.

D. Read File

클라이언트가 요청한 파일에 대해 읽는 작업을 수행한다. 이 과정은 file I/O 작업으로서 CPU가 대기해야 하는 작업이다. 이 작업은 event loop thread에서 처리해주지 않고 별도의 worker thread에 처리를 위임해서 CPU가 일하지 않는 상황을 피하도록 해준다. Event loop thread는 작업을 위임한 후 event queue에 존재하는 다른 event들을 처리해줄도록 한다. 별도의 worker thread에서 I/O 작업이 끝나게 되면 event queue에 처리한 결과를 바탕으로 새로운 event가 등록되며, 후에 event loop thread에 의해 response를 내는 방식으로 처리된다.

E. Make Response

클라이언트의 request에 대해 모든 처리를 끝내고 난 후 적절한 response를 구성한다. 상황에 알맞은 HTTP status code와 date 등의 response header 정보와 함께 각 오류에 대한 페이지 내용을 작성한다.

F. Write Response

작성한 내용을 클라이언트에게 전송한다. Header 정보를 포함하여 읽은 파일과 함께 클라이언트에게 보여주는 작업이다.

6. Project Architecture

A. Architecture Diagram

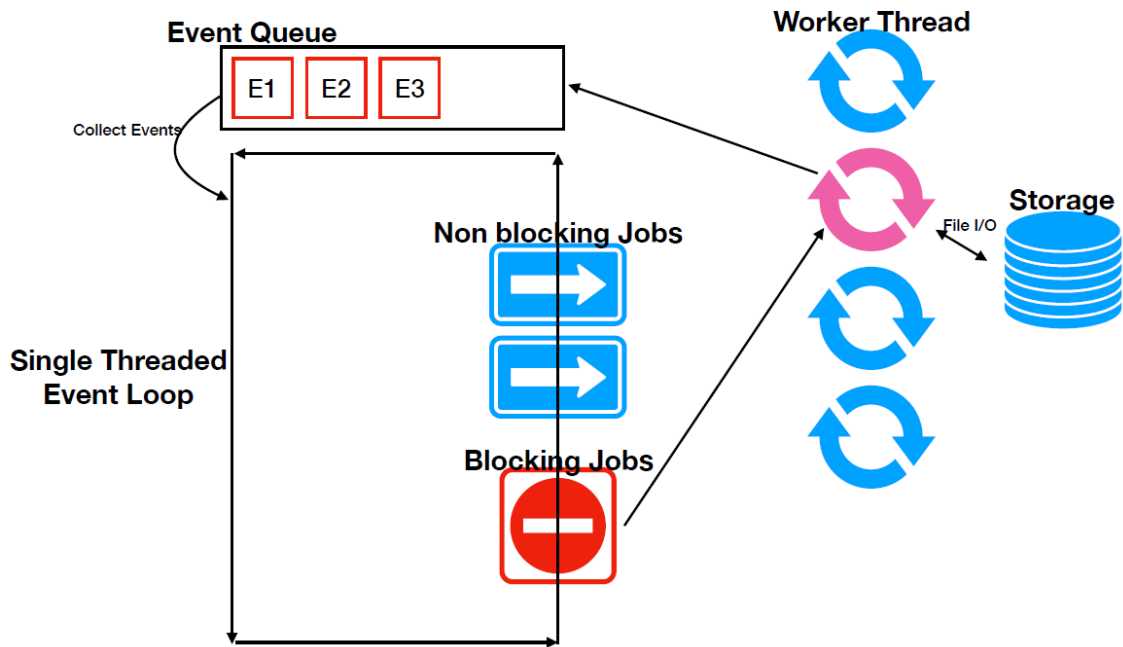


Figure 5. Architecture

B. Architecture Description

위의 그림은 본 프로젝트의 전체적인 구조를 대략적으로 나타낸 것이다. 해당 모델은 크게 single threaded event loop, event queue, thread pool의 세 가지로 나뉠 수 있다. 먼저 event queue는 수행해야 할 event들이 대기하는 장소로서, 클라이언트의 요청이 들어오면 새로운 event가 enqueue되며, 같은 클라이언트의 요청에 대해서도 앞서 말한 단계에 따라 세 부분으로 event의 정보가 달라질 수 있다. Single threaded event loop은 모든 클라이언트의 요청을 처리하는 thread로서 해당 architecture의 핵심 역할을 한다. Event queue에서 event를 하나씩 dequeue하여 요청에 대한 정보를 받고 그에 해당하는 응답을 내놓는다. 이 때, file I/O와 같은 blocking job을 만나면 해당 event를 잠시 미루고, non-blocking job인 다른 event를 dequeue하여 먼저 처리하도록 한다. Thread pool은 worker thread를 모은 집합이다. Blocking job이 일어날 때마다 새로운 worker thread에게 이 일을 전담하여 처리하게 되는데, 이렇게 생성되는 thread를 바로 가져다가 쓸 수 있도록 thread pool을 이용한다. File I/O 작업을

처리한 worker thread는 작업을 종료하고, 종료되었다는 event를 event queue에 enqueue하여 후에 event loop에서 올바르게 처리되도록 한다.

7. Implementation Spec

A. Input/Output Interface

서버에 들어오는 HTTP request의 버전은 1.1에 국한된다. Network, DB 접근을 제외한 file system의 정적인 file에 대한 접근이기 때문에 HTTP method도 GET만이 들어온다. GET만 허용하기 때문에 이를 명확히 하기 위해 다른 method가 들어왔을 때 405 Method Not Allowed로 응답한다. 요청이 정상적으로 들어온 경우 200 OK, 파싱할 수 없는 요청의 경우 400 Bad Request, 요청 받은 자원이 존재하지 않을 경우 404 Not Found로 각각 상황에 맞는 응답을 나타내도록 한다. 브라우저로 새로 고침 등을 할 경우에 똑같은 File을 다시 보내지 않기 위해 304 Not Modified로 응답하도록 한다.

