System Programming Project 2

담당 교수 : 김영재

이름 : 박성현

학번 : 20181632

1. 개발 목표

이번 프로젝트에서는 여러 client들의 동시 접속 및 서비스를 위한 Concurrent stock server을 구축한다. 주식 서버는 주식 정보를 저장하고 있고 여러 client들과 통신하여, 주식 정보 List, 판매, 구매의 동작을 수행한다. 주식 서버는 여러 개여 client들의 요청을 받을 수 있으며 각 client들은 여러 개의 요청을 보낼 수 있다. 주식 클라이언트는 server에 주식을 사고 팔 수 있으며 주식 가격과 재고의 조회를 요청 할 수 있다. 이번 프로젝트에서는 Event-based Concurrent Server과 Thread-based Concurrent server를 구축한다. 주식의 정보는 텍스트 파일의 형태로 저장이 되며 이진트리의 형태로 구현된다. 서버가 종료될 때, 변한 주식의 정보가 저장된다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

   - 첫번째 방법은 Event-based server이다. 이 방법은 하나의 process, 혹은 thread를 사용하여 여러 client들의 요청을 받는다. File descriptor들의 배열을 만들고 이 배열을 select함수를 통해 검사한다. Main 함수의 while loop에서 client들의 요청이 있을 때 마다, Accept함수를 통해 server와 연결시키고 각 client들에 대해 fd를 배정해 배열에 저장해 Select함수를 통해 변화가 감지된 fd를 찾아 check_clients 함수에서 요청을 실행한다.

2. Task 2: Thread-based Approach

   - 두번째는 Thread-based server이다. Process-based server와 비슷한 방법으로 수행된다. Process를 만드는 대신 thread를 여러 개 생성한다. Nthread 는 worker thread로 넉넉히 만들어주었다. Thread로 요청을 처리할 땐, 각 client의 connfd가 변화해선 안되며, 각 thread가 서로의 critical section에 들어가지 않게 주의해야한다. Connfd는 buffer에 따로 저장해 새로운 client가 연결되었을 때, 변하지 않도록 해주었고, 저장된 모든 자료에 접근, 혹은 수정시에 counting semaphore을 사용해 다른 thread의 접근을 차단했다.

3. Task 3: Performance Evaluation

   - 마지막 task는 성능 분석이다. 각 방법마다 client의 개수, 각 client가 보내는 요

청 수, 그리고 명령어의 종류를 변화시켜가며 분석을 진행했다. Client는 10개부터 100개까지 10개씩 변화를 주며 실행하였고, 각 client가 주는 요청 수도 10개부터 50개까지 변화를 주었다. 각 조건마다 실행시간을 측정해 동시 처리율을 계산했다. 동시처리율은 매 초마다 처리하는 order의 개수다. 총 order의 개수를 실행시간으로 나누어 구하였다. 또한 buy와 sell 명령어만 실행했을 때, show만 실행했을 때, 그리고 모든 명령어를 적절히 섞어 실행했을 때로 조건을 나누어 실험하였다.

B. 개발 내용

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

  ✓ Multi-client 요청에 따른 I/O Multiplexing 설명

  ◆ 이 방법으로 multi-client server을 구현할 때, 조심해야 할 점은 connfd의 변경이다. Server는 각각의 client들 마다 다른 connfd를 배정한다. 이 connfd는 새로운 client와 server가 연결되었을 경우, 바뀌게 된다. 그럼 바뀌기 전의 connfd는 사라지게 되며 그 connfd를 가진 client들의 요청은 모두 없어진다. 이를 해결하기 위해 pool이라는 구조체를 만들어 그 안에 connfd를 배열 형식으로 저장한다. 또한, ready_set이라는 fd_set형의 배열도 구조체 안에 포함되어 있다. 이 ready_set은 변화가 있는, 즉 요청이 발생한 connfd를 index로 하여 그 index의 값을 변화시킨다. select함수를 통해 이 배열을 모두 순회하며 변화가 있는 connfd의 개수를 nready로 뽑아낸다. 이후, check_client함수에서 이 nready만큼 변화된 readyset의 connfd에 해당된 client들의 요청을 수행한다.

