System Programming Project 5

담당 교수 : 김영재 교수님

이름 : 최재원

학번 : 20171700

1. 개발 목표

I/O multiplexing을 이용한 event-based 및 thread-based concurrent server를 구축하여 기본 TCP echo Client의 socket 통신을 concurrent로써 확장한다. 해당 프로그램은 client에서 요청하는 'buy, sell, show'에 따라서 Stock database에 있는 현재 주식량을 최신화하고 request에 대응하는 message 등을 client에게 보내고 출력한다.

그리고 Concurrent sever의 구현 방법에 따라서 상황에 따라 어떤것이 더 성능이 좋은지 비교해 본다.

2. 개발 범위 및 내용

A. 개발 범위

1. Select

서로 다른 ip로 접속한 client 에서 하나의 server로 buy, sell, show를 요청할 수 있고, 이에 따라 server에서 해당 요청에 맞는 메시지를 버퍼에 저장하여 rio_written을 통해서 넘겨주게 된다. show를 요청하면 현재 주식 정보에 대해서 client에서 출력할 수 있고, buy와 sell을 할 경우 success라는 메시지를 출력한다. 서로 다른 Ip로 접속한 client들에 대해서도 동일한 serer에서 컨트롤 하기 때문에, 한쪽에서 buy및 sell을 요청하여 주식 정보가 바뀌면, 다른 client에서도 바뀐 주식 정보에 대해서 접근하게 된다.

Multiclient로 접근할 수도 있는데, 해당 코드는 fork로 여러 client를 만들어 접근하게 되고, 이에 따른 concurrent processing이 이루어진다.

위까지의 출력에 관한 내용은 event와 thread가 동일하다.

Event-based concurrent server에 대한 구현은 기본적으로 I/O Multiplexing server이며 이는 각 소켓의 파일디스크립터를 리스트화하여 저장 시켜두고, 각 소켓의 입력/ 출력 스트림을 감시하다가 client로부터 데이터가 수신되어 입력스트림에 저장되면 이에 대한 것을 해당 소켓에게 client의 요청이 왔다고 알려주고, 서버와 통신을 하는 것을 말한다. 하지만 concurrent 하게 수행하기 위해서 pool이라는 구조체를 사용하며 listenfd가 해당 client를 가르키면, add_client()라는 함수를 통해 구조체에 적재해 둔 후, check_client()함수를 통해서 위에 대한 내용을 수행하게 된다.

결과적으로 multiclient 및 서로 다른 ip에서 접속한 여러 Client들의 요청을

Concurrent하게 처리할 수 있게 된다.

2. pthread

기본 socket 통신에 대한 출력 결과물을 event based에서와 같다. Pthread_create를 통해서 위의 event based에서와는 다르게 concurrent를 수행한다. Process의 fork와 비슷하게 client 하나마다 thread를 생성하여 worker로써 지정하게 되고, pool로 구현하지 않고 하나 하나 생성 수 detach를 통해서 소멸 시켜주었다. Pool보다는 생성 삭제에 대한 비용이 있고, 일일히 관리해야 된다는 단점이 있다. Pool은 각 작업에 대한 thread를 미리 생성해 놓기 때문에, 매번 thread를 만들 필요가 없기 때문에 성능이 해당 부분에 대해서 향상 될 수 있기 때문이다. 공통적으로는 data,및 code영역에 대한 동시 접근을 semaphore를 이용하여 lock을 함으로써 thread safe 상태를 만든다는 것이고, Thread base로 구현한 이번 프로그램에서는 thread함수에서 Store라는 함수를 호출하여 기본 주식 서버에 대한 내용및 semaphore에 대한 수행을 하게 된다.

B. 개발 내용

- **select**

✓ select 함수로 구현한 부분에 대해서 간략히 설명

우선 pool이라는 구조체를 사용하고, 해당 구조체의 clientfd 배열을 통해서 concurrent를 수행하기 때문에, init_pool() 함수를 통해 client의 fd를 연결하기 저에 구조체를 초기화 해주고, FD_ISSET 함수로 fd를 확인하게 되면 accept()함수로 연결 후, add_client() 함수를 통해서 clientfd에 연결된 client를 추가해 준다.

