System Programming Project 2

담당 교수 : 김영재 교수

이름 : 강 민 석

학번 : 20181589

1. 개발 목표

해당 프로젝트는 여러 client들의 동시 접속 및 서비스를 위한 concurrent stock server를 구축하는 것이 목표이다. 주식 서버는 주식 정보를 저장하고 있고, 여러 client들과 통신하여 주식 정보를 보여주고, 판매, 구매의 동작을 수행한다. 그리고 각 주식 클라이언트들은 서버에 주식 사기, 팔기, 가격과 재고 조회 등의 요청을 한다.

주식 서버는 event-driven 방식과 thread-based 방식으로 각각 구축한다. 각 방식으로 구현하기 전에, binary tree data structure을 이용해 주식 data를 구현한다. 주식 data의 관리는 stock.txt 파일로서 저장한다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   - 각각의 client는 fd를 trigger하고, server는 fd들을 monitoring하던 select에 의해서 동작을 시작한다.

   - server의 시작과 함께 stock.txt의 내용을 읽어 메모리에 적재한 후, binary tree를 구현하여 처리한다.

   - server의 종료 시에 업데이트 된 변경사항을 stock.txt에 정해진 형식에 따라 저장한다.

2. Task 2: Thread-based Approach

   - 각각의 client는 fd를 trigger하고, server는 worker thread pool을 생성한다.

   - master thread는 fd를 buffer에 insert하고, worker thread는 buffer에서 fd를 가져와 buffer에서 해당 fd를 지운 후에, fd에 해당하는 client를 service한다.

   - server의 시작과 함께 stock.txt의 내용을 읽어 메모리에 적재한 후, binary tree를 구현하여 처리한다.

   - server의 종료 시에 업데이트 된 변경사항을 stock.txt에 정해진 형식에 따라 저장한다.

3. Task 3: Performance Evaluation

- <sys/time.h> 라이브러리에 있는 gettimeofday 함수를 사용하여, 두 가지 동작

방식(event-driven, thread)의 elapse time을 측정하여 분석한다.

- 확장성에 대한 분석과, 워크로드에 따른 분석 두가지를 수행한다.

- 확장성은 각 방법에 대한 client command 수의 변화에 따른 초당 동시 처리율의 변화를 분석한다.

- 워크로드에 따른 분석은 1) 모든 client가 show만 요청하는 경우, 2) 모든 client가 buy 또는 sell을 요청하는 경우, 3) 세가지를 모두 섞어 random하게 요청하는 경우의 총 세가지로 나누어 분석한다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

- 서버는 active connections의 집합을 유지한다. 즉, connfd를 array 형태로 저장한다. 그 후, connfd나 listenfd 중 pending input이 있는 descriptor가 있나 select를 이용해 살펴보고, 있다면 상응하는 action을 취한다. 이를 반복한다.

- 먼저, pool에 대한 struct를 정의하고, init_pool, add_client, check_clients 함수를 정의한다. main 함수에서는 init_pool을 통해 pool을 초기화해준다. 이후 while문 내부에서 pool.ready_set = pool.read_set을 해주고, select 함수를 통해서 pool.ready_set에 pending input을 저장하고, pool.nready 또한 설정한다. 그리고 pending input이 있는(FD_ISSET이 set되어 있는) connfd에 대하여 client와 server를 연결하고 add_client로 client를 추가한다. 이후 check_clients 함수를 통해서 주식서버 기능을 수행한다.

✓ epoll과의 차이점 서술

- select 함수는 검사 가능한 fd의 최대 개수가 1024개로 제한된다. 그리고 I/O할 준비가 된 fd들에 대해서만 기능을 수행하기 위해서, 모든 fd를 순회하며 FD_ISSET으로 check를 해야 하므로 비효율적이다. 관리하는 fd set 전체를 전달하는 것이 아니라, 실제로 pending된 fd의 목록만 전달하면 더 빠를 것이다.

- 이러한 단점을 epoll 함수가 보완한다. epoll은 커널에 관찰대상에 대한 정보를 한번만 전달하고, 관찰대상에 변경이 있을 때만 변경사항을 알려준다. epoll_wait()

함수는 이벤트가 발생한 fd와 이벤트의 종류를 return해준다. 이 덕분에 모든 fd에 대해서 순회하면서 체크할 필요가 없다. 이벤트가 있는 fd들이 배열에 담겨오고, 그 개수를 알 수 있으므로 꼭 필요한 event만 순회하면서 처리할 수 있다.