  ✓ epoll과의 차이점 서술

  ◆ select함수는 readyset이 지속적으로 변하기 때문에 이 배열을 매번 끝까지 순회해야한다. 그러므로 O(n)의 시간이 필요하다. 또한 일반적으로 받을 수 있는 fd가 1024개로 제한되어있다. epoll함수는 이런 단점을 살짝 보완한 함수이다. epoll함수는 fd에 대한 순회를 운영체제에서 직접 담당한다. 또한 변화가 일어난 fd들을 따로 묶어서 관심대상에 대한 구조체를 만들어 관찰한다. epoll함수는 커널공간이 fd를 관리하여 select보다 더 빠른 처리가 가능하지만 프로세스가 지속적으로 커널에게 변화된 fd값을 요청해야하는 동기화 개념은 똑같다.

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

  ✓ Master Thread의 Connection 관리

◆ Master thread는 main thread로 NTHREADS만큼 worker 쓰레드를 만든다. Master thread는 fork 함수에서 parents의 형식으로 작동한다. Client들과의 connection을 만들어 connfd를 sbuf에 저장하고 생성된 worker thread들은 sbuf에 저장되어 있는 connfd를 가져와 order함수에서 명령어를 순차적으로 실행하고 connfd를 close한다. 또한 각 쓰레드는 Pthread_detach를 사용하여 thread함수가 종료되면 자동으로 소멸할 수 있게 하였다.

✓ Worker Thread Pool 관리하는 부분에 대해 서술

◆ Connfd는 main의 while loop에서 새로운 client와 연결이 있을 경우 생성되며 sbuf에 하나씩 저장된다. Connfd는 새로운 client가 연결될 때 마다 갱신되기 때문에 sbuf라는 buffer에다 저장했다. 이 buffer에 connfd의 값을 넣어줄 때도 semaphore을 사용한다. Sbuf의 값 역시 여러 thread가 접근하기에 mutex lock을 걸어준다. 또한, slot과 item에도 lock을 걸어서 하나씩 넣고 제거할 수 있게 하였다. Slot은 lock을 걸어 최대 slot의 개수가 넘어가지 않게 했다. Item은 connfd를 한 개씩 가져오기 위해 lock을 걸었다. Connfd를 insert 할 때, item은 0, slot은 n의 값을 가지고 시작한다. Insert 할 때, slot의 값을 1 줄여주며 빈 slot이 1 감소했다는 것을 알려준다. 그 후, mutex lock을 통해 값을 넣어준 후, item의 값을 증가시켜 buf에 들어있는 값이 존재한다고 알려준다. remove함수에서는 값이 존재하면 값을 가져온다. 가져온 후, slot의 개수는 다시 증가시키며 item의 개수는 감소시킨다. Slot과 item은 counting semaphore이기에 1보다 큰 값을 가질 수 있다. 하지만 mutex는 binary semaphore이기에 1아니면 0의 값을 가진다. 결국 한번에 데이터에 접근해서 값을 변경시키는 것은 하나의 thread이다. order함수 역시 buy와 show, 그리고 sell 할 때, 저장된 주식정보에 접근하기에 semaphore을 이용해 lock을 걸어 한번에 한 개의 thread만 접근할 수 있도록 처리하였다.

- **Task3 (Performance Evaluation)**

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

◆ 각 방법의 client의 개수와 각 client가 보내는 요청의 개수를 변화시키면서 동시 변화율을 측정하였다. 동시 변화율은 1초당 처리하는 요청의 개수로 (client*orders)/e_usec(elapsed time)으로 계산했다. 각 client들은 100개부터 10개까지 10개씩 줄였고 order역시 50개에서 10개씩 줄였다. 이 값들을 변화시키면서 측정한 이유는 가장 먼저 client들의 개수와 요청의 개수가 요청들을 처리하는 시간에 영향을 줄 것 같았기 때문이다. 또한 데이터에 접근해서 작성을 해야하는 buy-sell작업만 실행한 것과 접근만 하는 show만 실행한 것으로 분리해서 실행해 보았다.

