Multicore Programming Project 2

담당 교수 : 최재승 교수님

이름 : 손채훈

학번 : 20181085

1. 개발 목표

개발 목표는 다수 client들의 동시 접속 및 client들에 대한 서비스 제공을 위한 concurrent stock server를 구축하는 것이다. 서버는 주식 정보를 저장하고, 클라이언트 측에서 요청한 주식 사기, 팔기, 정보 조회등의 서비스를 제공하게 된다. Task 1과 Task 2 모두 위에서 언급한 동일한 목표를 위해 개발되는데, 큰 틀에서의 구현방식에서만 차이가 있게 된다. concurrency를 구현하는데, Task 1의 경우에는 단일 프로세스/쓰레드 속에서의 I/O multiplexing 기술을 사용하게되며, Task 2의 경우에는 여러개의 쓰레드를 이용하게 된다.

2. 개발 범위 및 내용

A. 개발 범위

- **아래 항목을 구현했을 때의 결과를 간략히 서술**

1. Task 1: Event-driven Approach

   Event-driven 접근 기법을 통해 concurrency를 구현하면, 각각의 client가 서버측에서 호출한 Accept 함수를 통해 새로운 소켓 파일 디스크립터를 생성하게 되고 그 값이 add_client함수를 통해서 connfd와 그와 관련된 정보들을 저장하고 있는 pool 구조체에 저장된다. 그 후 연결된 connfd에 I/O 이벤트가 발생하면 이벤트를 관찰 중이던 select함수에 의해 선택되어서, 서버측에 의해 서비스를 제공받게 된다.

2. Task 2: Thread-based Approach

   Thread-based 접근 기법을 통해 concurrency를 구현하면, main 함수의 for 반복문에서 다수의 thread를 미리 생성해두게 되고, 이 thread들이 계속 정해진 순서 없이 switching 하면서 client에게 서비스하게된다. Thread routine이 무한 루프로 구성되어있어 한 client에게 서비스를 제공한 후에는 또 다음 client에게 서비스를 제공할 수 있도록 구현된다.

3. Task 3: Performance Evaluation

   Task1, Task2에서 구현한 방식을 평가하는 metric을 제시하고, 이를 통해 성능 분석을 하였다. 평가를 위해 사용한 척도는 client의 수, 그리고 제공되는 서

비스의 종류이다. multiclient.c 파일의 메인 함수 부분의 시작 부분과 리턴 까지를 소요 시간으로 설정하였으며, project 2 specififcation에 제공된 링크에서의 elapse time을 측정하는 방식을 차용하였다.

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

Task1에서 활성화된 client들은 pool 구조체에 저장되게 된다. pool 구조체는 main함수에서 호출되는 init_pool함수에 의해서 initialize되게 되는데, 그 후 서버는 무한 루프에 들어가게 된다. 서버는 루프가 한번 실행될 때마다 select 함수를 통해서 2가지 I/O 이벤트를 관찰하도록 한다. 첫번째는 새로운 client의 connect 요청이고 그것은 Accept 함수를 통해서 받아들여진다. 두번째로는 이미 connect된 connfd로 부터의 입력이다. 서버는 add_client 함수를 통해서 pool에 connfd를 추가하게 되고, 결국 check_client를 호출하여 서비스를 제공하게 된다. 함수의 실행과정을 보면 한번에 하나의 client에 대한 서비스가 모두 끝난 후에야 다음 client에게 서비스를 제공할 수 있다. 그래서 엄밀한 의미의 fine-grained한 동시성을 구현하기는 힘들 수 있다.

✓ epoll과의 차이점 서술

select 함수가 부차적인 오버헤드를 발생시키게 되는 이유는 등록된 파일 디스크립터들을 운영체제에게 일괄적으로 보냈다가 다시 I/O 이벤트가 일어났는 지에 대한 결과를 받아오게 되는데, 이 과정을 반복문을 통해서 계속 반복해야하기 때문이다. epoll은 초기에 파일 디스크립터들을 넘기고, 그 중에서 I/O 이벤트가 발생하거나 다른 이유로 갱신된 부분들만 운영체제와 주고 받기 때문에, 그 과정에서의 오버헤드를 줄일 수 있고, 반복문도 갱신된 파일 디스크립터에 대해서만 실행시키면 되기 때문에 효율적이다. epoll 에서 select함수에 대응되는 함수로 epoll_wait가 있는데, select와 달리 인자로 관찰 대상의 정보를 매번 전달할 필요성이 존재하지 않는다.