B. Inter Module Communication Interface

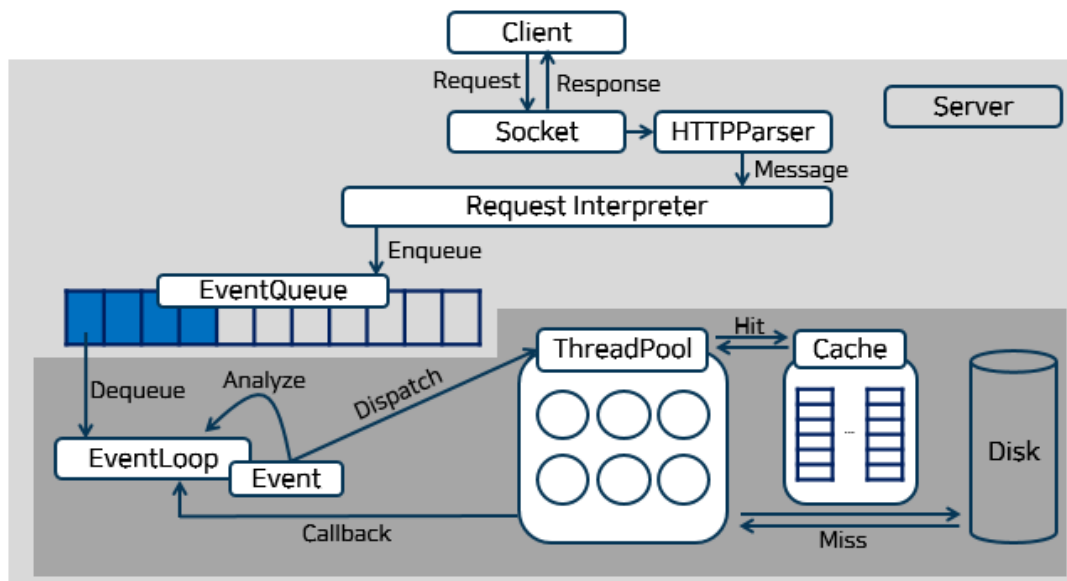


Figure 6. Module Communication

Module 간의 communication은 event 중심으로 이루어진다. 서버 소켓을 처음 열어놓은 main 모듈에서 클라이언트가 들어왔다는 것을 인지하면 해당 요청을 HTTPParser로 넘겨준다. 넘겨받은 정보를 바탕으로 파서는 해석을 진행하고 새로운 event를 구성하여

EventQueue로 enqueue한다. Event loop thread가 현재 처리하고 있는 event가 없다면 EventQueue에서 event를 dequeue하여 EventLoop으로 넘겨준다. Event를 처리하면서 I/O 작업이 필요하다면 I/O 작업만을 처리하는 thread에게 일을 넘겨준다. 이 과정에서 필요한 파일이 cache에 존재한다면 빠르게 불러오고, 없다면 파일을 저장하고 읽는 작업을 진행한다. I/O Thread에서 작업이 끝났다면 끝난 작업의 정보를 바탕으로 event를 생성하여 EventQueue에 다시 enqueue한다.