이 후 check_clients() 함수에서 일괄적으로 rio_readlineb() 함수로 client의 request를 buf에 받아오고, 이 내용을 통해서 store()함수를 호출하여 request에 대한 내용을 수행한다. 여기까지의 일련의 수행이 select 함수의 내용을 concurrent하게 수행하기 위해서 구현한 방식이다.

✓ stock info에 대한 file contents를 memory로 올린 방법 설명

sever 코드 실행시, 처음에 stock.txt로 부터 내용을 읽어와서 binary tree에 저장하게 된다. Binary tree로 구현한 이유는 show에 대한 request가 들어올 시, 빠른 속도로 stock에 대한 information을 가져오기 위함이다. Buy와 sell등 여러 수행을 하다가, Client와의 Connection이 끊기면, Stock.txt에 수정된 내용을 update하게

된다. 이 전까지는 tree에서만 바뀌게 되며, 결과적으로 update는 접속한 client 수만큼 하게되며, update 시기는 concurrent 수행에 의해서, connection이 끊기지 않은 다른 client와의 수행과 동시에 이루어 질 수 있고, 이에 대한 lock도 처리하게 된다.

- **pthread**

  ✓ pthread로 구현한 부분에 대해서 간략히 설명

  pthread_create() 함수로 client 접속마다 thread를 생성하게 되며, thread는 detach를 통해서 독립적으로 시행 후 reaping해주게 된다. Thread로 구현한 코드에서는 전체적으로 봤을때, buy,sell 과 show간에 reader – writer problem이 존재하고, 이밖에 update와 show가 같은 함수를 사용함에 따라 read -update 문제, read – read 문제 등이 존재한다.

  따라서 reader -writer problem을 위해서는 tree의 node마다 mutex 2개를 생성하여, first reader -writer에 대한 starvation문제를 해결하였다. 이로써 show를 수행하는 동안, 다른 client에서 buy를 수행할 때 해당 Node에 접근할 수 없으며, 또한 write starvation도 일어나지 않는다.

  전역변수로써 mutex도 하나 존재하는데, 이는 update와 read에서의 동일 함수 사용에 대한 문제를 해결하기 위함이다. 해당 함수에서는 static 변수를 index로써 사용하기 때문에 data 영역에 있는 해당 변수를 공유하여 문제가 생길 수 있다. 따라서 mut이라는 sem_t type의 변수를 사용하여, show에 대한 수행이 이루어지는동안은 update가 동시에 일어날 수 없도록 하였고, 반대의 경우에도 일어날 수 없도록 하여 safe thread를 구현하였다.