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

  ✓ Master Thread의 Connection 관리

  ✓ Master Thread에서 전반적인 connection관리는 다중 프로세스를 이용한 concurrency 구현과 큰 틀에서는 유사하다고 할 수 있다. 구현한 stockserver 코드에서는 Pthread_create 함수를 실행 할 때, 인자로 connfd를 전달하고 있지는 않다. 왜냐하면 먼저 thread를 생성한 후 무한 루프 속에서 Accept가 일어나기 때문이다. Accept를 통해 생성된 connfd를 peer thread에 전달하기 위해서 master thread에서 sbuf 패키지의 버퍼에 insert해준 후, peer thread 에서 pop하여 참조하고 있다. 각 peer thread에서는 client에 대한 서비스가 끝난 후에 connfd를 잘 close해주고 있다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  Worker thread들은 처음에 생성된 이후에 client와의 통신이 일어나기 전까지 blocked된 상태로 대기하게 된다. 위에서 설명했듯 accept 함수를 통해서 요청을 받으면, sbuf_t 구조체의 buf에 connfd를 삽입하고 worker thread에서 그 connfd를 remove함으로써 통신이 일어나게 된다. 하나의 client에 대한 서비스가 끝난 후에 다음 connfd가 존재한다면 서비스를 재시작하게 되고, 아니면 다시 blocked 된 상태로 대기하게 된다. 각 peer thread에서는 Pthread_detach 함수를 자기 자신의 thread id에 대해서 call하여, 스스로 reaping 해주고 있다.

- **Task3 (Performance Evaluation)**

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

  stockserver의 성능을 측정하기 위해서 동시처리율이라는 metric을 정의하고자 한다. 동시 처리율은 다음과 같은 식으로 정의될 수 있다.

  *동시 처리율 = ((클라이언트의 수 * 각 클라이언트의 요청 수) / 소요 시간) * 10^4*

  동시 처리율을 이와 같이 정의한 이유는, 총 요청 수를 소요 시간으로 나눠주면 직관적으로 코드의 성능을 측정할 수 있다고 생각했기 때문이다. 뒤에 10의 4제곱을 곱해준 것은 표현의 편리성을 위해서이다. 소요 시간은 앞서

말했 듯 multiclient.c 파일이 실행되고 끝날 때 까지의 시간이고, 클라이언트의 수는 1, 4, 16, 64, 100의 순서로 늘려가면서 실험을 진행하였다. 또한 multiclient 코드를 변형하면서,

1. buy or sell 요청만 서버에 들어오는 경우

2. show 요청만 계속해서 서버에 들어오는 경우

3. buy, sell, show 요청 모두 무작위로 서버에 들어오는 경우로 나눠서 실험을 이어 진행하였다.

실험에 사용되는 모든 stock.txt 파일의 item개수는 5개로 고정하였다.

✓ Configuration 변화에 따른 예상 결과 서술

show 요청이 계속해서 서버에 들어오는 경우가 가장 많은 elapse time이 소요될 것이라고 예상한다. 왜냐하면 show는 트리의 모든 노드를 순회해야하는 반면 buy나 sell의 경우는 노드를 찾으면 바로 반환해주면 되기 때문이다. 즉 위의 세가지 경우 중 1번이 가장 높은 동시 처리율을 보일 것이라고 예상한다.

그리고 I/O multiplexing을 통한 concurrency 구현 보다는 thread를 통한 concurrency 구현이 좀더 높은 동시처리율을 가질 것 같다. 왜냐하면 thread model이 I/O multiplexing 보다는 훨씬 fine-grained한 동시성 구현이 가능할 것이라고 예상하기 때문이다.

C. 개발 방법

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

Task1)

Task1의 경우는 stockserver.c 소스 코드 파일에 csapp 교재의 코드를 기반으로 수행하였다. 그 코드를 제외하고 내가 임의로 추가한 부분에 대해서 설명하겠다.

1. item 구조체를 구현하였다. 종목 이름(ID), 잔여 주식 수(left), 가격(price)의 정보를 갖고 있다.