- **Task2 (Thread-based Approach with pthread)**

  ✓ Master Thread의 Connection 관리

  - sbuf_t에 대한 structure를 정의한다. thread, sbuf_init, sbuf_deinit, sbuf_insert, sbuf_remove 함수를 작성한다. 그 후에 main에서 while문 전에 sbuf_init을 통해 sbuf를 초기화한다. 이후, 지정한 thread의 개수만큼 Pthread_create를 통해 worker thread pool을 만들어준다. master thread는 listenfd를 통해 들어온 요청 connfd를 sbuf_insert를 통해 buffer에 insert하고, 이 과정을 반복한다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  - worker thread는 buffer에서 fd를 가져와 buffer에서 해당 fd를 지운 후에, fd에 해당하는 client를 service한다.

  - worker thread 함수 내에서는 echo_cnt 함수를 호출함으로써, 주식서버에 필요한 기능들을 수행한다. 처리가 끝나면, connfd를 close해준다.

- **Task3 (Performance Evaluation)**

  ✓ 얻고자 하는 metric 정의, 그렇게 정한 이유, 측정 방법 서술

  - Performance Evaluation은 총 두가지를 수행할 예정이다. 확장성에 대한 분석과, 워크로드에 따른 분석 두가지를 수행한다.

  1) 확장성에 대한 분석: <sys/time.h> 라이브러리에 있는 gettimeofday 함수를 사용하여, 두 가지 동작 방식(event-driven, thread)의 elapse time을 측정하여 분석한다. 확장성은 각 방법에 대한 client command 수의 변화에 따른 초당 동시처리율의 변화를 분석한다. (즉, 얻고자 하는 metric은 1초 동안 동시에 처리하는 command의 수이다.) listenfd를 통해 요청을 듣는 시점부터, 마지막 command의 처리가 끝나는 시점까지 걸린 시간을 gettimeofday 함수를 이용해 측정할 것이다. 이후, 총 소요된 시간을 command 수로 나누면, 1초당 동시에 처리한 command의 수가 나올 것이고, 원하는 metric을 얻을 수 있다.

  2) 워크로드에 따른 분석: (1)모든 client가 show만 요청하는 경우, (2)모든 client가 buy 또는 sell을 요청하는 경우, (3)세가지를 모두 섞어 random하게 요청하는

경우의 총 세가지로 나누어 분석할 것이다. 이 분석에서 얻고자 하는 metric은 역시나 3가지 경우 각각의 동시처리율이고, 이를 통해 어떤 워크로드가 가장 높은 동시처리율을 보이는지(가장 빠른지) 측정할 수 있다.

✓ Configuration 변화에 따른 예상 결과 서술

1) 확장성에 대한 분석: event-based 의 가장 큰 단점 중 하나는 multi-core 의 혜택을 받지 못한다는 것이다. Process 가 하나이기 때문에 single CPU 만이 필요하기 때문이다. 반면 thread-based 는 thread 끼리 context switching 이 일어나기 때문에 multi-core 의 효과를 이용할 수 있다. 본인의 컴퓨터는 4 개의 core CPU 이므로, thread-based 가 event-based 보다 초당 동시처리율이 높을 것으로 예상된다.

2) 워크로드에 대한 분석: 모든 client 가 show 를 요청하는 경우에, 요청이 있을 때마다 binary tree 를 순회하며 모든 node 의 data 를 출력해야 하므로, buy+sell 을 요청하는 경우보다 느릴 것으로 예상된다.

C. 개발 방법

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

- 서버는 active connections의 집합을 유지한다. 즉, connfd를 array 형태로 저장한다. 그 후, connfd나 listenfd 중 pending input이 있는 descriptor가 있나 select를 이용해 살펴보고, 있다면 상응하는 action을 취한다. 이를 반복한다. 먼저, pool에 대한 struct를 정의하고, init_pool, add_client, check_clients 함수를 정의한다. binary tree data structure와 관련된 insert, delTree, search, print_preorder 함수 또한 정의한다.

- main함수에서는 stock.txt로부터 데이터를 읽어 온 후, binary tree를 관리하는 함수인 insert()를 통해 데이터를 binary tree 형태로 만들어 준다. 이후, main 함수는 init_pool을 이용해 pool을 초기화해 주고, select 함수를 통해서 pool.ready_set에 pending input을 저장한다. 그리고 pending input이 있는 connfd에 대하여 client와 server를 연결하고 add_client로 client를 추가한다. 이후 check_clients 함수를 통해서 주식서버 기능을 수행한다.

