**<2018년 창의적통합설계2 중간보고서>**

**Event Loop을 이용한**

**WebServer 구현**

**Team 19**

**경영학과 박성우**

**컴퓨터공학부 이승현**

**컴퓨터공학부 데이비드**

**지도교수님 : 신영길 교수님**

Table of Contents

1. Abstract 2

2. Introduction 3

3. Background Study 4

3.1. 관련 접근방법/기술 장단점 분석 4

3.2. 프로젝트 개발환경 5

4. Goal & Requirements 6

4.1.Event Loop의 구현 6

4.2.HTTP Parser 6

4.3. Event Loop과 Thread Pool 간의 통신 6

4.4. Cache의 구현 6

4.5.Event Loop을 사용하는 기존 서비스와의 비교 7

5. Approach 8

5.1.Read Request 8

5.2.Parse Request 8

5.3. Find Resource 9

5.4. Read File 9

5.5.Make Response 9

5.6.Write Response 9

6. Project Architecture 10

6.1. Architecture Diagram 10

6.2. Architecture Description 10

7. Schedule 12

8. Implementation Spec 13

8.1. Input/Output Interface 13

8.2. Inter Module Communication Interface 13

8.3. Modules 13

9. Current Status 15

9.1. Design Summary 15

9.2. Implemented Modules 16

10.Future Work 19

10.1.Additive Implementation 19

10.2.Demo Plan 19

11. Division & Assignment of Work 20

# **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 loop을 통해 요청을 처리한다. 그 single thread는 서비스가 제공되는 동안 infinite loop을 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.

# **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 것으로 전망된다.[[1]](#footnote-1) 또한 시스코의 2015-2020 글로벌 전망 보고서 발표에 따르면 2020년에 세계 인터넷 사용자 수는 41억 명에 달할 것이며, IP 트래픽은 현재보다 3배 이상 증가할 것으로 예상된다[[2]](#footnote-2). 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.

# **Background Study**

## **3.1. 관련 접근방법/기술 장단점 분석**

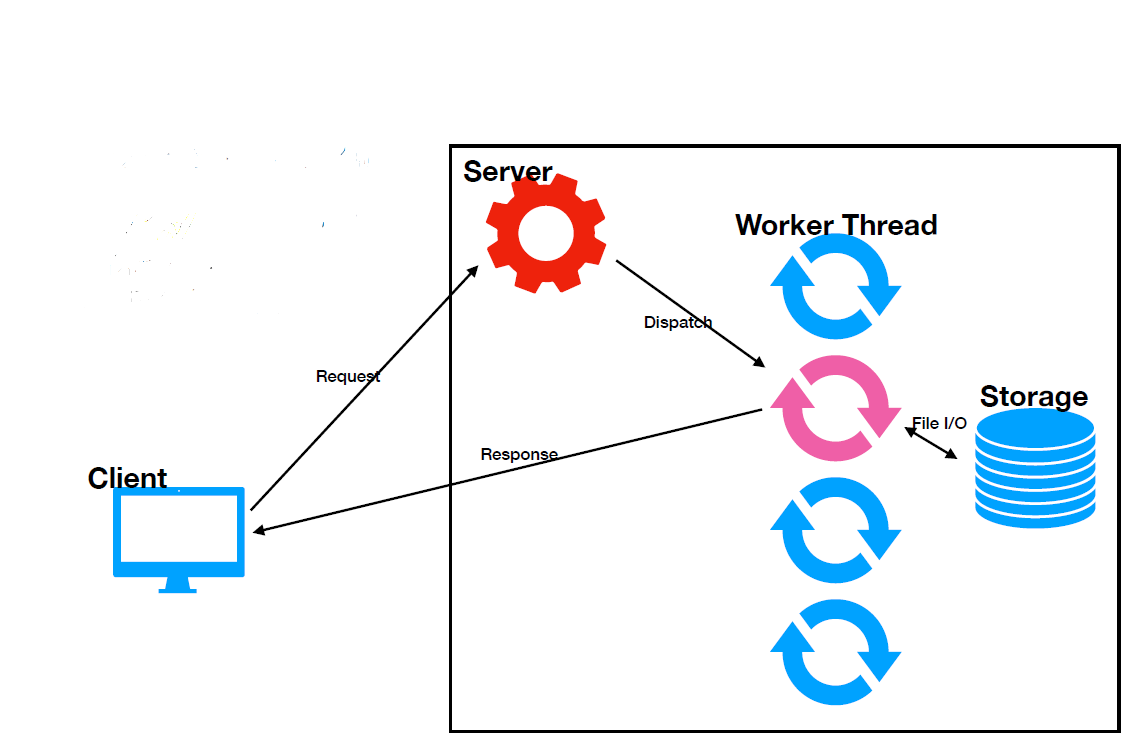


Figure 1. 전통적인 Web Server의 Architecture

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 기반으로서 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.