C. 개발 방법

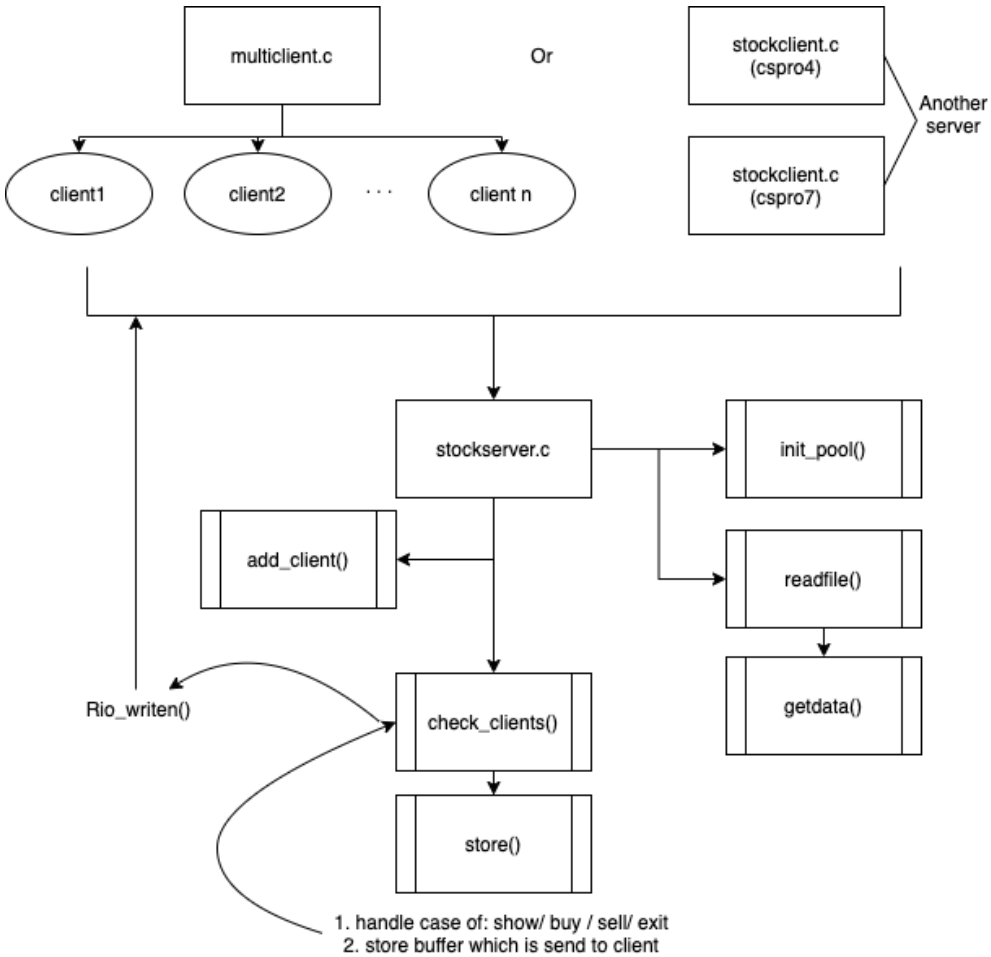
우선 event based concurrent server를 구현하기 위해서 pool이라는 구조체를 사용한다. 해당 구조체에는 clientfd 배열 및 clientrio, ready_set, read_set 변수 등이 존재하고, check_clients()함수 내애서 pool 구조체의 ready_set 변수 및 clientfd, clientrio 배열을 사용한다. 전체적으로 사용되는 함수는 init_pool, add_client, check_client이며, 각각의 함수에서 pool 구조체를 초기화하고, 연결된 client들에 대한 정보를 pool 구조체의 배열에 저장하고, 일괄적으로 Check_client 함수에서 수행하게 된다.

thread based에서와 동일하게 사용되는 부분은 주식 서버에 대한 동작을 수행하는 store 함수, getNode, addNode, paresline 등이 있다. Parseline은 buy 및 sell에 대한 명령어 tokenizing을 위한 부분이며, addNode와 readfile은 tree를 초반에 생성하기 위한 함수이고, getNode는 buy와 sell에대한 수행을 하기 위한 함수이다. Event-based에서는 check_client에서 rio_readlineb함수를 통해 전달된 buffer의 request내용을 store함수로 전달하고, thread-based 함수에서는 echo.c의 Echo 함수에서 rio_readlineb 함수를 통해 전달된 Buffer의 내용을 store 함수로 전달한다. Store 함수에서는 각 case에 따라 다른 함수를 호출하며, 버퍼에 client에 전달할 내용을 저장하며, 그 길이를 return한다. Check_client 함수 및 Echo 함수에서는 해당 정보를 받아, Rio_writen 함수를 통해서 Client로 보내게 된다. Stockclient.c 및 multiclient.c에 주어진 코드와 다르게 수정한 부분이 있는데, show 결과를 개행에 맞추어 출력하기 위해 일련의 과정을 fputc를 통해 수정하였다.