C. Modules

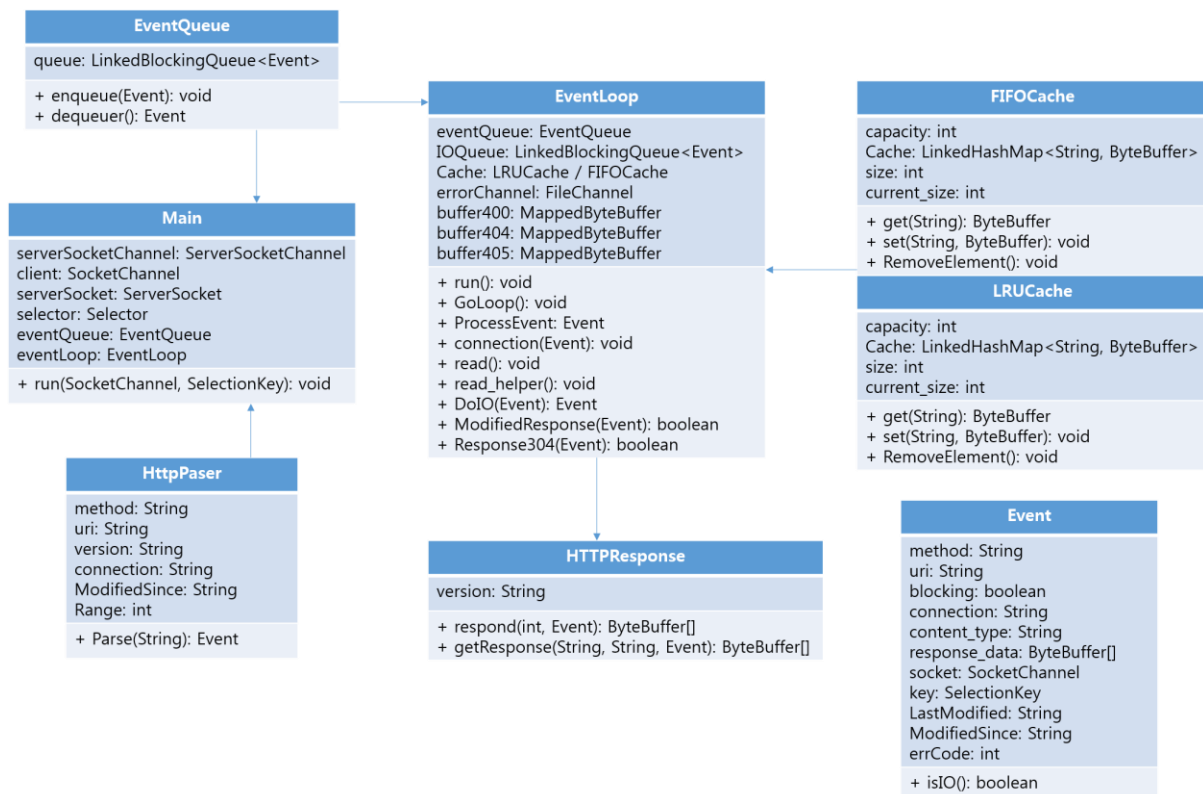


Figure 7. Modules Relation

① Event

Event는 클라이언트 요청의 정보를 담고 있는 객체이다. Main에서 클라이언트의 요청을 받으면 HttpParser는 이를 해석한다. 해석한 정보를 바탕으로 Event를 구성하고, 이를 EventQueue에 넣는다. 넣어진 Event는 EventLoop에서 처리된다.

Event는 클라이언트 요청의 method, uri를 포함하며, I/O 작업 여부, HTTP version 정보, Content type, 클라이언트에게 응답할 데이터, 연결된 소켓 정보와 Selector로부터 받은 Selection Key, 응답할 파일의 변경 정보, 클라이언트가 가지고 있는 파일 변경 날짜, (에

러가 있다면) 에러 코드 등으로 구성된다.

② HttpParser

```
GET /page.html HTTP/1.1
Host: localhost:9090
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br
Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: csrftoken=h7gySH952XU51obSRtRLSGj8yj1ova1RDPYYj0BAc4T6RIg2PmWIGLLa8feijFB3
```

Figure 8 Client Request Format

HttpParser는 Main에서 받아들이는 클라이언트의 요청을 파싱하여 Event를 만든다. 웹 브라우저는 요청을 보낼 때 GET /page.html HTTP/1.1 의 형태로 시작하여 여러 가지 Request header를 서버 쪽에 전송한다. 이를 바탕으로 Method, URI, Version 정보를 파싱하고 추가적인 헤더 정보를 파악하여 Event 객체를 생성한다.

③ EventQueue

자바의 라이브러리를 사용하여 Queue를 구현한다. 이 때 Queue는 일반적인 Queue가 아니라 java.util.concurrent 라이브러리 내에 있는 BlockingQueue를 사용한다. BlockingQueue는 일반 Queue와 달리 dequeue할 때 큐가 비어 있으면 새로운 원소가 들어올 때까지 기다린다(Block). Enqueue할 때도 마찬가지로 size가 꽉 차 있으면 자리가 날 때까지 기다렸다가 enqueueer 된다. BlockingQueue도 LinkedBlockingQueue, ArrayBlockingQueue 등 다양한 라이브러리가 존재하지만 클라이언트 요청의 사이즈가 정해져 있지 않았기 때문에 LinkedBlockingQueue를 사용하였다.

④ FIFOCache, LRUCache

URI를 바탕으로 응답할 데이터를 반환하는 LinkedHashMap을 만들어 Cache를 구현하였다. Cache의 전체 용량은 스펙 상 자유자재로 구현 가능하여 기본적으로 50MB의 크기로 선언하였으며, 전체 원소 개수는 50개로 제한하였다. 전체 용량이 제한이 되어 있다 보니 용량이 너무 큰 파일의 경우 캐시에 적재하는 것이 오히려 해가 되는 경우가 있다. 여러 번 접근하는 작은 파일들을 캐시에 담지 못할 위험이 있기 때문이다. 이를 대비하여 개별 원소 또한 10MB 이상이 되면 캐시에 담지 못하게 하였다.

FIFOCache는 사이즈가 꽉 차 있거나 원소 개수가 최대에 달한 경우 LinkedHashMap의 마지막 원소를 지운다. LRUCache도 마찬가지로 마지막 원소를 지우는데, 대신 LRUCache

는 우선순위를 유지하기 위해 원소를 Get할 때 LinkedHashMap에서 원소를 지웠다가 다시 넣어주는 작업을 진행한다.

⑤ HTTPResponse

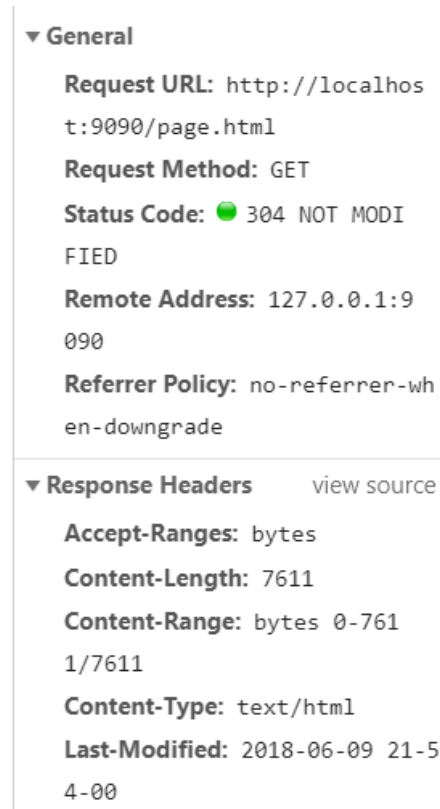
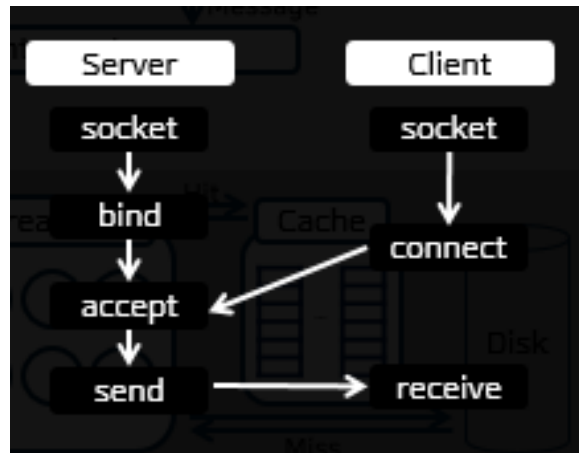


