

System Programming Project 2

담당 교수 : 김영재

이름 : 송재현

학번 : 20190554

1. 개발 목표

- 해당 프로젝트에서 구현할 내용을 간략히 서술.
- (주식 서버를 만드는 전체적인 개요에 대해서 작성하면 됨.)

기존의 Iterative Server와 다르게 여러 명의 client가 동시에 접속하여 Concurrent하게 서비스할 수 있는 Concurrent 주식 서버를 구현한다. 주식서버에는 txt파일로 저장되어있는 주식 정보를 binary search tree형태로 load하여 구매, 판매, 주식 목록 조회, exit과 같은 요청을 할 수 있다.

2. 개발 범위 및 내용

A. 개발 범위

- 아래 항목을 구현했을 때의 결과를 간략히 서술

1. Task 1: Event-driven Approach
2. Task 2: Thread-based Approach
3. Task 3: Performance Evaluation

1. Select 함수를 이용해 어떤 descriptor에 pending input이 있는지 확인한다. 이 때 도착한 pending input이 있으면 이를 event라고 한다. 이를 이용하여 여러 명의 client가 주식 서버에 connection을 요청할 수 있게 되고 주식 서버에서 구매, 판매, exit, 목록 조회를 할 수 있다. Select를 이용하여 pending input이 있는 file descriptor들만 작업을 수행해주어 여러 client를 concurrent하게 서비스 할 수 있도록 한다.

2. Thread를 이용하여 Event-based Server처럼 여러 명의 client가 주식 서버에 connection을 요청할 수 있게 된다. thread를 미리 여러개 생성해두고 client로부터 요청이 오면 connection 요청을 수락해 미리 생성한 thread를 이용하여 client의 작업을 처리해주어 여러 client를 concurrent하게 서비스 할 수 있도록 한다.

3. Event-driven 방식과 Thread-based 방식의 성능을 평가하고 분석한다. 둘의 동시 처리율을 client의 개수에 따라 분석하고 client의 buy, sell, show 요청 빈도에 따라서 동시 처리율이 어떻게 변하는지 분석한다.

B. 개발 내용

- 아래 항목의 내용만 서술

- (기타 내용은 서술하지 않아도 됨. 코드 복사 붙여 넣기 금지)

- Task1 (Event-driven Approach with select())

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

Select 함수를 사용해서 I/O Multiplexing을 할 수 있다. listenfd를 read_set에 넣어서 select 함수를 통해 Event가 있는지 확인해주고 반복문 안에서 계속해서 이를 확인하도록 한다. 이 때 listenfd에 해당하는 부분이 set이 되면 Client로부터의 connection 요청으로 간주하고 이의 return 값 역시 read_set에 넣어주어 확인하도록 한다. 이를 통해 Client로부터의 요청이 있는지 확인할 수 있다. Exit을 요청할 경우 conned를 close해주어 client와의 연결을 끊어주었다.

✓ epoll과의 차이점 서술

epoll 함수는 전체 fd에 대해서 반복문을 사용하지 않고 kernel에게 정보를 요청하는 함수를 호출할 시에 감시하는 대상에 대한 정보를 넘기지 않는다. 따라서 select의 단점을 보완한 함수임을 알 수 있다. 정보를 넘기지 않는 방법을 사용하기 때문에 fd의 array를 운영체제가 이를 관리한다.

- Task2 (Thread-based Approach with pthread)

- ✓ Master Thread의 Connection 관리
- ✓ Worker Thread Pool 관리하는 부분에 대해 서술

Master thread는 처음에 worker thread를 미리 많이 만들어둔다. 이후 Client로부터 Connection 요청이 오면 이를 accept하고 connfd를 buffer에 넣는다. 이 buffer는 synchronization을 위해 master thread가 producer를 worker thread가 consumer의 역할을 하도록 한다.

Client로부터 connection 요청이 오면 master thread에서 이를 accept하고 connfd를 buffer에 넣어주는데 이 때 미리 만들어둔 suspend 상태에 있는 worker thread 중 하나가 깨어나 client에게 connfd를 통해 서비스를 제공할 수 있도록 한다. 이 때 각 worker thread는 thread 함수에 따라서 동작하는데 본 프로젝트에서는 thread를 detach로 하여 kernal에 의해 자동적으로 reaping될 수 있도록 하였다. 또한 writer reader problem에 대해서 buy와 sell은 writer, show는 reader로 하여 First reader writers synchronization으로 reader writer problem을 해결하였다.