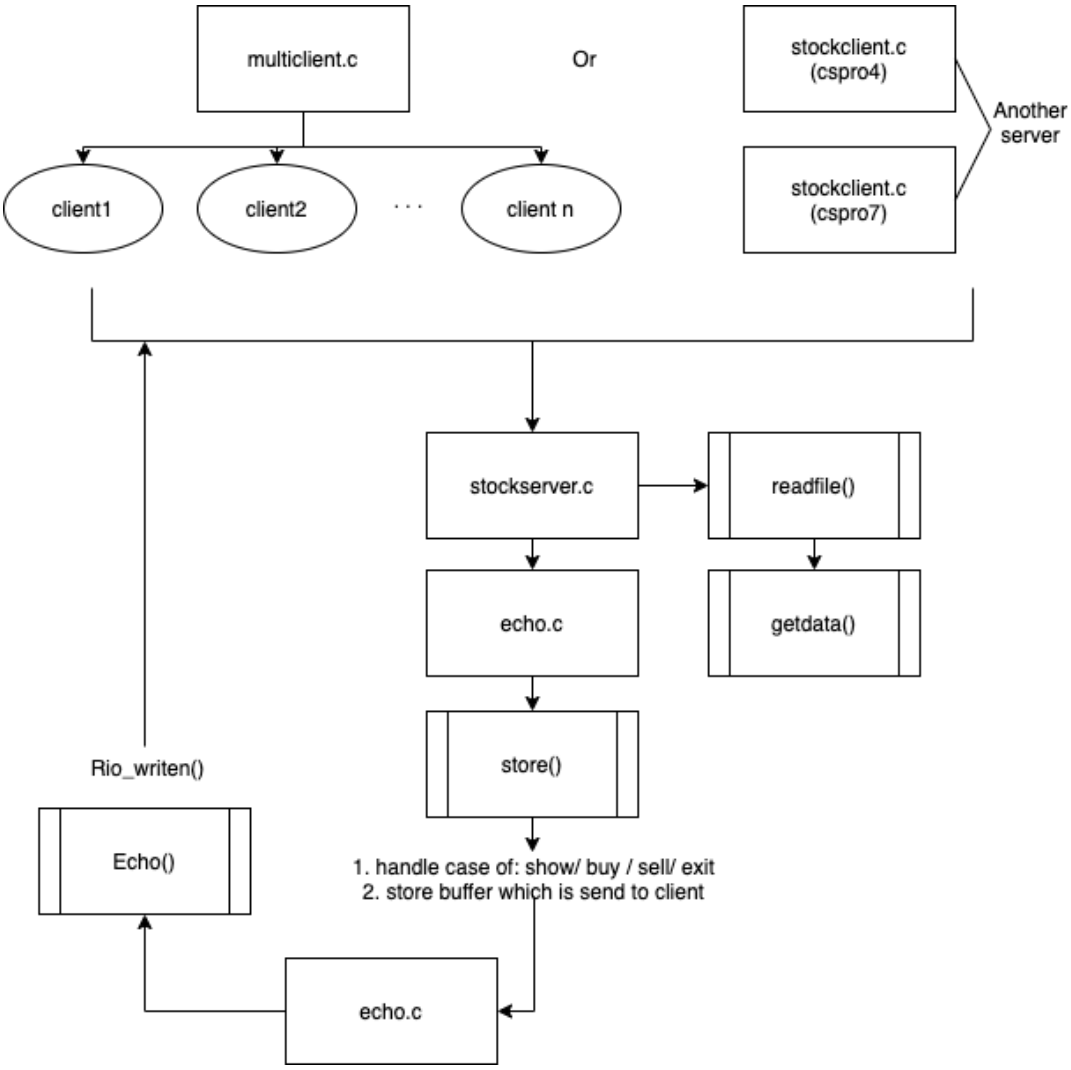
3. 구현 결과

A. Flow Chart

1. Select



2. pthread



B. 제작 내용

- II. B. 개발 내용의 실질적인 구현에 대해 코드 관점에서 작성.
- 개발상 발생한 문제나 이슈가 있으면 이를 간략히 설명하고 해결책에 대해 설명.

1. Select

처음 select를 구현할 때 단순히 multiplexing I/O 를 기반으로, select 함수를 직접 이용하여 구현하였는데, concurrent하게 동작하지 않아 pool 구조체를 사용하게 되었다. Polling을 통해서 kernel에게 지속적으로 준비 상태를 물어봐야 하기 때문에 이를 check_clients 함수에서 수행하게 되고, 이를 위한 이전의 과정들은 init_pool 함수 및 add_client함수에서 진행하게 된다. Init_pool은 client들과 연결

되기 이전에 pool 구조체를 초기화하며, add_client는 해당 pool구조체에 client에 대한 정보를 저장한다. Check_client 함수에서는 저장된 client들에 대해서 Rio_readlineb 함수를 통해서 Request를 받아와 store 함수를 거쳐서 Rio_writen 함수로 결과를 전달해 준다. 모든 수행이 끝난 client는 close() 함수를 통해서 connection이 끊기게 된다.