- check_clients 함수 내부에서는 client에서 Rio_writen을 통해 쓴 command을

Rio_readlineb를 통해 읽어온다. 그리고 해당 command에 따라 기능이 달라진다. 먼저, command가 show일 경우에는 print_preorder를 이용해 tree를 출력한다. command가 exit일 경우에는 connfd를 Close해줌으로써 더 이상 처리하지 않는다. command가 buy이거나 sell일 경우에는, buy or sell 하려는 주식을 search 함수를 통해 node를 찾고, data를 변경하고 client에 success를 출력한다.

- 요청이 더 이상 없다면 delTree를 통해 binary tree를 해제해준다.

- Task2 (Thread-based Approach with pthread)

- binary tree data structure 등의 기본적인 내용은 task1과 유사하다. 대신, task1과의 가장 큰 차이점은 concurrent하게 data structure에 접근하므로 semaphore를 활용해야 한다는 것이다. 어떤 client가 i종목 주식을 읽을 때, 다른 client가 i값을 update하게 되면 올바르게 동작하지 않는다. 즉, readers-writers problem solution을 고려한 fine-grained locking이 필요하다. 이를 위해서 item data에 sem_t mutex, w 변수를 추가해 주었다.

- First readers-writers problem의 solution을 적용하면, show에서 호출하는 print_preorder가 reader에 해당되고, buy와 sell은 writer에 해당된다.

3. 구현 결과

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

- 1개의 stockserver가 열려 있고, 2개의 stockclient가 정상적으로 동작하는지 확인.

<server 화면>

```
cse20181589@cspro:~/sp_proj2/task_1$ cat stock.txt
1 145 1000
5 41 3700
3 151 1200
2 60 2000
4 39 5000
6 113 4200
7 36 3200
8 31 6800
9 188 5900
10 109 2300
cse20181589@cspro:~/sp_proj2/task_1$ ./stockserver 60017
Connected to (172.30.10.8, 56792)
server received 5 bytes on fd 4
Connected to (172.30.10.8, 56800)
server received 9 bytes on fd 5
server received 5 bytes on fd 5
server received 10 bytes on fd 4
server received 5 bytes on fd 4
server received 5 bytes on fd 5
client on fd 5 disconnected
server received 5 bytes on fd 4
client on fd 4 disconnected
^C
cse20181589@cspro:~/sp_proj2/task_1$ cat stock.txt
1 145 1000
5 41 3700
3 151 1200
2 60 2000
4 49 5000
6 113 4200
7 36 3200
8 21 6800
9 188 5900
10 109 2300
cse20181589@cspro:~/sp_proj2/task_1$
```

<client1(fd4)화면>

```
cse20181589@cspro8:~/sp_proj2/task_1$ ./stockclient 172.30.10.11 60017
show
1 145 1000
5 41 3700
3 151 1200
2 60 2000
4 39 5000
6 113 4200
7 36 3200
8 31 6800
9 188 5900
10 109 2300
sell 4 10
[sell] success
show
1 145 1000
5 41 3700
3 151 1200
2 60 2000
4 49 5000
6 113 4200
7 36 3200
8 21 6800
9 188 5900
10 109 2300
exit
cse20181589@cspro8:~/sp_proj2/task_1$
```

<client2(fd5)화면>

```
cse20181589@cspro8:~/sp_proj2/task_1$ ./stockclient 172.30.10.11 60017
buy 8 10
[buy] success
show
1 145 1000
5 41 3700
3 151 1200
2 60 2000
4 39 5000
6 113 4200
7 36 3200
8 21 6800
9 188 5900
10 109 2300
exit
cse20181589@cspro8:~/sp_proj2/task_1$
```

server에서 concurrent하게 client를 처리할 수 있음을 잘 보여준다. 또한, 서버 종료 시점에 stock.txt가 update가 잘 되어있음도 확인할 수 있다. client 1과 client 2에서 buy, sell, show, exit이 모두 정상적으로 작동하고, 만약에 buy시에 주식의 수가 부족하다면 Not enough 메시지를 출력한다. 이 예시는 Task2에서 작동 여부를 확인하도록 하겠다.