- Task3 (Performance Evaluation)

- ✓ 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술
- ✓ Configuration 변화에 따른 예상 결과 서술

client 수가 많아짐에 따라 thread-based server가 Event-based server에 비해 더 효율적으로 나타날 것으로 보인다. Event-based server는 multicore의 장점을 살릴 수 없고 fine grained concurrency가 불가능하기 때문에 대체적으로 pthread가 더 좋은 성능을 낼 것으로 예측된다.

client의 요청에 따라서는 thread 방식은 read에 해당하는 buy와 sell을 할 때 semaphore에 의해서 하나의 client에만 접근하는 synchronization overhead로 인해서 event based 방식이 더 좋은 효율을 낼 것으로 보이고 show를 할 때는 write에 해당하기 때문에 First reader writers synchronization으로 문제를 해결한 본 프로젝트에서는 thread 방식이 event based 방식에 비해 더 좋은 효율을 낼 것으로 보인다.

본 실험은 5개의 주식 종목을 대상으로 client의 수를 늘려가는 방식으로 진행한다. client당 30개의 request를 요청하도록 하였고 client 수를 각각 1, 5, 10, 20, 40일 때를 바탕으로 진행한다.

두 번째로 client가 buy, sell, show를 섞어서 요청하는 경우를 실험한다. 세 번째로 client가 buy, sell만 하는 경우를 실험하고 마지막으로 client가 show만 하는 경우를 실험한다.

C. 개발 방법

- B.의 개발 내용을 구현하기 위해 어느 소스코드에 어떤 요소를 추가 또는 수정할 것인지 설명. (함수, 구조체 등의 구현이나 수정을 서술)

Event 기반과 thread 기반 모두 bst를 기반으로 동작하는 것은 공통적이기에 bst를 stock의 id를 key로 하여 생성한다.

Event 기반 서버의 경우 Select 함수를 통해 read_set을 감시하고 이를 통해 listenfd와 각 client의 요청을 accept한다. Listenfd가 select이후 set되어 있으면 add client를 통해 descriptor set에 descriptor를 추가하고 check_clients를 통해 descriptor가 ready되어 있으면 그에 해당하는 요청사항에 따라서 해당 작업을 수행한다.

Thread 기반 서버의 경우 stock.txt를 이용해 bst를 구성할 때 mutex와 writer에 대한 semaphore를 초기화 시켜주어 node에 추가시킨다. 또한 서버가 accept한 connfd를 저장하는 buffer 구조체를 만든 후 이를 초기화 시켜준 뒤 connfd를 buffer에 삽입한다. Thread pooling을 통해서 코드를 구현하였으므로 미리 thread를 많이 생성시켜두고 각 thread들은 detach 모드로 설정한 후에 connfd를 제거, 이후 client의 요청을 처리하는 코드를 작성한다. 이 때 조회, 판매, 구매 시에 First-reader-writer 방식으로 synchronization 코드를 추가해준다. 이외에도 파일 입출력시에 file_mutex를 통해 semaphore를 추가해준다.