2. **pthread**

전체적인 흐름을 보면, thread_create를 통해서 연결된 client 마다의 수행을 처리할 worker thread를 생성하고, 이를 Thread 함수에서 detach 함수를 통해서 Reap 해주는 과정이다. Thread 함수에서는 echo 함수를 호출하고, Ehco 함수에서는 store 함수를 호출한다. Client request에 대해서 Store 함수에서 이루어지는 진행 과정은 Event에서와 비슷 하지만, 가장 큰 차이점은 safe thread를 만들기 위한 locking 과정이다.

처음엔 reader-writer 의 starvation 문제만 생각하여, tree의 node 마다 mutex 2개 및 readcount 변수를 만들어서 reader-writer 문제를 해결하였지만, 예상치 못한 문제는 update와 show의 수행에서 발생했다. Update와 show는 getdate라는 동일한 함수를 사용하는데, 해당 함수에서는 tree의 내용을 local로 선언한 stack 구조체에 저장하는 수행을 한다. Getdata함수에서는 static int 타입의 sidx 변수를 사용하는데, static 변수가 data 영역에 저장되고, 이에 대한 thread interleaving을 고려하지 못해서 error가 발생하였고, 이에 대한 해결책으로 전역변수로 mut 변수를 선언하여 binary semaphore로써 update와 show에 대해서 동일하게 P와 V 함수를 사용하여 lock 해주었다.