2. 주식 정보를 저장하기 위한 이진 트리를 구현하기위한 node 구조체를 구현하였다. item구조체를 가리키는 변수 data, 좌, 우 자식노드를 가리키기 위한 포인터를 내포하고 있다.

3. init_tree 함수를 구현하였다. stock.txt 파일을 읽기 모드로 열고 파일 안에 있는 정보를 한 줄 씩 읽어서 tree node 하나에 저장한다. tree node는 힙에 동적 할당하는 방식으로 생성하였다. 파일의 모든 정보를 트리에 옮긴 후 파일을 닫아주었다.

4. insert_node 함수를 구현하였다. 트리에 주식 정보를 삽입하기 위한 함수로 init_tree함수를 호출하면 그 안에서 호출된다. main에서 직접 호출되는 경우는 없다.

5. free_node 함수를 구현하였다. ctrl+c를 입력받으면 sigint handler가 호출되면서 free_node함수를 호출하게 되는데, 주식 정보를 저장하고 있던 tree node들의 메모리를 해제해주는 역할을 한다.

6. search 함수를 구현하였다. 인자로 들어온 ID와 일치하는 노드를 찾으면 반환해준다. 일치하는 노드가 없으면 null을 반환해준다.

7. show 함수를 구현하였다. client 측에서 show 요청이 들어오면 실행되게되는 함수로, 버퍼에 트리 노드의 내용을 전부 넣어준다. 버퍼의 내용은 rio_writen 함수를 통해서 connfd에 작성되게 된다.

8. buy 함수를 구현하였다. search 함수를 통해서 인자로 들어온 ID와 일치하는 노드를 찾으면, 사려고하는 주식 수보다 잔여 주식수가 크거나 같으면 사가는 사려고하는 주식 수 만큼 잔여 주식수를 차감한 후에 함수를 종료한다. 적절한 메

시지도 출력해준다. 만약 그렇지 않다면 충분한 잔여 주식수가 없다는 에러 메시지를 출력한 뒤 종료된다.

9. sell 함수를 구현하였다. buy함수에서 조건문을 통해 경우를 분기하는 점만 빼면 buy함수의 구현과 동일하다.

10. sigint_handler를 따로 구현하였다. sigint 시그널이 서버 프로세스에 들어오면 리슨 용도의 디스크립터를 close하고, 파일을 열어서 store_data 함수를 통해 현재 주식 정보를 작성해준뒤, 파일을 닫고 free_node 함수를 통해 트리의 메모리를 해제하고 프로세스를 종료한다.

11. sigint_handler안에서 호출되는 store_data 함수를 구현하였다. 트리의 정보를 stock.txt 파일에 저장해준다.

Task2)

Task2도 교재의 코드를 기반으로 수행하였고, Task1에서 구현된 함수와 구조체를 모두 포함하고 있으므로, 그 함수와 구조체 안에 추가된 부분에 대해서 작성하겠다.

1. item 구조체의 내용을 수정하였다. item 구조체 안에 현재 노드에 대해 읽기 작업을 수행하고있는 thread의 수를 나타내는 readcnt 변수와, 각각 writer, reader를 위한 세마포어인 w, r을 추가하였다.

2. init_tree 함수의 내용을 수정하였다. 파일의 한 줄을 읽어와서 트리의 노드 하나에 저장하는 과정에서 세마포어 w, r을 1의 값으로 initialize 하고 readcnt의 값을 0으로 초기화해주는 과정을 추가하였다.

3. show 함수의 내용을 수정하였다. show 함수에서 노드의 내용을 읽어들이고 데이터를 버퍼에 넣어주는 과정에서 세마포어 r을 통해 상호 베타성을 획득한 뒤 readcnt 값을 1 증가시켜 주었다. 그 후 세마포어 r의 값을 다시 1 증가시키고, 만약 readcnt가 1이라면 세마포어 w의 값을 1 감소시켜, write 작업이 이루어질 수 없도록 lock한다. 읽기 작업이 끝난 후에는 다시 세마포어 r을 통해서 readcnt 값을 안전하게 감소시켜주고, 마지막 reader라면 w세마포어 값을 1 증가시켜 write 작업이 이루어질 수 있도록 해준다. 마지막으로 세마포어 r의 값을 1 증가시켜준다.