Figure 9 Response in Chrome

EventLoop에서 처리한 결과를 바탕으로 클라이언트에게 직접 보낼 ByteBuffer를 생성한다. 반환된 ByteBuffer의 첫 번째 원소는 Response Header 정보가 들어가고, 두 번째 원소로 데이터가 들어간다.

⑥ Main



ServerSocket을 선언하여 주소와 binding을 한 뒤 클라이언트의 접속을 Accept한다. 이 때, 서버 소켓은 nonblocking으로 설정한다. IO Multiplexing을 구현하기 위해 Selector를 사용하여 Read Key, Write Key, Accept Key마다 다른 작업을 수행할 수 있도록 구현하였다. 접속을 요청하여 Accept된 Client는 Accept Key를 가지고 Selector에 들어간다. Acceptable한 key를 처리하는 과정에서 이는 다시 Read Key를 가지고 Selector에 등록된다. Read Key를 가지는 작업들은 run 메소드를 통해 클라이언트의 요청을 서버로 가져와서 하나의 Event 객체를 생성한다. 생성된 Event 객체는 응답할 데이터를 포함하여 Write Key를 가지고 Selector에 다시 등록되며, Writable한 key를 처리하는 과정에서 해당 데이터를 클라이언트에게 전송한다.

⑦ EventLoop

클라이언트의 요청을 파싱하여 생성된 Event의 Response를 구성하는 모듈이다. 해당 모듈 안에서 총 2개의 Thread가 돌아간다. 하나는 EventQueue에서 Event를 dequeue하여 I/O 작업을 하는지 판단하는 Thread가 실행되며, 나머지는 I/O Queue에서 Event를 dequeue하여 실제 파일을 읽어내는 일을 한다.

8. Solution

A. Implementations Details

① HttpParser Module

Event Parse(String request)	클라이언트로부터 받은 전체 Request를 개행 문자 단위로 분리한다. 첫 번째 라인을 띄어쓰기 단위로 다시 나누어 method, uri, version을 추출한다. 이 과정에서 하나라도 파싱 불가능한 것이 있다면 Event를 생성할 때 에러 코드를 400으로 설정한다. (Error code 400: Bad Request). 나머지 라인들에서 header가 Connection, If-Modified-Since, Range인 것들을 파싱하여 알맞은 String 변수에 각각 넣는다. 입력된 정보들을 바탕으로 Event 객체를 생성하고 이를 반환한다.
-----------------------------	--

② FIFOCache Module

ByteBuffer get(String uri)	LinkedHashMap의 get 메소드를 사용하여 uri를 key로 가지고 있는 value를 반환한다.
void set(String uri, ByteBuffer value)	캐시에 넣으려는 데이터가 10MB 이상이라면 함수를 종료한다. 현재 사이즈(current_size)와 넣으려는 데이터의 사이즈를 더한 값이 캐시의 사이즈(50MB)보다 크다면 RemoveLastElement() 함수를 실행한다. 그보다 작아질 때 캐시에 데이터를 put하고 현재 사이즈를 업데이트한다.
void RemoveLastElement()	LinkedHashMap의 가장 끝 원소를 제거한다. 현재 사이즈(current_size)에서 제거한 원소의 크기를 뺀다.

③ LRUCache Module

ByteBuffer get(String uri)	LinkedHashMap의 get 메소드를 사용하여 uri를 key로 가지고 있는 value를 가져온다. 그리고 가져온 값을 LinkedHashMap에서 remove하고 다시 해당 uri와 value로 LinkedHashMap에 put한다. 모든 작업을 완료한 후 value를 반환한다.
void set(String uri, ByteBuffer value)	캐시에 넣으려는 데이터가 10MB 이상이라면 함수를 종료한다. 현재 사이즈(current_size)와 넣으려는 데이터의 사이즈를 더한 값이 캐시의 사이즈(50MB)보다 크다면 RemoveLastElement() 함수를 실행한다. 그보다 작아질 때 캐시에 데이터를 put하고 현재 사이즈를 업데이트한다.
void RemoveLastElement()	LinkedHashMap의 가장 끝 원소를 제거한다. 현재 사이

	즈(current_size)에서 제거한 원소의 크기를 뺀다.
--	-----------------------------------

④ HTTPResponse Module

ByteBuffer[] respond(int code, Event event)	EventLoop에서 response data를 받기 위해 가공된 Event와 Status Code를 바탕으로 적절한 Status Message를 생성하며, 이들을 바탕으로 getResponse 메소드를 호출하여 Response Data를 반환한다.
ByteBuffer[] getResponse(String code, String status, Event event)	Status code, Status Message, HTTP version으로 Response Line을 만든다. 또한, 클라이언트에게 전송할 데이터의 사이즈를 바탕으로 Content-Length, Content-Type, Content-Range, Last-Modified에 대한 Response Header 정보를 완성하고, Response Line과 함께 ByteBuffer 형태로 합친다. Argument로 넘겨받은 Event에서 클라이언트에게 실제로 응답할 데이터와 이전에 합친 ByteBuffer로 구성된 새로운 ByteBuffer Array를 반환한다.