✓ Configuration 변화에 따른 예상 결과 서술

◆ Client의 개수가 많을수록 동시처리율이 느릴 것으로 예상된다. Order 개수가 늘어날 수록 동시 처리율이 늘어날 것이다. client개수가 늘어나고 order 개수가 늘어날 수록 thread의 경우에는 lock을 잡는 overhead가 늘어나기에 더 오래 걸릴 것 같다. Event-based의 방법의 경우 크게 차이가 없을 것 같다. show함수가 buy-sell만 실행한 것보다 더 빨리 실행될 것이다. Show는 단지 읽는 곳에만 lock을 걸면 되지만 buy-sell은 읽고 쓰는 first-readers-writers problem을 해결해야하기에 시간이 더 걸릴 것 같다.

C. 개발 방법

- 가장 기본적으로 추가한 함수와 구조체는 이진트리의 구성을 위한 것이다. item이라는 구조체를 만들어 주식의 정보와 오른쪽과 왼쪽으로 내려가는 node pointer을 넣었다. 또한 insert를 이용해 node보다 큰 값은 오른쪽, 작은 값은 왼쪽에 넣었다. Crtl+c를 눌렀을 때, 종료될 수 있도록 signal handler함수도 작성했다. Exit과 Crtl+c를 눌렀을 때, 이진트리는 write_data함수를 통해 stock.txt파일에 정보를 작성하며 작성 후, 이진트리를 free_tree함수를 통해 free 시킨다.

- Event-Based server

✓ 가장 먼저 pool 구조체를 추가했다. pool구조체에는 connfd를 추가하기 위한 maxfd, maxi가 있고 connfd저장 배열인 clientfd, 그리고 fd의 변화를 보기위한 fd_set형의 read_set과 값 복사를 위한 ready_set이 있다. main함수에서 listenfd를 만든 후, init_pool을 통해 pool을 초기화 시켜준다. Read_set으로 계속 test할 경우 실행 중에 값이 변경 될 수 있으니 readyset에 복사해 사용한다. Client의 요청이 있을 경우, connection을 accept하며 그 client의 connfd를 add_client함수를 통해 pool에 넣어준다. 그 후, check_client함수를 통해 명령을 실행한다. Check_client함수는 pool에 값이 있고, readset의 bit이 0보다 크면, 즉 변경점이 있다면 그 connfd를 통해 client의 요청을 실행한다. Show 명령의 경우 show 함수를 통해 현재 tree에 저장되어 있는 모든 값들을 하나의 전역변수인 str문자열로 만들어준다. 이 문자열을 Rio_writen 함수를 통해 client로 전달한 후, 출력한다. Buy의 경우 buy함수를 통해 원하는 값을 tree를 순회하며 찾는다. 이진트리이기에 root부터 순회하며 큰값이면 오른쪽, 작은 값이면 왼쪽으로 내려가서 찾는다. 만약 원하는 수량이 재고보다 적을 경우, 에러를 출력한다. 이 외의 경우에는 수량에서 재고를 뺀 값으로 주식의 재고를 업데이트 해준다. Sell의 경우엔 sell함수를 통해 값을 찾은 후, 그 주식의 재고에 판매수량만큼 값을 더해 업데이트 해준다. Exit의 경우 write_data함수로 값을 sotck.txt.파일에 작성 후, free_tree를 통해 free 시킨 후 종료한다.