- Task2 (Thread-based Approach with pthread)

- 1개의 stockserver가 열려 있고, 2개의 stockclient가 정상적으로 동작하는지 확인.

<server 화면>

```
cse20181589@cspro:~/sp_proj2/task_2$ cat stock.txt
1 95 1000
5 6 3700
3 72 1200
2 266 2000
4 49 5000
6 10 4200
7 77 3200
8 81 6800
9 15 5900
10 122 2300
cse20181589@cspro:~/sp_proj2/task_2$ ./stockserver 60017
Connected to (172.30.10.8, 56824)
thread 1148315392 received 5 (5 total) bytes on fd 4
thread 1148315392 received 9 (14 total) bytes on fd 4
Connected to (172.30.10.8, 56826)
thread 1139922688 received 10 (24 total) bytes on fd 5
thread 1139922688 received 5 (29 total) bytes on fd 5
thread 1148315392 received 10 (39 total) bytes on fd 4
thread 1139922688 received 5 (44 total) bytes on fd 5
thread 1139922688 received 5 (49 total) bytes on fd 5
thread 1148315392 received 5 (54 total) bytes on fd 4
^C
cse20181589@cspro:~/sp_proj2/task_2$ cat stock.txt
1 95 1000
5 6 3700
3 72 1200
2 266 2000
4 49 5000
6 44 4200
7 77 3200
8 81 6800
9 35 5900
10 122 2300
cse20181589@cspro:~/sp_proj2/task_2$
```

<client1(fd4)화면>

```
cse20181589@cspro8:~/sp_proj2/task_2$ ./stockclient 172.30.10.11 60017
show
1 95 1000
5 6 3700
3 72 1200
2 266 2000
4 49 5000
6 10 4200
7 77 3200
8 81 6800
9 15 5900
10 122 2300
buy 9 20
Not enough left stock
sell 6 34
[sell] success
exit
cse20181589@cspro8:~/sp_proj2/task_2$
```

<client2(fd5)화면>

```
cse20181589@cspro8:~/sp_proj2/task_2$ ./stockclient 172.30.10.11 60017
sell 9 20
[sell] success
show
1 95 1000
5 6 3700
3 72 1200
2 266 2000
4 49 5000
6 10 4200
7 77 3200
8 81 6800
9 35 5900
10 122 2300
show
1 95 1000
5 6 3700
3 72 1200
2 266 2000
4 49 5000
6 44 4200
7 77 3200
8 81 6800
9 35 5900
10 122 2300
exit
cse20181589@cspro8:~/sp_proj2/task_2$
```

Thread-based Approach에서도 정상적으로 작동하고 있음을 확인할 수 있다. 특히, buy에서 주식의 수가 부족할 때, Not enough left stock 메시지를 출력함을 볼 수 있다. 또한, 서버 종료 시점에 stock.txt 역시 정상적으로 update된다.

4. 성능 평가 결과 (Task 3)

- Performance Evaluation은 총 두가지를 수행할 예정이다. 확장성에 대한 분석과, 워크로드에 따른 분석 두가지를 수행한다.

1) 확장성에 대한 분석

- 측정 시점: listenfd를 통해 요청을 듣는 시점부터, 마지막 command의 처리가 끝나는 시점까지 걸린 시간을 gettimeofday 함수를 이용해 측정할 것이다. 이후, 총 소요된 시간을 command 수로 나누면, 1초당 동시에 처리한 command의 수를 분석한다.

- 결과 예측: event-based는 multi-core를 활용하지 못하는 반면, thread-based는 multi-core를 활용할 수 있으므로, thread-base가 동시처리율이 더 높을 것이다 (측정 PC는 4-cored CPU를 사용한다).