C. 시험 및 평가 내용

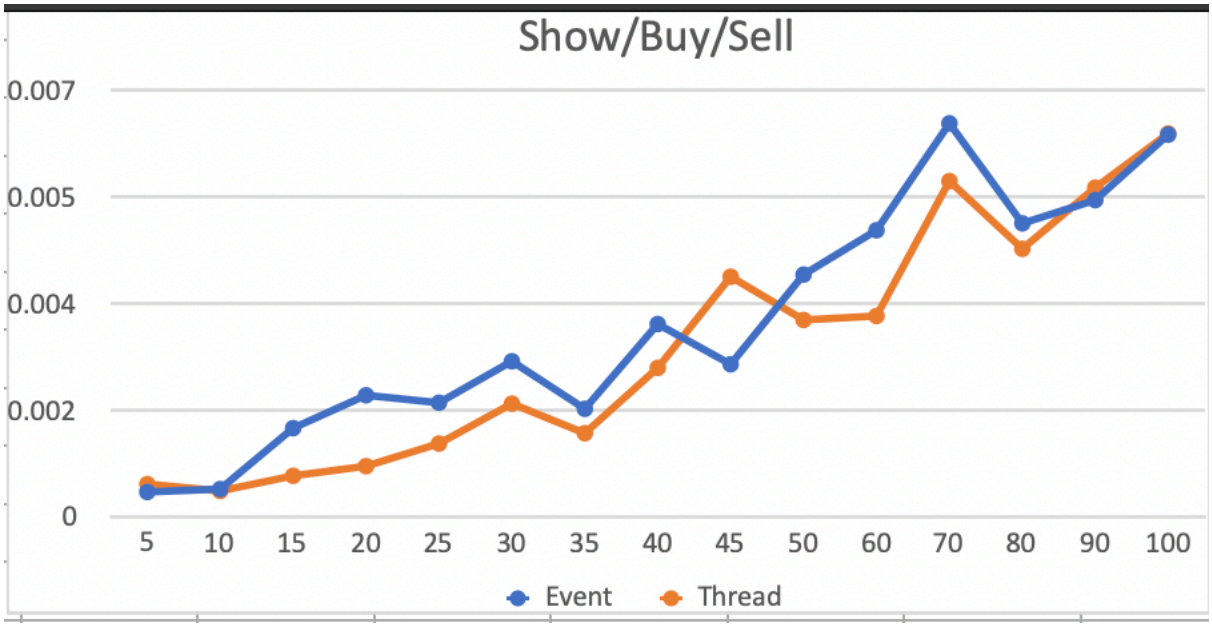
<예상>

1. Event based는 기본적으로 one logicla control flow 이고, fine – grained 가 힘들기 때문에, thread가 점차 증가하고, read – write의 요청이 반복될 수록 thread based 보다 수행이 떨어질 것으로 예상하였다.
2. Update에 대한 수행의 횟수를 어떻게 하느냐에 따라서 event와 thread에서 수행의 효율이 달라질 것으로 예상하였다.
3. 수업시간에 배운 내용을 토대로, event based의 overhead가 가장 낮고, thread-based가 중간정도의 overhead이기 때문에 전체적인 양상은 event-

based가 더 좋을 것으로 예상하였다.

<결과>

1. Show / Buy / Sell

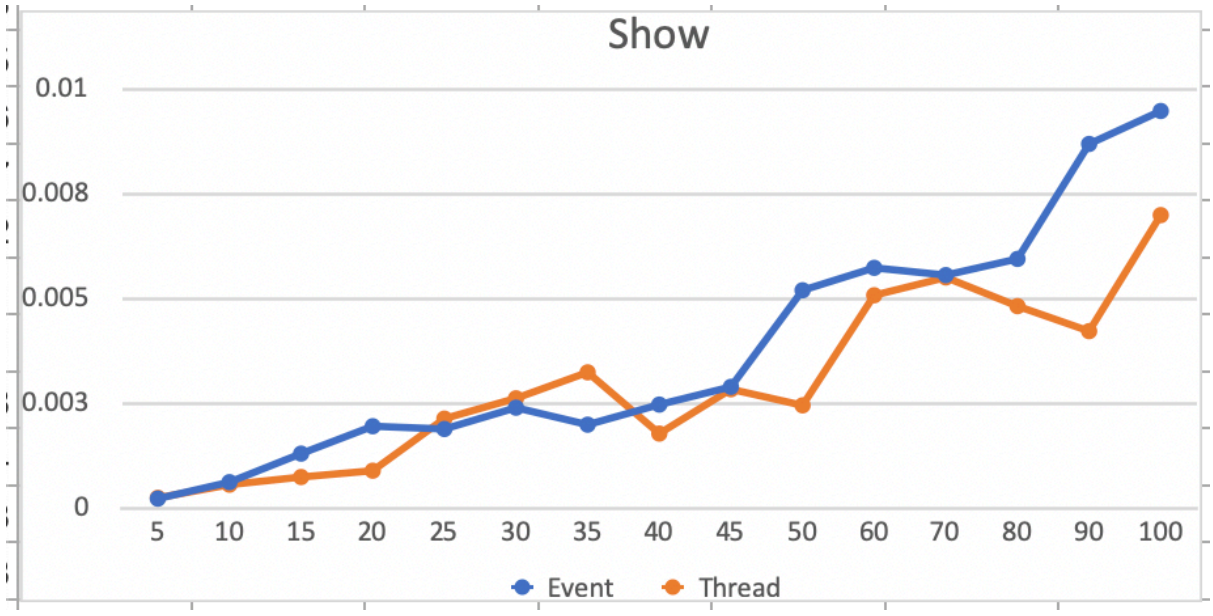


Show / buy / sell 에 대한 event와 thread에 대한 결과는 위와 같다 event의 수행 시간과 thread에서의 수행 시간은 거의 비슷하였지만, event에서 조금 더 오래 걸린것을 확인할 수 있다. 이론적으로는 event based에서의 수행이 overhead가 더 적지만, read 와 write를 복합적으로 수행하는 client의 수가 증가함에 따라서 thread에서의 효율이 점차 증가함을 확인하였다.