- Thread-Based server

  ✓ 이 방법의 경우, 가장 먼저 sbuf_t라는 구조체를 생성했다. 이 구조체는 connfd를 담기 위한 구조체이며 값을 저장하는 배열과 sem_t형식의 mutex가 3개가 있다. Mutex의 경우 하나의 thread만 접근 가능할 수 있게 하기 위해 binary semaphore을 사용했으며 0,1의 값을 가져 값을 변경할 때 사용한다. Slots는 남은 칸수, item은 들어간 item의 개수를 위한 것이다. 이 두 값들은 단지 값을 넣거나 뺄 수 있는 것을 확인하는 용도이기에 binary가 아닌 counting semaphore을 사용했다. 또한 item 구조체에 readcnt와 sem_t mutex, w를 추가했다. Readcnt는 접근한 thread의 수 이며 mutex는 값에 접근하기 위한, w는 값을 변경하기 위한 값이다. Main 함수에서 sbuu를 sbuf_init함수를 통해 초기화 시킨다. connfd배열의 크기는 1000으로 설정했다. 그리고 worker thread를 생성한다. NTHREADS는 100개로 충분히 넉넉하게 만들었다. While문 안에서 client가 connection요청을 받을 때 마다 connfd를 sbuf에 sbuf_insert 함수를 통해 넣어준다. Thread 함수에서는 Pthread_detach를 통해 종료 후, 알아서 free되도록 설정했다. Connfd를 sbuf_remove를 통해 하나씩 빼와 order함수에서 그 client가 요청한 명령들을 수행하도록 했다. 수행 후, 그 connfd는 Close될 수 있게 하였다. order함수는 check_client함수와 pool의 값들을 체크하는 기능을 빼면 동일하다. Sbuf_insert와 remove함수는 B의 개발내용의 worker thread pool관리에서 서술한 내용과 동일하게 작동한다. 가장 중요한 변경점은 show와 buy, sell이다. Show는 tree의 data에 접근하기에 함수가 호출되면 그 node의 mutex lock을 잡는다. Mutex와 w모두 잡아 다른 thread가 쓰거나 읽을 수 없다. 한 노드에 대해 문자열 생성이 완료되면 다시 lock을 풀어준다. 재귀함수를 통해 다음노드에 접근했을 때도 같은 방식으로 수행된다. Buy 함수의 경우, 원하는 값을 찾았을 때, mutex와 w의 lock을 걸어준다. 만약 재고가 부족할 경우, 값의 변경은 없기에 다시 lock을 풀어주며 return한다. 값을 찾았을 경우, reading에 대한 lock을 풀고 w에 대한 lock을 걸어 값을 변경시에 다른 thread가 값을 변경 할 수 없게 한다. Sell의 경우에는 값을 변경하는 경우만 있기에 w에 대한 lock만 걸어주었다.

- Task 3

  ✓ 값의 분석을 위해 multiclient에 변경점을 주었다.(제출파일은 변경하지 않은 초기 파일) 최대 client 개수를 100개로 설정하고 값을 10개씩 줄였다. 각 client의 order의 개수도 50개에서 10개씩 줄여가며 실험했다. 또한, fork를 통해 child를 생성하기전 gettimeofday를 통해 시작시간을 설정했다. 모든 child를 waitpid로 제거해주고 난 후, 다시 gettimeofday를 통해 마감시간을 구했다. 이 값들의 차이에 100만을 곱해 sec단위로 바꾸었다. 동시 처리율은 위에서 언급한바와 같이(client*orders)/e_usec로 계산했다. 이 결과값들은 10*5=50개이기 때문에 엑셀에 직접 쓰도록 했다. 또한 mod를 통해 명령어의 옵션을 normal, buy-sell, show만 로 지정했다.