- 출력 결과 캡처

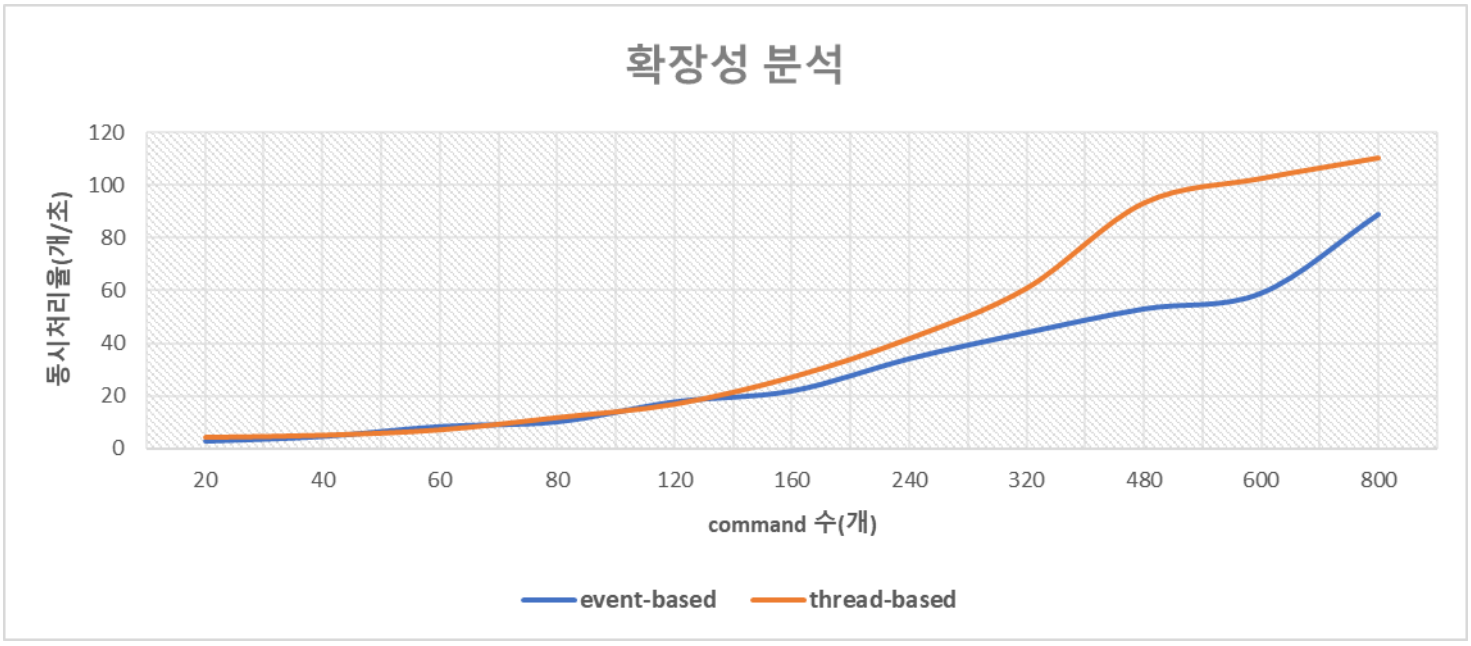
```
cse20181589@cspro:~/sp_proj2/task_1$ ./stockserver 60017
Connected to (172.30.10.8, 56890)
Connected to (172.30.10.8, 56892)
server received 8 bytes on fd 4
Connected to (172.30.10.8, 56894)
server received 10 bytes on fd 5
Connected to (172.30.10.8, 56896)
server received 9 bytes on fd 6
server received 10 bytes on fd 7
server received 8 bytes on fd 4
server received 9 bytes on fd 5
server received 9 bytes on fd 6
server received 8 bytes on fd 7
server received 9 bytes on fd 4
server received 10 bytes on fd 5
server received 8 bytes on fd 6
server received 9 bytes on fd 7
server received 9 bytes on fd 4
server received 9 bytes on fd 5
server received 8 bytes on fd 6
server received 9 bytes on fd 7
elapsed time: 8380352 microseconds
```

```
cse20181589@cspro8:~/sp_proj2/task_1$ ./multiclient 172.30.10.11 60017 4
child 41319
child 41320
child 41321
child 41322
[buy] success
[sell] success
[sell] success
[sell] success
[buy] success
[sell] success
[sell] success
[buy] success
[sell] success
[sell] success
[buy] success
[sell] success
[buy] success
[sell] success
[buy] success
[buy] success
cse20181589@cspro8:~/sp_proj2/task_1$
```

client수=4, 각 client command수=10, 총 command수=40

- 분석 결과

command 수(개)	event-base(초)	thread-base(초)	event 동시처리율(개/초)	thread 동시처리율(개/초)
20	6.709055	4.553367	2.981045766	4.392354054
40	8.380352	7.689025	4.773069198	5.202220047
60	7.026278	8.257366	8.53937177	7.266239622
80	7.734934	6.717174	10.34268683	11.90977039
120	6.703863	7.079715	17.90012714	16.94983485
160	7.268309	5.884958	22.0133734	27.18795954
240	7.013684	5.744218	34.21882138	41.7811441
320	7.249265	5.262464	44.14240616	60.80801693
480	9.036551	5.155549	53.11761091	93.10356666
600	10.160216	5.85905	59.05386263	102.4056801
800	8.994565	7.261344	88.94260034	110.1724419



- 예상한 대로, command 수가 늘어날수록 thread-based concurrent server가 동시처리율이 높아짐을 확인할 수 있다.