3. 구현 결과

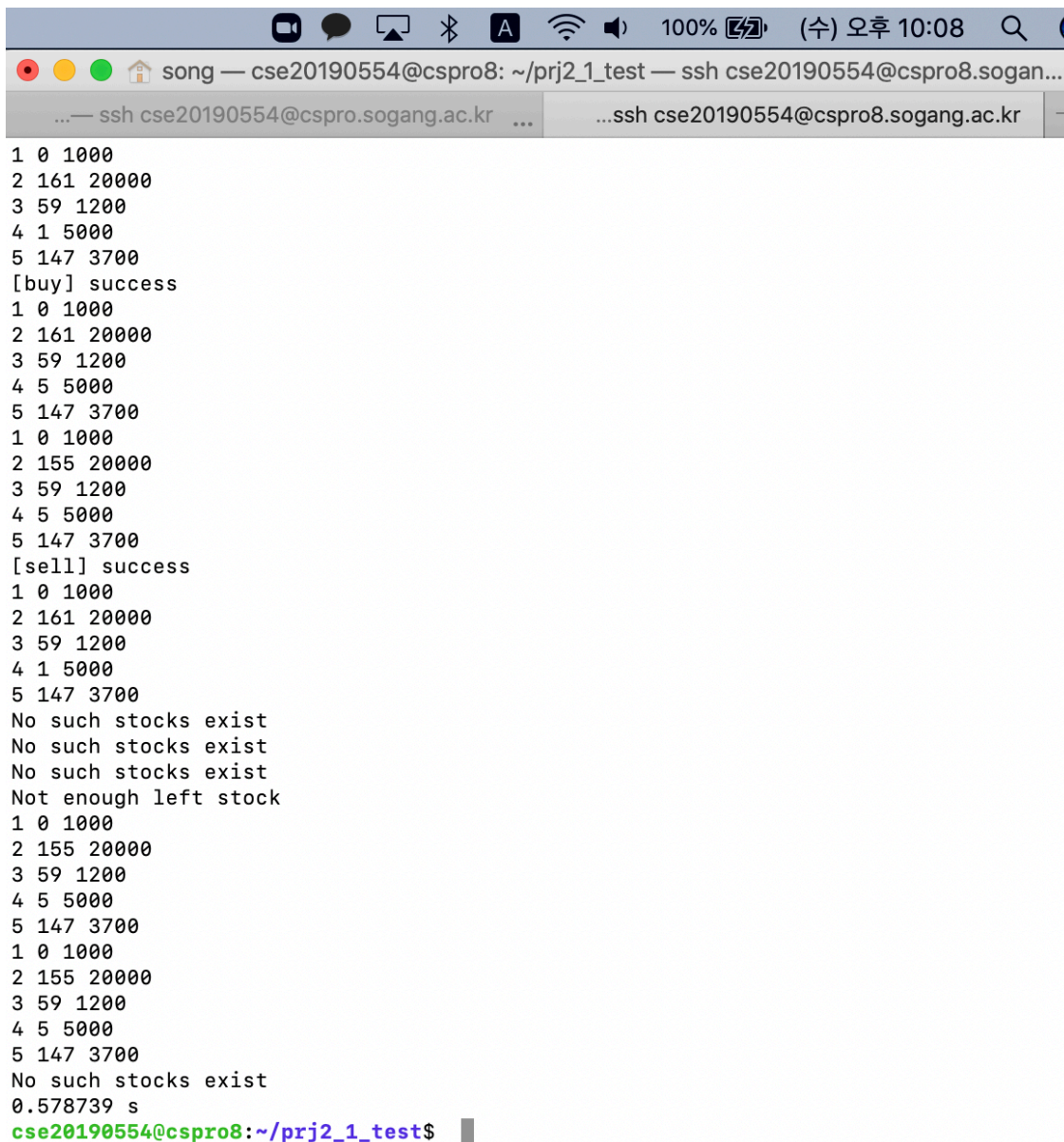
- 2번의 구현 결과를 간략하게 작성
- 미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가

구현결과 Event 기반 서버와 thread 기반 서버 모두 concurrent하게 작동하는 것을 확인할 수 있었다. Event 기반 서버의 경우 감시하고자 하는 Read_set을 유지하기 위해 ready_set을 select함수에 인자로 포함시키고 이를 통해 pending input이 있는지 확인하였다. listenfd에 pending input이 있으면 connection을 accept하고 client를 추가하였고 마지막으로 check_client를 통해 client의 요청을 수행할 수 있었다.

Thread 기반 서버 역시 concurrent하게 작동하는 것을 확인할 수 있었다. Thread pooling 방식을 이용하였으므로 미리 SBUFSIZE만큼 thread를 생성해두었고 thread기반 서버에서는 connection 요청에 따라 buffer인 sbuf에 connfd를 추가해주었다. thread는 detach방식으로 kernel에서 reaping하도록 하였고 buffer에서 connfd를 제거 한후 client의 요청을 service해주는 방식으로 작동하였다.