## **3.2. 프로젝트 개발환경**



Figure 2. 프로젝트 개발 환경

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.

# **Goal & Requirements**

**4.1. 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.

**4.2. 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.

**4.3. 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.

**4.4. 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.

**4.5. 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.

# **Approach**

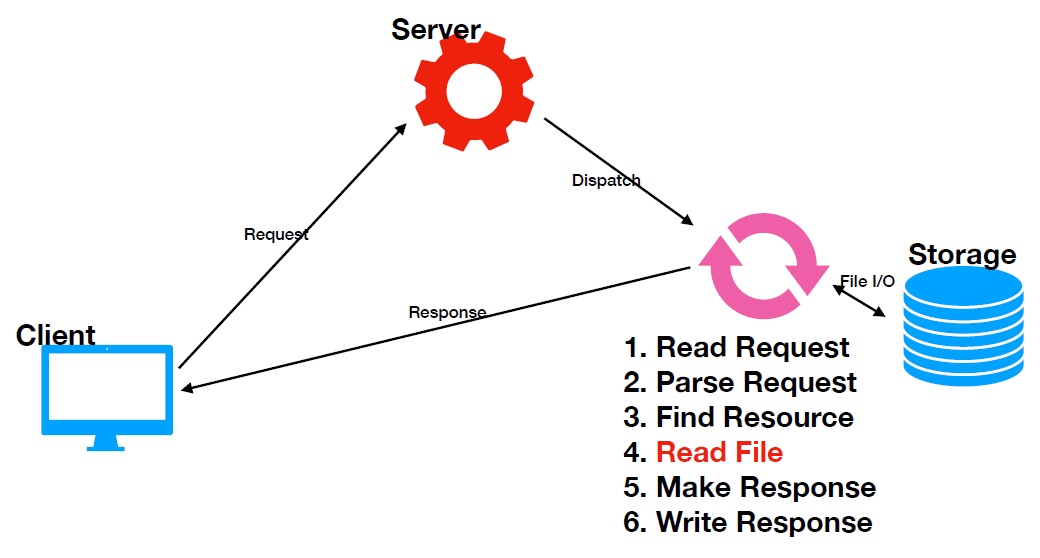


Figure 3. Approach

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.

**5.1. Read Request**

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

**5.2. 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.

**5.3. 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.

**5.4. 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 처리해주도록 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.