4. buy 함수와 write함수에서 세마포어 w를 통해 상호 베타성을 획득한 이후에 안전하게 노드의 데이터가 변환될 수 있도록 하였다.

3. 구현 결과

- **2번의 구현 결과를 간략하게 작성**

Task1에서 의도한 대로 concurrent 하게 작동하는 I/O multiplexing을 구현할 수 있었다. 하지만 확실히 순수하게 동시성을 구현하는 데에만 thread를 이용한 stockserver보다 2~3배 많은 코드량이 필요하였다. 그리고 예상한대로 한 connection이 progress하는 동안 다른 connection은 서비스 받을 수 없었다.

Task2에서는 multi thread를 사용하는데에 있어서 치명적인 문제들이 확실히 semaphore를 사용함으로써 race 문제를 어느정도 해결할 수 있었다. 그러나 다른 요인에 의한 race condition은 존재할 가능성이 있다.

- **미처 구현하지 못한 부분에 대해선 디자인에 대한 내용도 추가**

csapp 교과서에서는 내가 구현한 reader 우선 방식이 아닌 writer우선 방식에 대한 언급도 하고 있다. 사실 내 생각에는 이러한 주식 서버에서는 reader 우선 방식으로 reader - writer problem을 해결하되, 최대한 fine-grained를 구현하는 것이 가장 좋은 방법인 것 같은데, writer를 우선으로 하는 방식도 필요한 경우가 있을 것 같다. 정확한 최신 데이터를 필요로 하는 경우는 그럴 것이다. 그런 경우에는 이번에는 반대로 writer가 데이터를 쓰려고 할 때 writer mutex를 lock해준 후 writer count를 1 증가시켜준다. 그 후 writer count가 1이라면 reader를 block해준 후 writer mutex를 unlock 해준다. 데이터를 쓰고 난 후에는 writer mutex를 lock해준 후 writer count를 1 감소시켜준다. 그 후 writer count가 0이라면 reader를 unblock해주고, writer mutex를 unlock 해준다.

4. 성능 평가 결과 (Task 3)

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

1) 실험 환경

서버 (cspro9.sogang.ac.kr)

```
top - 10:03:13 up 42 days,  5:01, 25 users,  load average: 48.37, 48.42,
Tasks: 857 total,  49 running, 784 sleeping,  22 stopped,   2 zombie
%Cpu(s): 99.9 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,
MiB Mem :  64221.4 total,  18787.9 free,  17009.2 used,  28424.2 buff/cac
MiB Swap:   8192.0 total,   8192.0 free,      0.0 used.  46506.5 avail Me
```

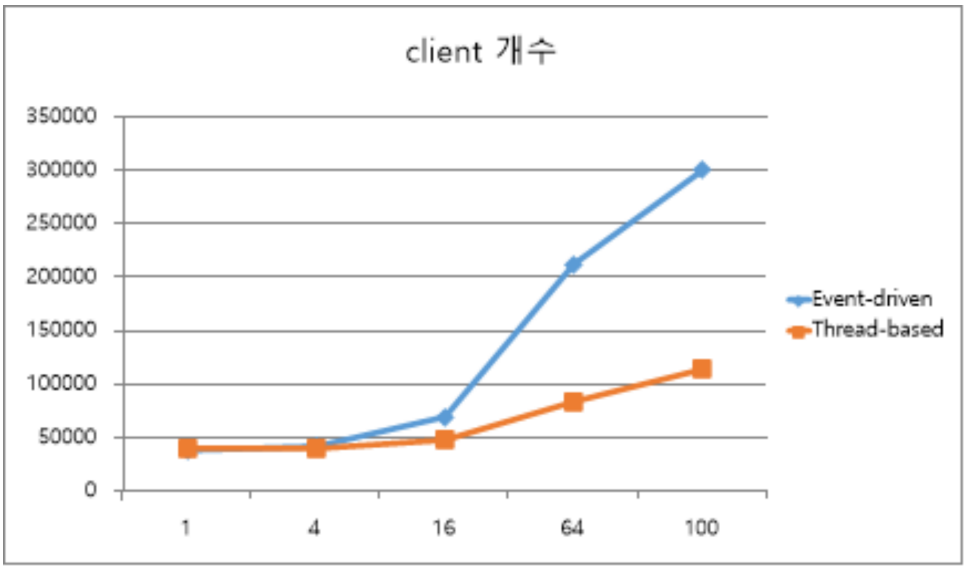
클라이언트 (cspro.sogang.ac.kr)

```
top - 03:03:52 up 41 days, 31 min, 55 users,  load average: 125.08, 125.15,
Tasks:  21 total,   1 running,  20 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.2 us,  0.6 sy,  0.0 ni,  0.1 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0
KiB Mem : 65853320 total, 41743192 free, 13144740 used, 10965388 buff/cache
KiB Swap:   628732 total,    94744 free,   533988 used. 51296488 avail Mem
```