4. 성능 평가 결과 (Task 3)

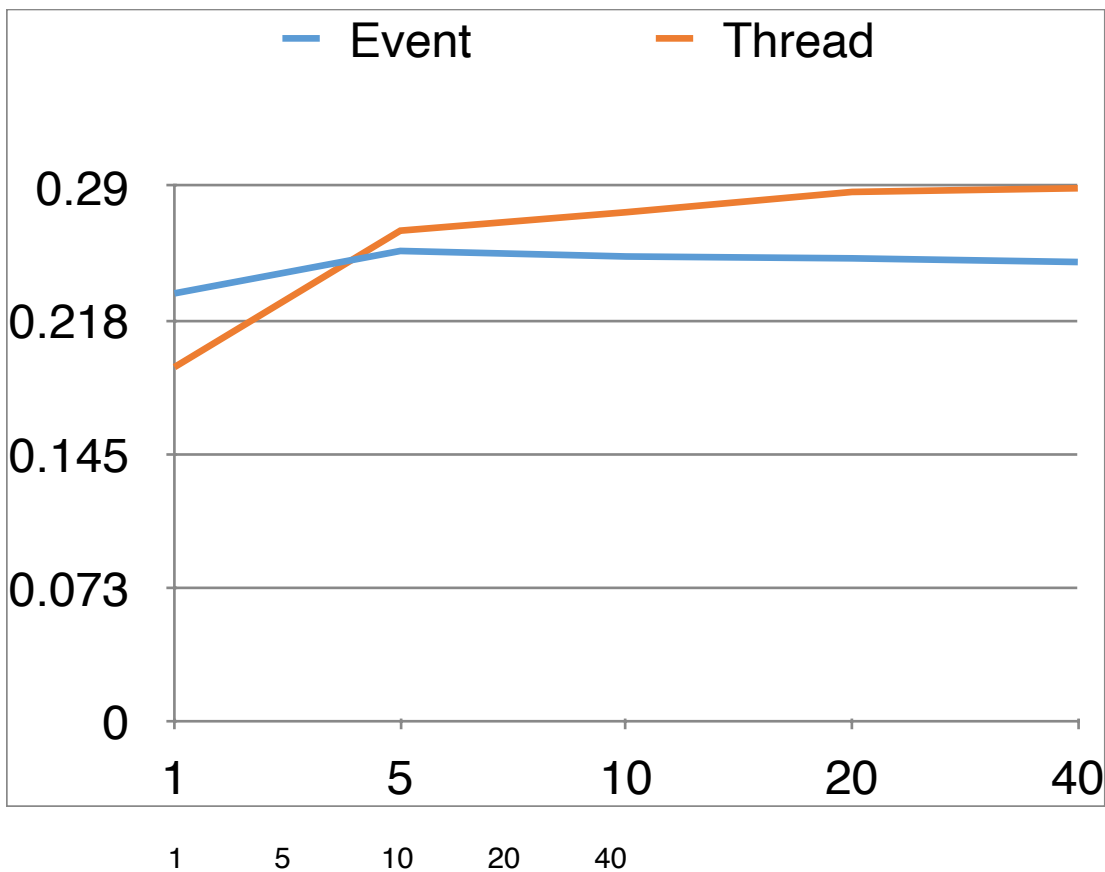
- 강의자료 슬라이드의 내용 참고하여 작성 (측정 시점, 출력 결과 값 캡처 포함)



```
1 0 1000
2 161 20000
3 59 1200
4 1 5000
5 147 3700
[buy] success
1 0 1000
2 161 20000
3 59 1200
4 5 5000
5 147 3700
[sell] success
1 0 1000
2 161 20000
3 59 1200
4 1 5000
5 147 3700
No such stocks exist
No such stocks exist
No such stocks exist
Not enough left stock
1 0 1000
2 155 20000
3 59 1200
4 5 5000
5 147 3700
1 0 1000
2 155 20000
3 59 1200
4 5 5000
5 147 3700
No such stocks exist
0.578739 s
cse20190554@cspro8:~/prj2_1_test$
```

앞에서의 예측은 client 수가 많아짐에 따라 일반적으로는 thread는 event에 비해 더 효율적일 것이고 show만 처리할 때에는 thread 기반이 더 좋은 성능을 buy, sell만 할 때에는 event 기반이 더 효율적일 것이라고 예상했다. gettimeofday()함수를 사용하여 시간을 측정하였고 usleep은 효율성을 떨어뜨리므로 usleep을 제외하고 측정하였다. 위에서와 같이 결과값이 나오는 것을 확인할 수 있는데 모든 결과값을 캡처하기 어려워 그래프로 표현하였다.