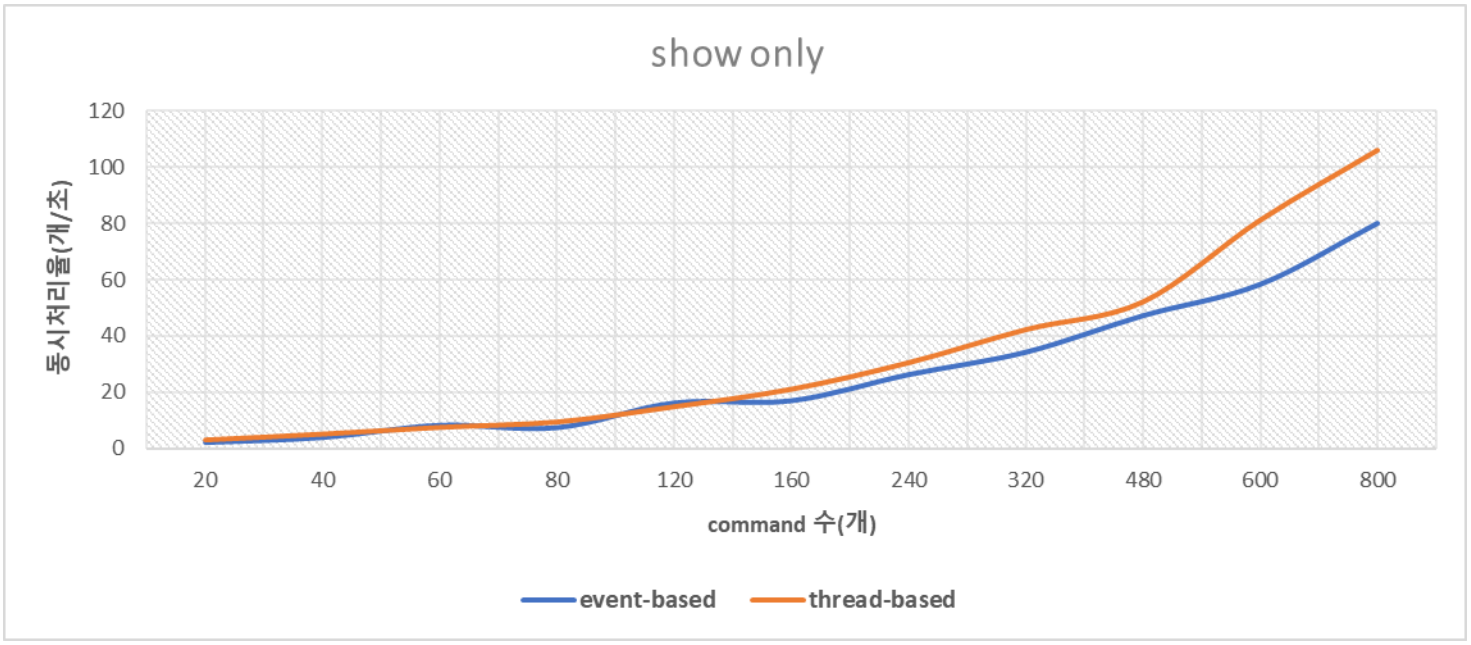
2) 워크로드에 따른 분석: (1)모든 client가 show만 요청하는 경우, (2)모든 client가 buy 또는 sell을 요청하는 경우, (3)세가지를 모두 섞어 random하게 요청하는 경우의 총 세가지로 나누어 분석한다.

- 측정 시점: 1)과 동일하다.

- 결과 예측: 모든 client가 show를 요청하는 경우에, 요청이 있을 때마다 binary tree를 순회하며 모든 node의 data를 출력해야 하므로, buy+sell을 요청하는 경우보다 느릴 것이다.