**5.5. 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.

**5.6. Write Response**

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

# **Project Architecture**

## **6.1. Architecture Diagram**

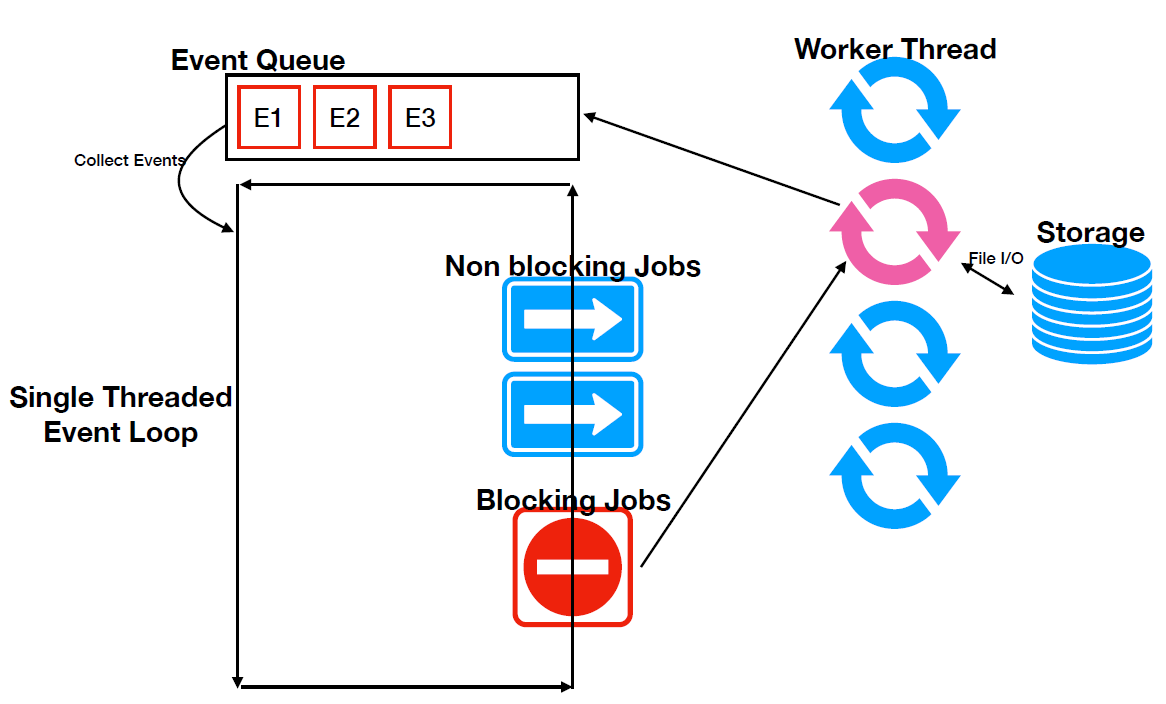


Figure 4. Architecture

## **6.2. 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 세부적으로 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.