동시처리율은 (client 처리 요청 개수 / 시간)으로 계산하였다. 다음은 x축이 client의 수이고 y축이 동시처리율을 나타내는 그래프이다.

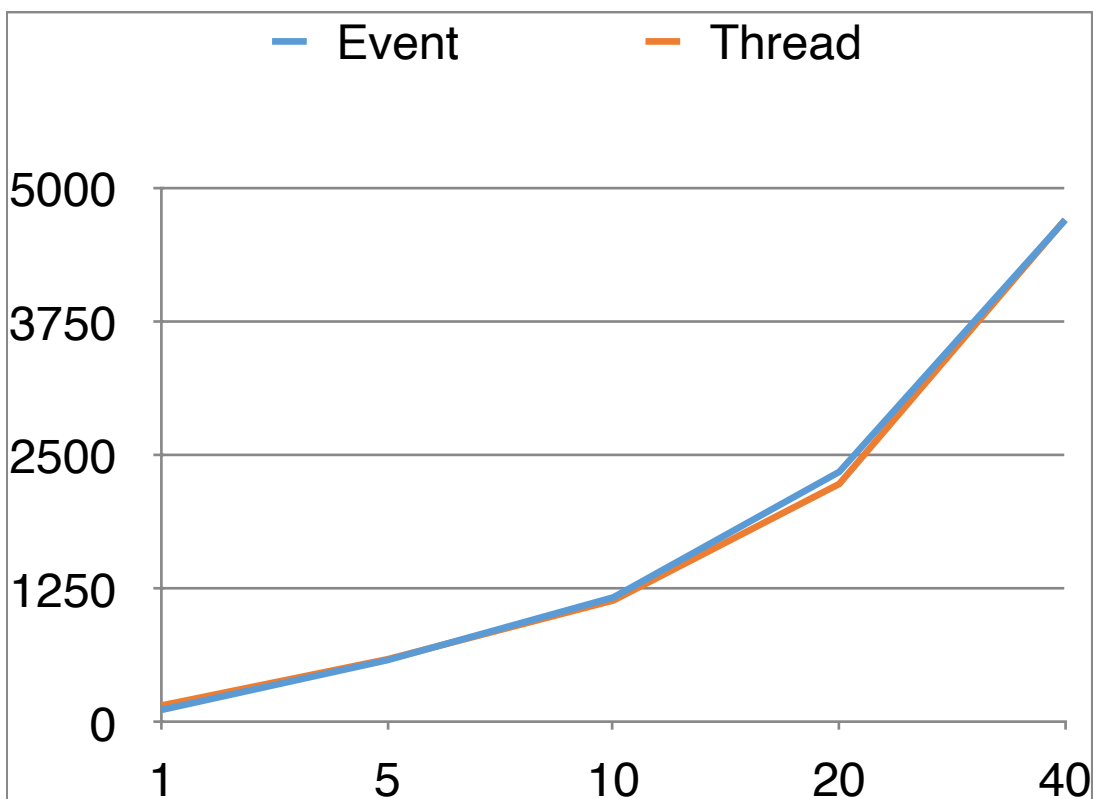


Event 0.232 0.255 0.252 0.251 0.249

Thread 0.192 0.266 0.276 0.287 0.289

위의 그래프에서 확인할 수 있듯이 client의 수가 적을 때에는 Event based가 더 효율적인 것으로 보이나 client의 수가 늘어날 수록 예상과 같이 Thread-based가 대체적으로 더 효율적인 것을 알 수 있다.