⑤ EventLoop Module

void run()	GoLoop() 메소드를 무한히 실행시킨다.
void GoLoop()	먼저 EventQueue에 있는 Event를 dequeue한다. 해당 Event가 I/O 작업이 필요한 상태라면 Event의 Response_data를 채우기 위해 Cache를 검사한다. Cache에 해당 uri에 해당하는 value가 없다면 Event를 IOQueue에 enqueue하고 함수를 끝낸다. Cache에 값이 존재한다면 바로 읽어들이 Event의 response data로 설정한 뒤 클라이언트에게 응답할 수 있도록 ProcessEvent 메소드를 실행한다. I/O 작업이 필요한 상태가 아니라면 (response_data가 이미 채워져 있거나 파일을 따로 요구하는 것이 아니라면) 바로 ProcessEvent 메소드를 실행한다.
void ProcessEvent(Event event)	Argument로 전달받은 Event의 errCode가 0이 아니면 해당 errCode와 event를 바탕으로 HTTPResponse 모듈의 respond 메소드를 호출하여 클라이언트에게 응답할 데이터를 받아온다. errCode가 0이라면 Status Code를 판단하는 Response304 메소드를 호출하여

	<p>code를 받아오고, 해당 코드와 event를 바탕으로 HTTPResponse 모듈의 respond 메소드를 호출하여 클라이언트에게 응답할 데이터를 받아온다.</p> <p>받아온 데이터를 소켓 채널을 통해 클라이언트에게 Write한다. 만약 Write한 데이터의 크기가 실제 데이터의 크기보다 작은 경우, event의 Response_data를 쓰고 있는 데이터로 변경한 후 event의 SelectKey를 Write Key로 변경한다.</p>
void connection(Event event)	<p>ProcessEvent 메소드가 작업을 완료한 후 항상 같이 불리는 메소드다. 클라이언트에서 Connection Header를 keep-alive로 보내지 않았으면 모든 작업을 완료한 후 소켓을 닫는다. 또한, Event가 에러를 발생시켰다면 모든 작업을 완료한 후 항상 소켓을 닫도록 한다.</p>
void read()	<p>read_helper() 메소드를 무한히 실행시킨다.</p>
void read_helper()	<p>먼저 IOQueue에 있는 Event를 dequeue한다. 해당 Event를 바탕으로 DoIO() 메소드를 실행하고, 반환된 Event를 바탕으로 ProcessEvent() 메소드를 실행시킨다.</p>
Event DoIO(Event event)	<p>event의 method 값이 GET이 아닌 경우 event의 errCode를 405로 설정한다. 또한, event에서 요구하는 File이 존재하지 않는 경우 errCode를 404로 설정한다. 만약 파싱이 되지 않아 이미 errCode가 400인 경우 따로 처리를 하지 않고 넘어간다. 언급한 모든 경우에 대해 I/O 작업이 끝났으므로 event의 blocking 필드를 false로 설정해줌으로써 해당 event의 I/O가 끝났다는 것을 알리고 그 event를 반환한다.</p> <p>정상적인 요청인 경우 ModifiedResponse() 메소드를 통해 event의 LastModified 필드를 업데이트하고, 클라이언트가 요청하는 파일을 ByteBuffer로 읽는다. 읽은 ByteBuffer는 event의 response_data에 포함된다. 또한, Cache의 set 메소드를 불러 읽은 데이터를 캐시에 넣는 작업도 진행한다. 모든 작업이 끝났다면 위와 마찬가지로 event의 blocking 필드를 false로 설정하고 그 event를 반환한다.</p>
boolean ModifiedResponse(Event event)	<p>Argument로 전달받은 event의 uri를 바탕으로 파일의 이름을 받아오고, 그 이름에 해당하는 파일이 있는지 검사한다. 파일이 없다면 false를 반환한다. 파일이 있다면 해당 파일의 마지막 수정 날짜를 "yyyy-MM-dd HH-mm-ss" 형태로 불러와 event의 LastModified 필</p>

	드를 업데이트한다. 해당 필드를 업데이트한 후 true를 반환한다.
boolean Response304 (Event event)	클라이언트의 If-Modified-Since 헤더와 event의 Last-Modified 필드를 비교한다. 클라이언트가 알고 있는 마지막 수정 날짜와 파일의 최종 수정 날짜가 같다면 true를 반환하고, 아니면 false를 반환한다.

⑥ Main Module

void main(String[] args)	<p>서버를 열 IP와 port 번호를 지정하고 해당 주소에 서버 소켓을 연결한다. 서버 소켓을 nonblocking으로 설정하고 Selector에 OP_ACCEPT key를 걸어 놓는다. 이를 바탕으로 접속한 클라이언트는 Selector 내에서 Accept key를 지니게 된다. 기본적인 설정을 완료한 후 event-Loop를 미리 실행시킨다.</p> <p>Selector에서 key를 꺼내는 작업을 무한히 실행시킨다. 만약 Selector에 아무 key도 없다면 block되지만, 접속하는 클라이언트가 있다면 내부가 실행된다. 꺼낸 key가 Acceptable하다면 클라이언트 소켓을 받아와 해당 클라이언트를 Selector에 Read Key와 함께 등록시킨다. 꺼낸 key가 Readable하다면 해당 key와 클라이언트 소켓 정보를 바탕으로 run() 메소드를 실행시킨다. 꺼낸 key가 Writable하다면 Write할 ByteBuffer를 다 쓸 때까지 key를 바꾸지 않고 계속 쓰는 작업을 반복하다가, 모든 작업을 완료했을 때 해당 클라이언트를 다시 Read Key로 바꿔주고 EventLoop의 connection() 메소드를 실행시킨다.</p>
void run(SocketChannel client, SelectionKey key)	<p>client로부터 데이터를 읽는다. 읽은 데이터의 크기가 -1이라면, 읽는 과정에서 에러가 발생한 것이므로 client를 close한다. 그렇지 않다면 읽은 데이터를 바탕으로 HttpParser의 Parse() 메소드를 실행하여 Event를 생성한다. 생성된 Event의 socket 필드를 client로, key 필드를 key로 설정한 후, 해당 Event를 eventQueue에 enqueue한다.</p>