2. Buy / Sell

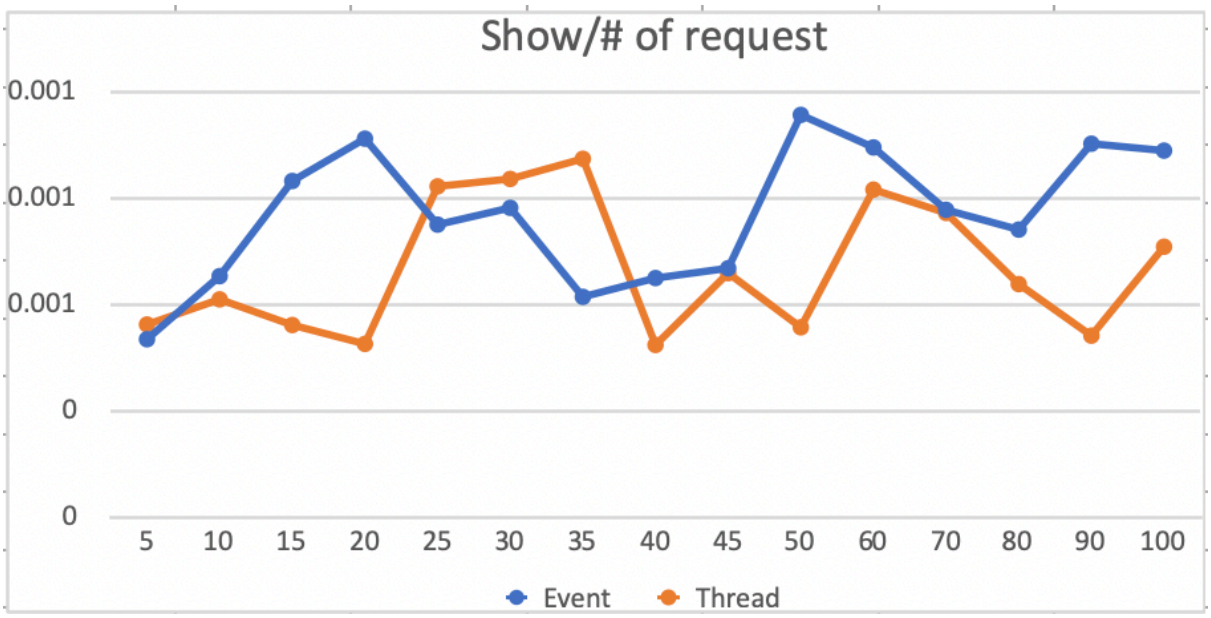
Write만 하는 경우인 buy/ sell 의 경우 thread와 event가 거의 비슷하였지만, 전체 적으로보았을 때는 Event에서의 효율이 더 좋음을 확인할 수 있다. Write는 시간이 가장 적게 걸리는 수행이므로 비교하기 어렵지만, 이론에 따라서 event-based에서의 효율이 조금더 좋게 나타남을 알 수 있다.

3. Show



Show 는 read만을 수행하는 경우이고, 이 경우는 시간이 상대적으로 다른 수행들보다 조금 더 오래 걸린다. 결과를 보면, thread에서의 수행이 좀 더 좋은 것을 볼 수있고, 예상과는 다르게 시간이 오래 걸리는 request에 대해서 thread의 효율이 더 좋은 것 으로 보인다. 이는 event가 thread에 비해서 concurrent 효율성이 떨어진다고 해석할 수 있을 것이다.

4. Show divide by (# of request x # of clients)



더 정확한 비교를 위해서 위 show (Read) 에서 수행한 결과에 대해서 client 개수 와 request 요청수를 곱한 값으로 show의 수행의 시간을 나눈 후 더 확연한 비교를 위해 100을 곱한 결과 값이다. 전체적으로 어떤 것이 효율이 더 좋다고 말할 수 없지만, client 수가 증가함에 따라 request 수가 비고적 많을 때 thread에서의 효율이 더 좋은 것으로 보아, 1의 예상과 같게 read 에서의 특정 request 에 대해서 concurrent 효율이 thread가 event보다 더 좋은 것을 볼 수 있다. 따라서 결과적으로는 overhead는 event가 더 적지만, Client 증가에 따른 효율성이 더 좋은 thread based cuncurrent server 구현이 더 이상적 형태라고 할 수 있다.