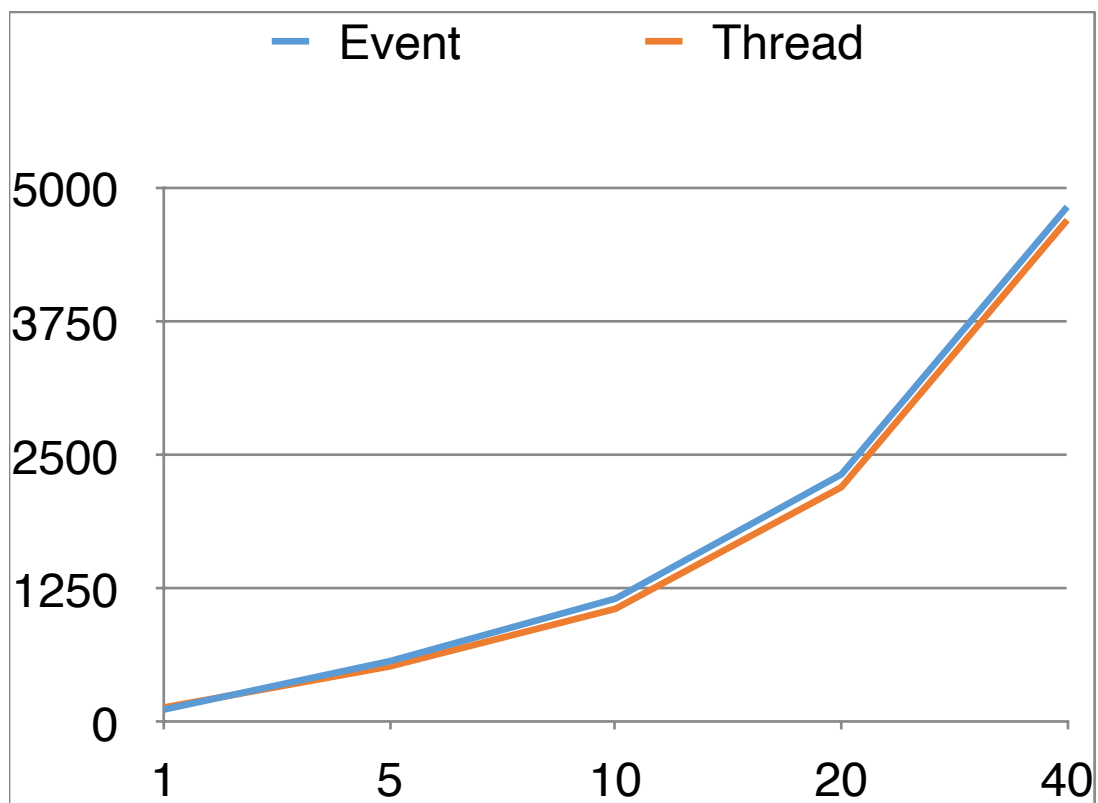
Thread-based의 경우 fine graine하기 때문에 Event based에 비해 더 효율적으로 결과가 나왔다고 생각할 수 있다. Event based의 경우 여러 개의 thread를 이용해 처리하는 thread-based와는 달리 순차적으로 처리하기 때문에 동시처리율이 client 수가 늘어날 수록 비슷하게 유지되는 경향이 있다.



	1	5	10	20	40
Event	119	584	1171	2344	4698
Thread	160	592	1142	2231	4702

(단위 ms)