# **Schedule**

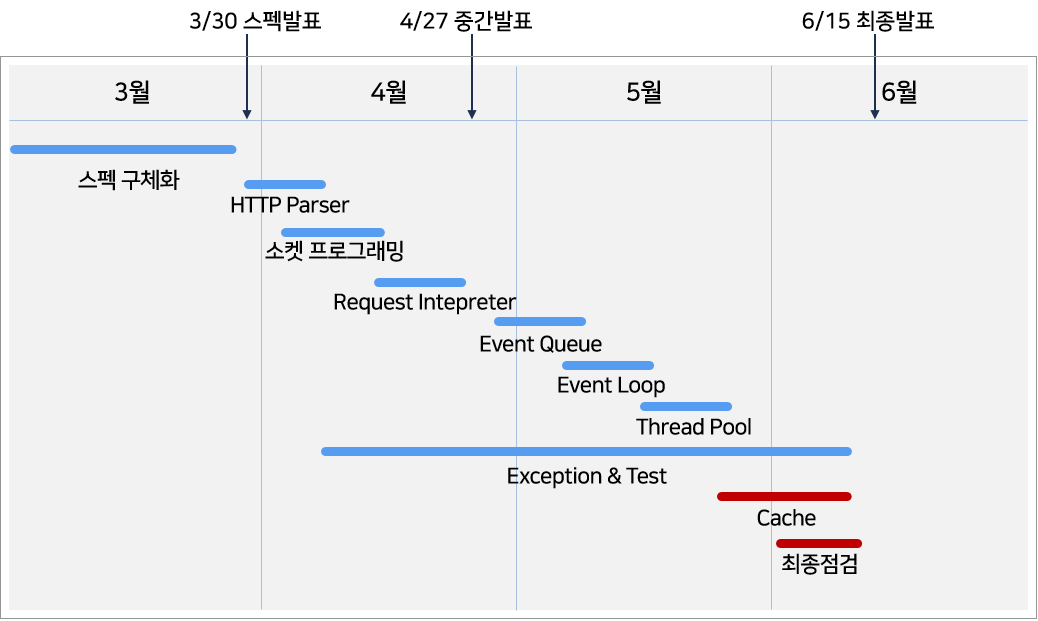


Figure 5. Schedule

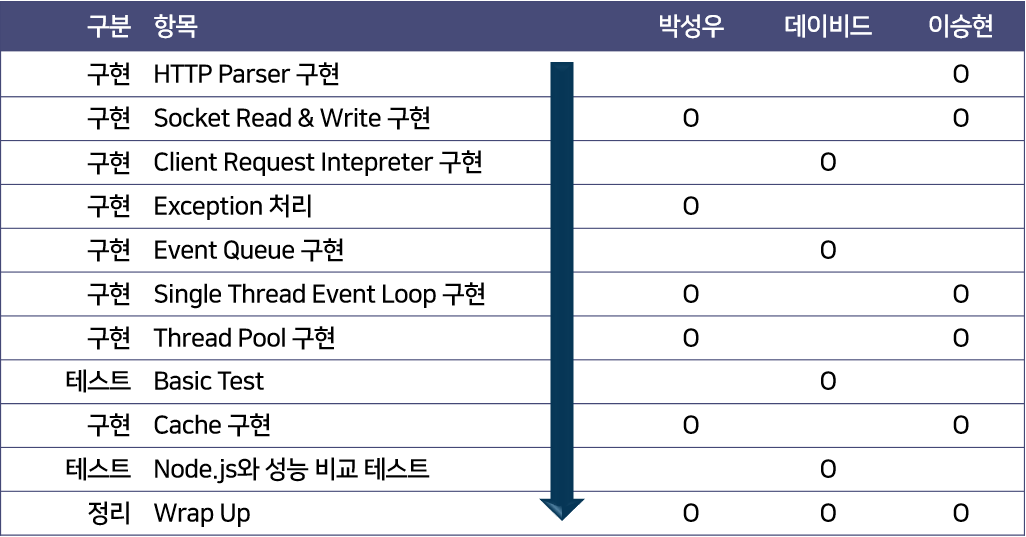


Figure 6. Assignment

# **Implementation Spec**

## **8.1. 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.

## **8.2. Inter Module Communication Interface**

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 ThreadPool에서 일을 하지 않는 thread에게 일을 넘겨준다. 이 과정에서 필요한 파일이 cache에 존재한다면 빠르게 불러오고, 없다면 파일을 저장하고 읽는 작업을 진행한다. ThreadPool에서 I/O 작업이 끝났다면 끝난 작업의 정보를 바탕으로 event를 생성하여 EventQueue에 다시 enqueue한다.