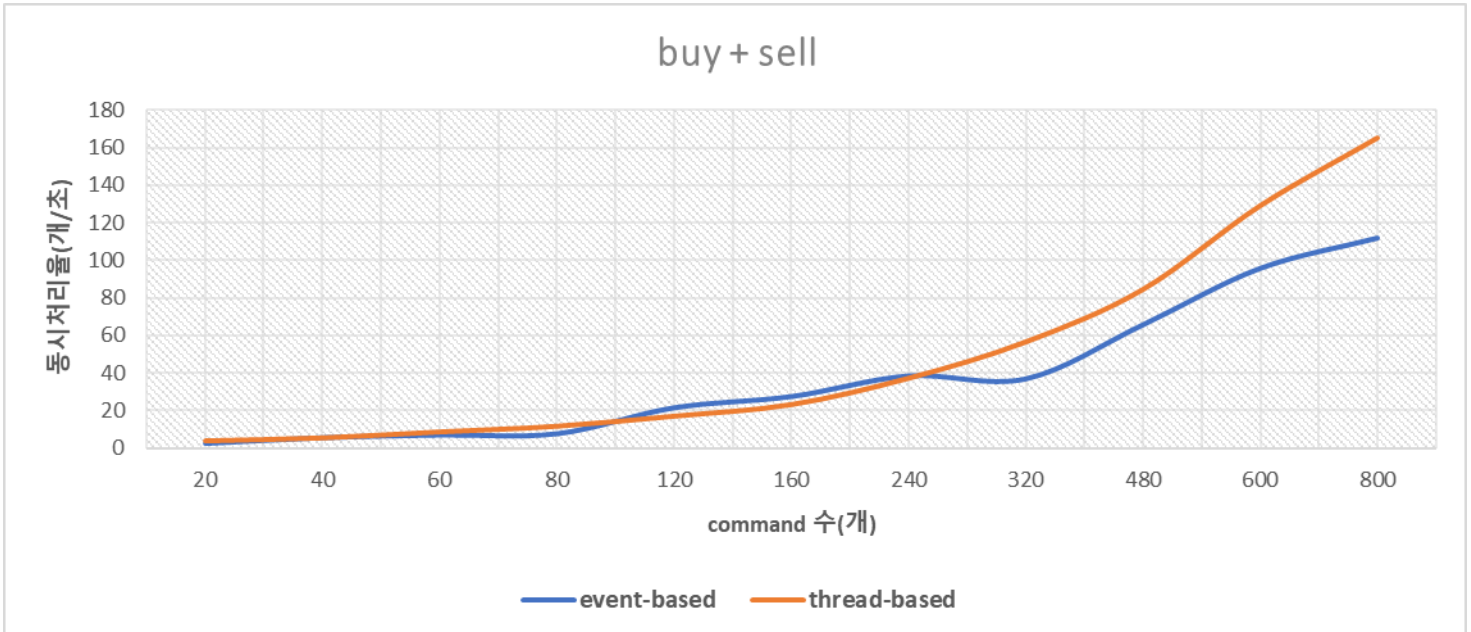
(1) 모든 client가 show만 요청하는 경우

command 수(개)	event-base(초)	thread-base(초)	event 동시처리율(개/초)	thread 동시처리율(개/초)
20	8.704775	6.516352	2.297589541	3.069201909
40	9.823449	7.768176	4.071889618	5.149213921
60	7.120056	7.935213	8.426900013	7.561233706
80	10.551757	8.448457	7.581675734	9.469184728
120	7.344087	8.051525	16.33967571	14.90400887
160	9.324987	7.579212	17.15820086	21.11037401
240	9.079326	7.854337	26.43368021	30.5563665
320	9.303998	7.560452	34.39381651	42.32551176
480	10.128854	9.200557	47.38936902	52.17075444
600	10.252362	7.384992	58.52309936	81.24585646
800	9.973514	7.540331	80.2124507	106.0961382