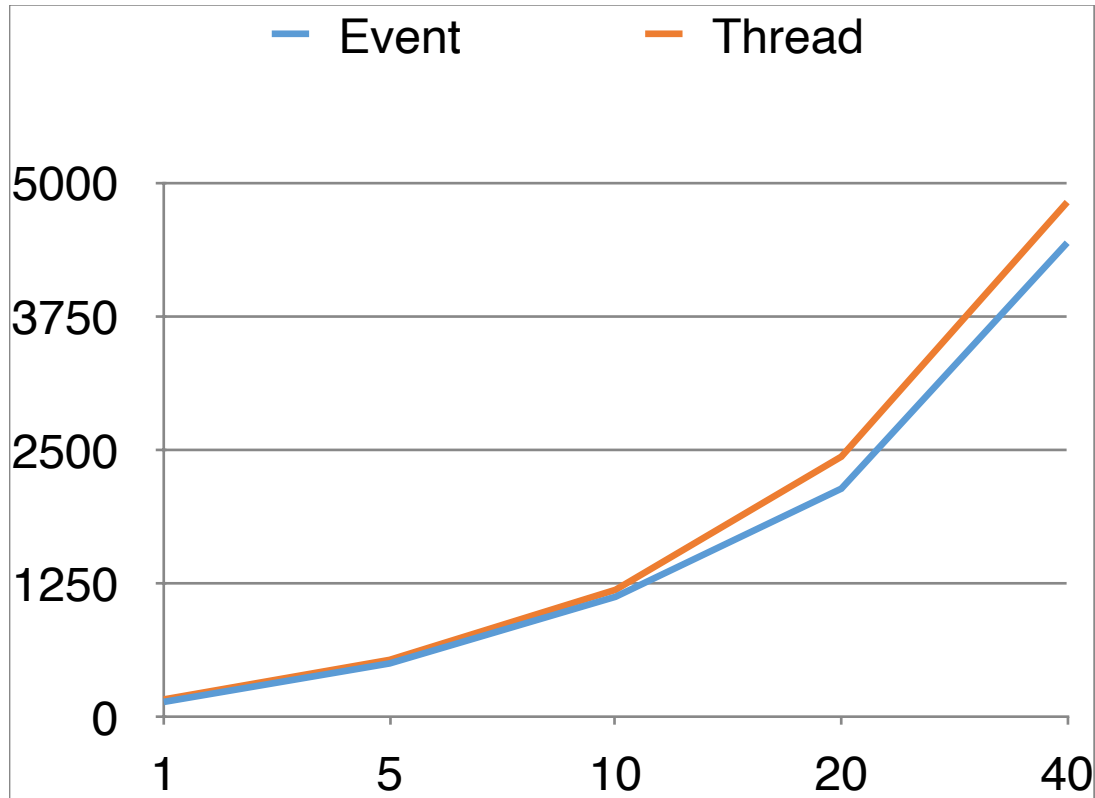
위의 그래프는 client가 random하게 buy sell show를 request했을 때의 결과이다. 실험 전 예상은 대체적으로 thread가 효율적이라는 것인데 그 예상에 걸맞는 결과를 보여주고 있다. 하지만 client가 적을 때는 thread based가 더 비효율적인 경우도 보이는데 그 이유는 synchronization 으로 인한 overhead가 발생하여 적은 수의 client가 요청을 할 때는 Event based가 더 효율적일 수도 있다는 결과를 보여준다. 또한 random하게 buy, sell, show를 정하기 때문에 비교적 show가 많은 편이 유리한 Thread based의 경우 show 명령이 적게나온다면 실험을 적게 했기 때문에 더 비효율적으로 보일 수 있다.



	1	5	10	20	40
Event	122	574	1159	2321	4822
Thread	141	525	1064	2201	4699

위의 그래프는 client가 random하게 명령을 요청하지 않고 show만을 요청했을 때의 그래프이다. 위에서의 그래프보다 대체적으로 Thread 기반 서버가 더 효율적인 것이 명확하게 드러남을 알 수 있다. buy와 sell이 없기 때문에 synchronization overhead가 발생하지않아 이와 같은 결과가 나타남을 알 수 있다.

First reader writer 로 코딩하였기 때문에 여러 client가 show를 요청하고 이를 처리할 수 있기 때문에 위와 같은 차이가 났다고 생각할 수 있다.



	1	5	10	20	40
Event	139	499	1122	2133	4432
Thread	162	532	1187	2432	4812

위의 그래프는 client가 buy와 sell 명령만을 요청하였을 경우의 그래프이다. 위에서의 그래프들과는 달리 Event based 서버가 더 효율적임을 알 수 있다. 이는 예상했던 결과와 같은 결과인데 Thread-based의 경우 synchronization overhead에 의해 show는 빠르게 처리할 수 있지만 buy와 sell은 한 번에 하나의 thread만 접근할 수 있기 때문에 Event based에 비해 얻는 이점이 없다고 볼 수 있다. 따라서 위와 같은 결과가 나온 것으로 보인다.

대체적으로 실험 결과 예상했던 것과 비슷한 결과가 나온 것을 알 수 있다. 실험 결과가 수업시간에 배운 내용들과 일치하는 것을 확인할 수 있었다. 네트워크 환경에 따라 짧은 시간동안 진행되기 때문에 결과값이 크게 달라질 수 있다는 점에서 실험 결과가 원치 않게 나올 수 있었으나 request를 30개로 요청하여 이러한 불확실성을 줄인 점, 여러 번 실행하여 평균값으로 결과를 추출한 점에서 실험 결과가 이상적인 결과와 유사하게 나온 것으로 생각된다.