## **8.3. Modules**

**a. Main.java**: 서버 소켓을 열고, 클라이언트의 요청을 받아들인다. 서버를 닫으라는 특정 신호가 있을 때까지 소켓을 닫지 않는다.

**b. EventLoop.java**: Event의 정보를 기반으로 처리하여 응답을 내보내는 logic을 구현한 모듈이다. Event가 blocking job이 필요한 지의 여부도 판단한다.

**c. Event.java**: 클라이언트의 요청에 대해 파서가 해석한 정보를 담고 있는 하나의 Event 객체를 구현하는 모듈이다.

**d. EventQueue.java**: Event들을 보관하는 queue를 구현한 모듈이다. 기본적인 queue와 같이 enqueue, dequeue 등의 기능을 가지고 있다.

**e. ThreadPool.java**: 실제 thread pool의 역할을 하여 event의 blocking job을 전담하는 모듈이다.

**f. HTTPParser.java**: String 형태로 들어온 HTTP request를 바탕으로 method, URI 정보를 추출하고, header를 해석하는 역할을 하는 모듈이다.

**g. Cache.java**: 파일 접근 시 캐시 역할을 하는 모듈이다. 캐시에 파일이 존재하면 바로 정보를 꺼내어 전달하고, 없으면 I/O 작업이 완료된 후 캐시에 저장한다.

# **Current Status**

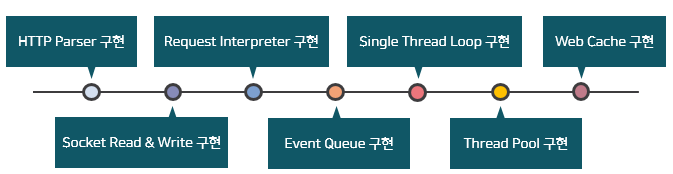


Figure 7. Implementation Flow

**9.1. Design Summary**

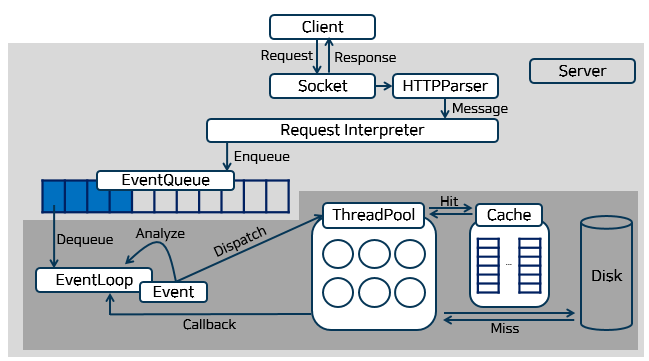


Figure 8. Design Summary

Module 간의 Communication 개요는 Figure.8과 같다. 새로운 클라이언트가 엑세스를 시작하면 module 내의 selector가 socket binding 과정을 통해 socket을 할당하고, request를 HTTPParser를 통해 parsing해서 필요한 정보를 얻는다. 그리고 그 정보를 EventQueue에 enqueue하고, EventLoop이 하나씩 dequeue하면서 blocking job인지 non-blocking job인지 판단해서 worker thread에게 dispatch하거나 자신이 스스로 처리하는 과정을 반복한다. 이때 worker thread는 disk aceess를 하기 전에, cache hit 여부를 먼저 체크하게 되는데, hit이라면 바로 데이터를 해당 event에 wrap해서 event queue에 다시 넣어주게 되고, cache miss라면 직접 disk에 access함으로써 동일한 과정을 반복한다.

**9.2. Implemented Modules**

**a. Socket Binding**

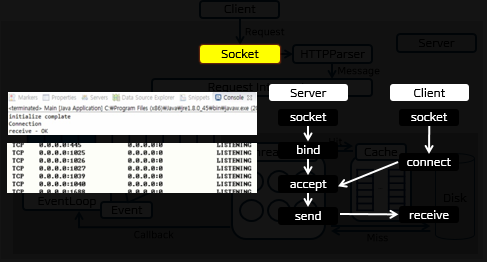


Figure 9. Socket Binding

서버 소켓을 열고, 클라이언트의 요청을 받아들인다. 서버를 닫으라는 특정 신호가 있을 때까지 소켓을 닫지 않는다. Java의 ServerSocket과 Socket, 그리고 java selector가 이미 제공되기 때문에 그대로 import해서 사용하였다.

**b. HTTPParser.java**

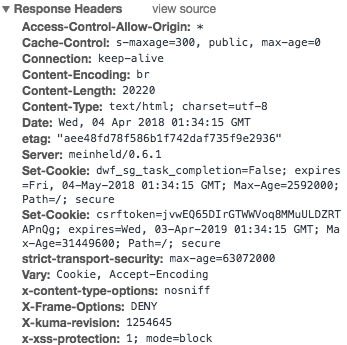
****

Figure 10. Connection Header

Request를 parse하는 parse method와 parsing한 request를 바탕으로 응답을 생성하는 makeResponse method로 크게 나누어진다. Event의 정보를 기반으로 처리하여 응답을 내보내는 logic을 구현한 모듈이다. Event가 blocking job이 필요한 지의 여부도 판단한다. 여기서 type Event는 해당 method가 어떤 method인지, 어떤 URI인지, connection status를 함께 저장하며, 지원하는 response type은 200, 304, 305, 400, 404 등이 있다. 한편, connection header도 parse하는 과정이 포함되어 있는데, 현재의 전송이 완료된 후 네트워크 접속을 유지할지 말지를 제어하게 된다. 만약 전송된 값이 keep-alive면, 연결은 지속되고 끊기지 않으며, 동일한 서버에 대한 후속 요청을 수행할 수 있다.

**c. Request Interpreter**



Figure 11. Job Classification

클라이언트의 요청에 대해 파서가 해석한 정보를 담고 있는 하나의 Event 객체를 구현하는 모듈이다. 모든 Job을 6가지의 배타적인 작업으로 나누며, 이 중에서 5가지는 non-blocking job, 나머지 1개는 직접적으로 file access를 하는 blocking job으로 세분화하였다. 그리고 event queue에 들어있는 하나의 event마다 해당 event가 어떤 status에 있는지 기록하는 역할을 한다. 일이 진행될 때마다 status는 바뀌게 되고, 마지막 job인 socket write가 완료되면, 클라이언트에게 완전한 응답을 return할 수 있게 된다.