B. Implementations Issues

① HTTP Response Status

구현한 HTTP Status code는 총 5가지로, 200 OK, 304 NOT MODIFIED, 400 BAD REQUEST, 404 NOT FOUND, 405 NOT ALLOWED가 있다. 각각 정상적으로 응답이 온 경우, 요청한 파일이 마지막 요청 이후로 바뀌지 않은 경우, 서버에서 파싱할 수 없는 요청을 보낸 경우, 서버에 존재하지 않는 파일을 요청한 경우, GET method가 아닌 다른 method를 부른 경우를 의미한다. 이 중 304 NOT MODIFIED는 클라이언트가 해당 파일의 마지막 수정 날짜를 확인할 수 있어야 하기 때문에 서버에서 'Last-Modified'라는 특정한 헤더를 클라이언트에게 보내줘야 한다. 이 헤더를 받은 클라이언트는 후에 같은 파일을 서버에 요청할 때 헤더에 'If-Modified-Since' 항목을 추가한다. 서버는 이 헤더를 받아 추가적인 파싱을 진행하여 정상적인 응답을 한다고 하더라도 이를 304로 보낼지 200으로 보낼지 판단할 수 있게 된다.

② 대용량 파일 처리



Figure 10. 10 MB image

Non-blocking으로 진행되기 때문에 큰 용량을 가진 파일의 경우 한 번에 write가 불가능한 경우가 종종 있다. 클라이언트의 요청은 대부분 2KB 이내이기 때문에 한 번에 read를 할 수 있는 양이지만, 수 MB 단위의 파일을 write하는 경우 다 써질 때까지 Blocking하지 않고 해당 작업을 반복해서 이벤트로 꺼내와 다시 써주는 작업을 진행해야 한다. 이 부분을 처리해주지 않으면 클라이언트에게 제대로 쓰이지 못한 파일이 전송되고 결과적으로 보여지는 것은 아무것도 없게 된다.

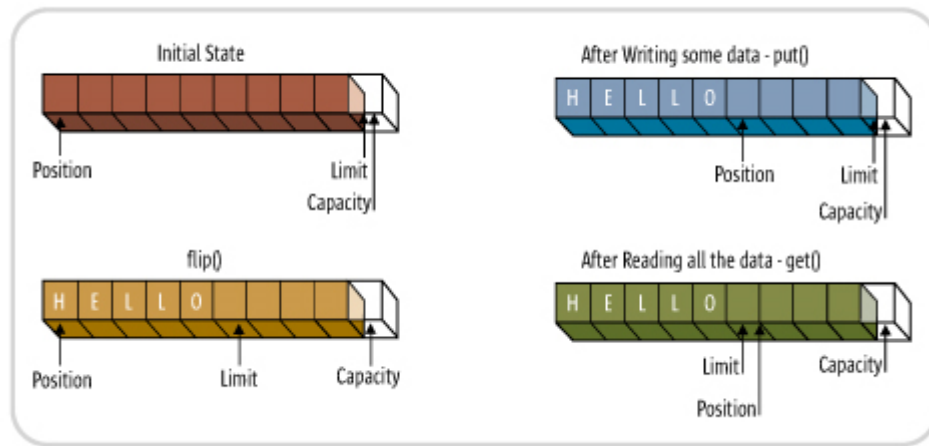


Figure 11. Java ByteBuffer Attribute

이를 처리하기 위해 Java ByteBuffer 자료형의 특성과 SelectKey가 제공하는 메소드를 적극적으로 활용하였다. 클라이언트에게 응답할 response_data는 ByteBuffer 형태로써, Event 객체마다 가지고 있는 필드이다. 이를 처음 썼을 때의 사이즈를 계산하여 그 사이즈가 해당 ByteBuffer의 capacity 혹은 limit과 같다면 작업을 종료하고, 그것이 아니라면 Write 이벤트를 다시 등록해주는 과정을 거쳐야 한다. 이 때, ByteBuffer는 Write 하면서 자신의 position을 자동으로 업데이트하는데, 누적되는 사이즈 계산이 복잡하고 어려운 경우 ByteBuffer의 position과 limit를 비교하여 Write가 끝났는지 확인하는 방법도 있다. Write 이벤트를 등록하는 것은 Event의 key 값을 Write key로 변경하는 것이다. 그리고 후에 Selector가 Write Key를 꺼낸 경우 그 때 마저 쓰이지 못한 ByteBuffer를 해당 client에게 전송하는 것이다.

이를 위해서는 해당 Key에서 client와 쓰일 ByteBuffer 값을 모두 갖고 있어야 한다. 자바의 SelectKey는 Object를 attach할 수 있는 메소드를 갖고 있다. 이 메소드를 활용하여 Write 이벤트를 등록하는 시점에 해당 key에 처리되고 있는 event 객체를 그대로 attach하였다. 그리고 후에 이 key가 처리될 때 다시 key에 attach되어 있는 객체를 Event 자료형으로 불러와 남은 데이터를 계속해서 써내려 가도록 처리하였다. 처리 결과 30MB 이상의 이미지 파일도 정상적으로 렌더링되는 것을 확인하였다.