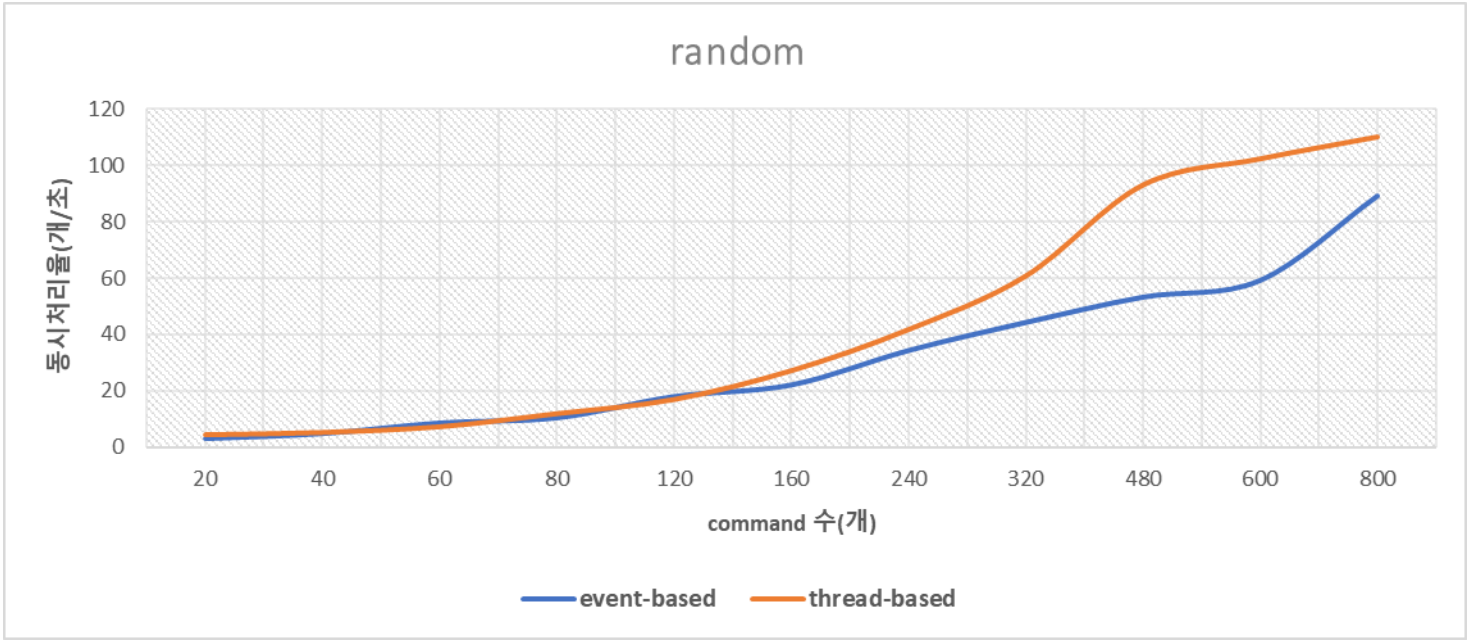
(2)모든 client가 buy 또는 sell을 요청하는 경우

command 수(개)	event-base(초)	thread-base(초)	event 동시처리율(개/초)	thread 동시처리율(개/초)
20	7.307674	5.240233	2.736848962	3.816624184
40	6.881068	7.373701	5.813051114	5.424684294
60	8.262994	6.949933	7.26129052	8.633176751
80	10.070862	6.86899	7.943709287	11.64654483
120	5.513326	7.061507	21.76544612	16.99353976
160	5.767381	6.895953	27.74222823	23.20201428
240	6.199521	6.44344	38.71266829	37.24718473
320	8.616859	5.653355	37.13650183	56.60355665
480	7.276016	5.67473	65.97016829	84.58552213
600	6.249781	4.644292	96.00336396	129.1908433
800	7.135228	4.839676	112.1197529	165.3003218



(3)세가지를 모두 섞어 random하게 요청하는 경우

command 수(개)	event-base(초)	thread-base(초)	event 동시처리율(개/초)	thread 동시처리율(개/초)
20	6.709055	4.553367	2.981045766	4.392354054
40	8.380352	7.689025	4.773069198	5.202220047
60	7.026278	8.257366	8.53937177	7.266239622
80	7.734934	6.717174	10.34268683	11.90977039
120	6.703863	7.079715	17.90012714	16.94983485
160	7.268309	5.884958	22.0133734	27.18795954
240	7.013684	5.744218	34.21882138	41.7811441
320	7.249265	5.262464	44.14240616	60.80801693
480	9.036551	5.155549	53.11761091	93.10356666
600	10.160216	5.85905	59.05386263	102.4056801
800	8.994565	7.261344	88.94260034	110.1724419



-예상한 대로, 모든 client가 show를 요청하는 경우에, buy+sell을 요청하는 경우보다 느린 것을 알 수 있다. 그리고 random한 경우에는, show와 buy+sell의 사이의 동시처리율을 보인다.