**d. EventQueue.java**

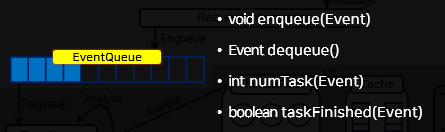


Figure 12. Event Queue

Event들을 보관하는 queue를 구현한 모듈이다. 기본적인 queue와 같이 enqueue, dequeue 등의 기능을 가지고 있다. 그 외에 numTask method는 event queue에 대기하고 있는 event의 개수를 return하는 method이며, event queue안에 들어있는 특정 event가 완료된 event인지를 확인할 수 있는 method를 구현하였다.

# **Future Work**

**10.1. Additive Implementation**

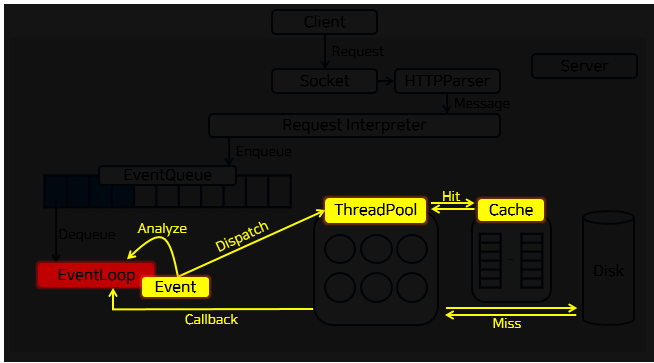


Figure 13. Unimplemented Modules

이미 구현한 socket read, write 과정, HTTP Parser, EventQueue를 제외한 나머지 module에 대한 구현이 추가로 필요하다.

**10.2. Demo Plan**

Node.js와의 비교는 여러 client의 동일한 load 요청에 대한 응답 시간을 기준으로 한다. 브라우저는 Chrome을 대상으로 하며, Apache JMeter 등의 테스트 도구를 사용한다. File access는 static file system에 대해서만 일어난다고 가정한다. 그리고 cache replacement policy는 LRU, FIFO에 대해 2회 반복 시행한다. 한편, 1MB 이상의 파일 접근의 경우 DMA을 통해 disk에서 곧바로 transfer가 일어날 수 있도록 구현하며, 1MB 이하 파일의 경우 cache에 보관하도록 한다. Node.js와 우리가 구현한 서버에 대해 동일한 요청을 많이 보내서 응답 시간의 평균을 구할 것이다. 이때 load는 클라이언트의 수로 변화시킬 것이며, 클라이언트의 수는 1K, 5K, 10K, 50K, 100K, 500K, 1M, 5M, 10M, 50M에 대해 반복 시행한다. 시간에 따른 응답 시간의 평균을 비교 인덱스로 삼아서 Node.js와 비교하여 80% 이상의 성능을 보인다는 것을 증명하는 식으로 demo를 진행할 것이다. 테스트 결과는 아래 그림에서 보듯이 JMeter가 그래프 혹은 표의 형태로 제공한다.

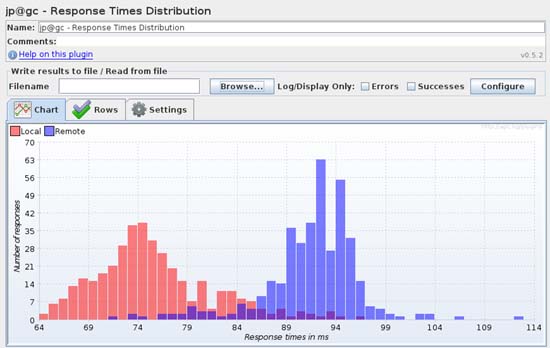


Figure 14. JMeter Example

1. **Division & Assignment of Work**

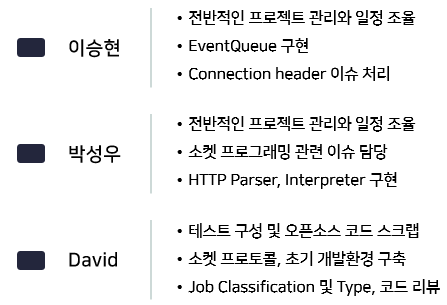


Figure 15. Division of Work

1. <Internet Users and Penetration in Urban vs. Rural India>, eMarketer, 2018.02.20 [↑](#footnote-ref-1)
2. <Cisco Global Cloud Index 2015-2020>, GCI Forecast, 2016.06.13 [↑](#footnote-ref-2)