③ Range Header

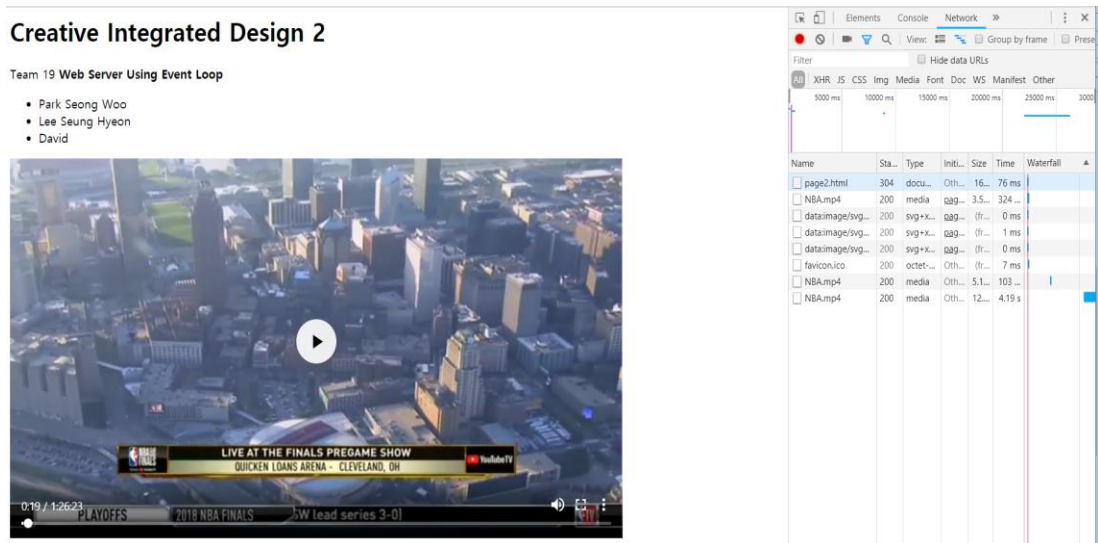


Figure 12. 600 MB Video(1h 30m)

대용량 비디오 파일의 경우 이미지와 달리 한 번에 파일 데이터 전체를 요구하지 않는다. 일반 파일과 달리 Request에 'Range'라는 헤더가 추가돼서 계속적으로 요청이 들어온다. 이 Range 헤더를 처리해주지 않으면 서버 입장에서는 비효율적인 작업을 계속하게 된다. 사용자가 특정 시간의 영상을 보기 위해 재생바를 클릭한 경우 웹 브라우저는 해당 위치에 해당하는 지점부터 비디오를 달라는 요청을 Range 헤더를 통해 보내지만, 서버는 영상의 처음부터 그 지점까지 모든 데이터를 다 읽어낸 후에야 클라이언트에게 제대로 응답을 할 수 있게 된다.

```
GET /NBA.mp4 HTTP/1.1
Host: 127.0.0.1:9090
Connection: keep-alive
Accept-Encoding: identity;q=1, *,q=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36
Accept: */*
Referer: http://127.0.0.1:9090/page2.html
Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: csrftoken=A2A1MAGGRti9AG8ucNhjXj3HU14hd2oeJb6X1BSB2sUi7N74kwcPB9YcR4YA4JA8r
Range: bytes=3211264-

5898195
GET /NBA.mp4 HTTP/1.1
Host: 127.0.0.1:9090
Connection: keep-alive
Accept-Encoding: identity;q=1, *,q=0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36
Accept: */*
Referer: http://127.0.0.1:9090/page2.html
Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
Cookie: csrftoken=A2A1MAGGRti9AG8ucNhjXj3HU14hd2oeJb6X1BSB2sUi7N74kwcPB9YcR4YA4JA8r
Range: bytes=10420224-
```

Figure 13. Range Header

이를 방지하기 위해 Range 헤더가 들어오면 이를 따로 파싱하여 그 정보를 Event 객체에 필드로 저장하도록 구현하였다. 그리고 파일을 읽을 때, FileChannel의 map 메소드와 MappedByteBuffer 자료형을 사용하여 Range Header가 나타내는 시작점부터 파일을 읽을 수 있도록 구현하였다.

9. Results

A. Experiments

2가지로 실험을 진행하였다. 첫번째는 웹 브라우저에서 직접 파일을 띄우는 형태로 진행하였고, 두번째는 Jmeter를 사용하여 부하 테스트를 진행하였다. 웹 브라우저에서는 크기가 작은 수 KB 단위의 파일부터 수십 MB에 달하는 이미지까지 제대로 렌더링되는 것을 확인하였다. 또한, Figure 12의 그림처럼 재생 시간이 1시간이 넘는 크기가 아주 큰 비디오 파일들에 대해서도 무리 없이 재생되는 것을 확인하였다.

부하테스트는 Cache를 달지 않은 채로, Cache를 탑재한 채로 각각 진행하였으며, NodeJS 서버를 열어 같은 파일을 요청하며 3가지를 모두 비교해보았다. 파일은 대략 10MB의 이미지를 포함한 html파일이다. 요청은 (1/60)초 간격으로 300개의 Thread를 하나씩 생성하여 요청을 주며, 각 Thread마다 50번의 요청을 반복하여 총 15000번의 Requests를 부과한 모습이다.