2) client 개수에 따른 성능

client 개수에 따른 event driven 방식과 thread based 방식의 성능 변화는 위와 같은 환경에서 실험되었다. 한 client당 10개의 요청을 서버에 보내며, thread의 수는 4개, sbuf size는 16이다. 결과는 10번 시행하여 가장 평균에 가까운 값의 캡쳐를 차용하였으며, client 개수가 1, 4, 16, 64, 100 순서로 증가한다. 원래 코드대로 show, buy, sell 요청을 random하게 서버로 전달한다. 주식 종목 개수는 5개로 고정하였다.

N \ model	event driven	thread based
1	elapsed time: 36812 microseconds	elapsed time: 39342 microseconds
4	elapsed time: 40185 microseconds	elapsed time: 39114 microseconds
16	elapsed time: 68006 microseconds	elapsed time: 47098 microseconds
64	elapsed time: 210982 microseconds	elapsed time: 83010 microseconds
100	elapsed time: 299016 microseconds	elapsed time: 113307 microseconds



event-driven, thread-based 서버 모두에서 clinent 개수가 증가할 수록 위 그래프와 같이 소요 시간이 증가하는 것을 알 수 있다. 예측한대로 client 수가 많을 수록 더 높은 수준의 concurrency가 구현되어 있는 thread 서버의 소요 시간이 더 디게 증가하는 것을 알 수 있다. 확장성 측면에서 thread 서버가 더 높은 효율을 갖는다는 것을 알 수 있다. 다음으로는 위에서 정의한 동시처리율 metric에 따른 그래프를 제시하겠다.

동시 처리율은 두 모델 모두 client가 많아질 수록 개선되는 형태를 띄었지만, thread based 모델과 달리 Event-driven 모델은 확실히 동시 처리율이 상승하다가 비슷한 수치를 유지하였다. thread based 모델은 client가 64개일 때 대폭 개선된 동시 처리율을 보여주었고, 100개 일 때 증가폭은 살짝 감소했으나, 여전히

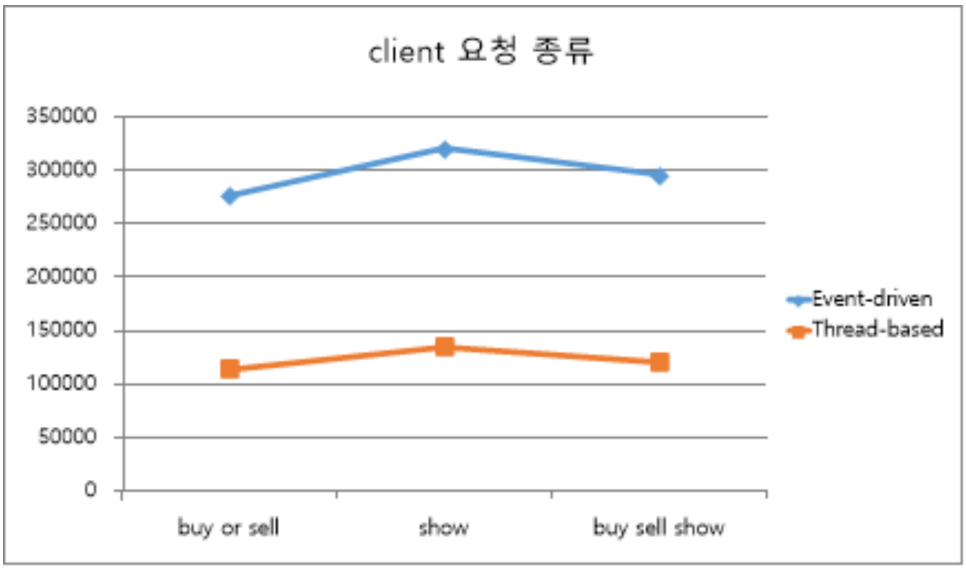
가파른 증가폭을 보여주었다. 역시 thread-based 모델이 더 높은 concurrency에 대한 확장성을 보여준다고 할 수 있다.