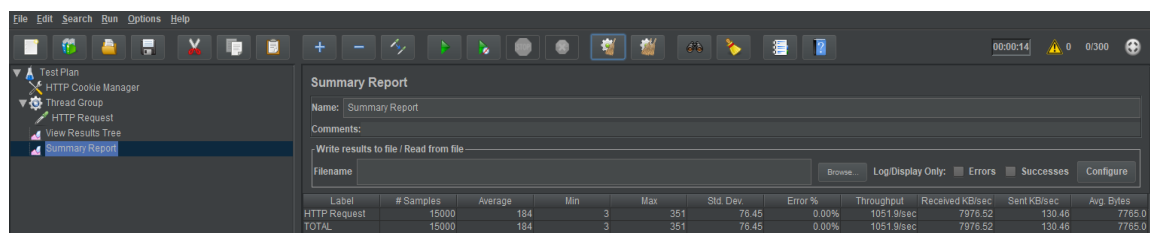


Figure 14 shows the Jmeter Summary Report for a test without cache. The report displays the following data:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K/sec	Sent K/sec	Avg. Bytes
HTTP Request	15000	184	3	351	76.45	0.00%	1051.9/sec	7976.52	130.46	7765.0
TOTAL	15000	184	3	351	76.45	0.00%	1051.9/sec	7976.52	130.46	7765.0

Figure 14. No Cache (15000 requests)

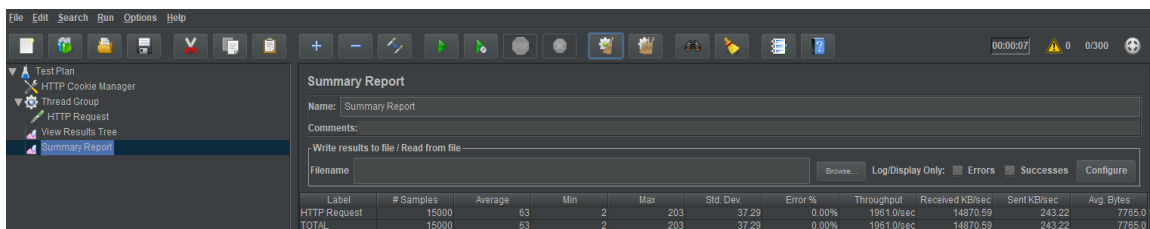


Figure 15 shows the Jmeter Summary Report for a test with cache. The report displays the following data:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received K/sec	Sent K/sec	Avg. Bytes
HTTP Request	15000	63	2	203	37.29	0.00%	1961.0/sec	14870.59	243.22	7765.0
TOTAL	15000	63	2	203	37.29	0.00%	1961.0/sec	14870.59	243.22	7765.0

Figure 15. With Cache (15000 requests)

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	15000	138	1	318	60.52	0.00%	1253.2/sec	9675.87	155.43	7906.0
TOTAL	15000	138	1	318	60.52	0.00%	1253.2/sec	9675.87	155.43	7906.0

Figure 16. NodeJS Test (15000 requests)

다음은 NodeJS와 Cache를 탑재한 자체 서버에 대한 비교이다. 마찬가지로 똑같은 파일을 GET하는 요청으로, 0.1초 간격으로 1000개의 Thread를 하나씩 생성하여 요청을 주며, 각 Thread마다 100번의 요청을 반복하여 총 100000번의 Requests를 부과한 모습이다.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	100000	137	0	1565	237.03	0.00%	738.4/sec	5700.92	91.58	7906.0
TOTAL	100000	137	0	1565	237.03	0.00%	738.4/sec	5700.92	91.58	7906.0

Figure 17. NodeJS Test (100000 requests)

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	100000	5	0	734	20.70	0.00%	998.3/sec	7570.29	123.82	7765.0
TOTAL	100000	5	0	734	20.70	0.00%	998.3/sec	7570.29	123.82	7765.0

Figure 18. Ours with Cache (100000 requests)

B. Result Analysis and Discussion

전반적으로 Cache를 탑재한 경우 자체 서버가 NodeJS보다 빠른 성능을 보였다. 같은 파일에 대한 요청을 반복해서 하다 보니 Cache의 존재 유무가 성능에 큰 영향을 끼쳤다. Cache를 탑재하지 않은 자체 서버의 경우에는 아직 NodeJS보다 떨어지는 성능을 보였다.

10. Division & Assignment of Work

구분	항목	박성우	데이비드	이승현
구현	HTTP Parser 구현			0
구현	Socket Read & Write 구현	0		0
구현	Client Request Interpreter 구현		0	
구현	Exception 처리	0		
구현	Event Queue 구현		0	
구현	Single Thread Event Loop 구현	0		0
구현	Thread Pool 구현	0		0
테스트	Basic Test		0	
구현	Cache 구현	0		0
테스트	Node.js와 성능 비교 테스트		0	
정리	Wrap Up	0	0	0

Figure 19 Division & Assignment

11. Conclusion

본 프로젝트는 기존 Thread model의 웹 서버를 개선하기 위해 개발된 Event model의 웹 서버를 학습하고 직접 구현해보는 데 의의가 있다. 그 과정에서 현업에서 실제로 많이 사용되고 있는 IO multiplexing 기법을 자바의 라이브러리를 활용하여 직접 설계해보고, 대용량의 파일에 대해서도 안정적인 서비스를 할 수 있는 시스템을 구축하였다. 성능을 개선하기 위해 자바의 LinkedHashMap을 이용하여 FIFO Cache와 LRU Cache도 구현하였으며, 실제로 같은 파일에 대한 요청이 계속해서 이루어질 때 뛰어난 성능 향상을 이루었다.

팀원 모두 네트워크에 대한 지식이 전무하였지만 프로젝트를 통해 전반적인 소켓 프로그래밍에 대해 익히는 계기가 되었다. 클라이언트의 요청이 HTTP Protocol에 따라 어떤 식으로 들어오는지, 서버는 그 요청을 어떻게 해석하고 반응하는지에 대해 자세히 배울 수 있었다. 또한, 우리가 구현한 서버의 응답이 웹 브라우저에 직접 뜨는 것을 확인하며 클라이언트에게 응답이 어떠한 식으로 보여지는지도 깨달을 수 있었다.