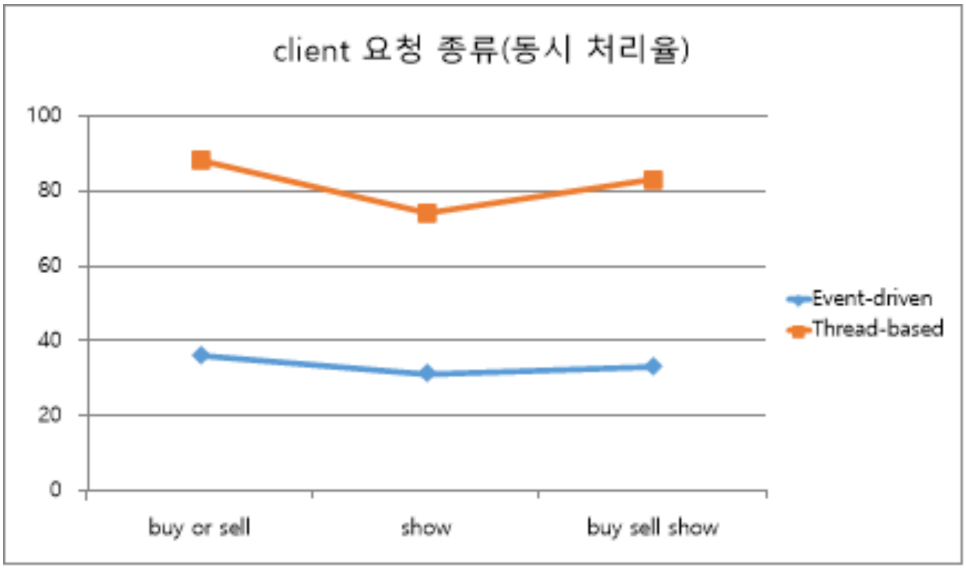
3. client의 요청 종류에 따른 성능

Client의 요청 종류에 따른 event driven 방식과 thread based 방식의 성능 변화는 역시 위와 같은 환경에서 실험되었다. 한 client당 10개의 요청을 서버에 보내며, thread의 수는 4개, sbuf size는 16이다. 결과는 10번 시행하여 가장 평균에 가까운 값의 캡쳐를 차용하였으며, 서로 다른 요청 옵션을 서버로 전달한다. 주식 종목 개수는 효과를 좀더 선명하게 관찰하기 위해서 10개로 상향 조정하여 실험을 진행하였다. buy와 sell은 실행 시 주식 종목 번호와 일치하는 노드를 찾아서 처리하는 등의 연산이 유사하기 때문에 한 그룹으로 묶어서 실험하였다. client 수는 100개로 고정하였다.

request \ model	Event driven	thread based
buy or sell	elapsed time: 274587 microseconds	elapsed time: 112736 microseconds
show	elapsed time: 319174 microseconds	elapsed time: 133797 microseconds
buy sell show	elapsed time: 294393 microseconds	elapsed time: 119622 microseconds

위 도표를 보면 이전에 예측한 결과대로 elapsed time이 도출 된 것을 알 수 있다. 전반적으로는 event-driven model이 더 많은 시간이 소요된 것과는 별개로, server에 들어온 요청의 종류에 따라 실행 시간이 유의미한 형태를 띈다는 것을 알 수 있다. buy나 sell만 이루어지는 경우 노드 전체를 순환해야하는 경우는 sell만 같은 수의 요청을 하는 것보다는 적을 수 밖에 없으므로 더 적은 시간이 소요되었고, sell은 모든 노드를 순회해야되기 때문에 buy, sell, show 모든 요청을 임의로 실행하는 경우보다 많은 시간이 소요되었다.





위 그래프들은 client 요청 종류에 따른 소요 시간과 동시 처리율을 나타낸 그래프이다. 소요 시간과 동시 처리율은 반비례하는 관계이므로 그래프의 형태가 반대로 나타나는 것을 확인할 수 있다.