3. 구현 결과

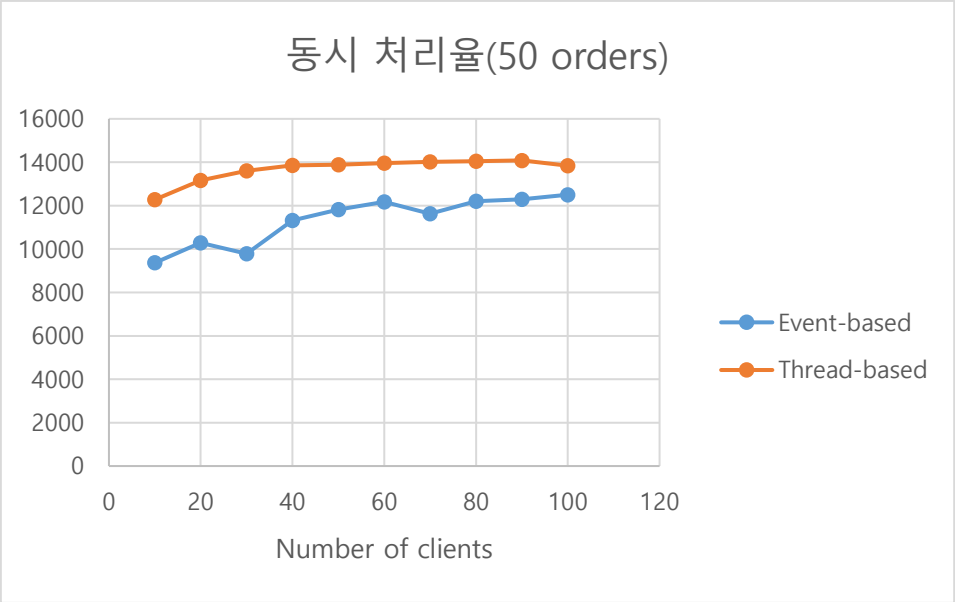
✓ Task1,2 모두 multiclient로 4개의 client를 가지고 실행했을 경우 에러없이 잘 실행되었다. 값이 올바르게 변경되는지 확인하기 위해 얼마나 사고 파는지 값을 출력해 본 결과 오류없이 잘 실행되었다.

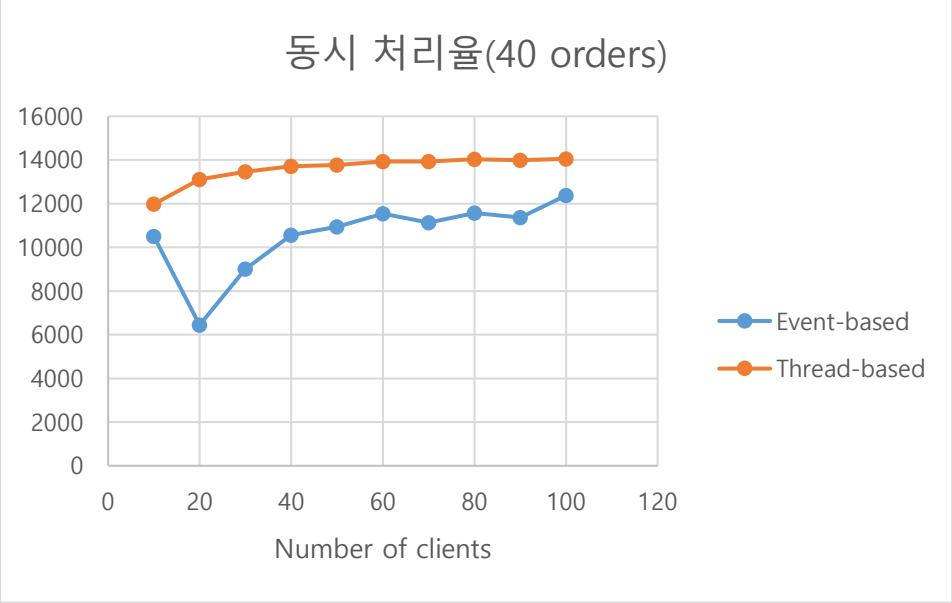
4. 성능 평가 결과 (Task 3)

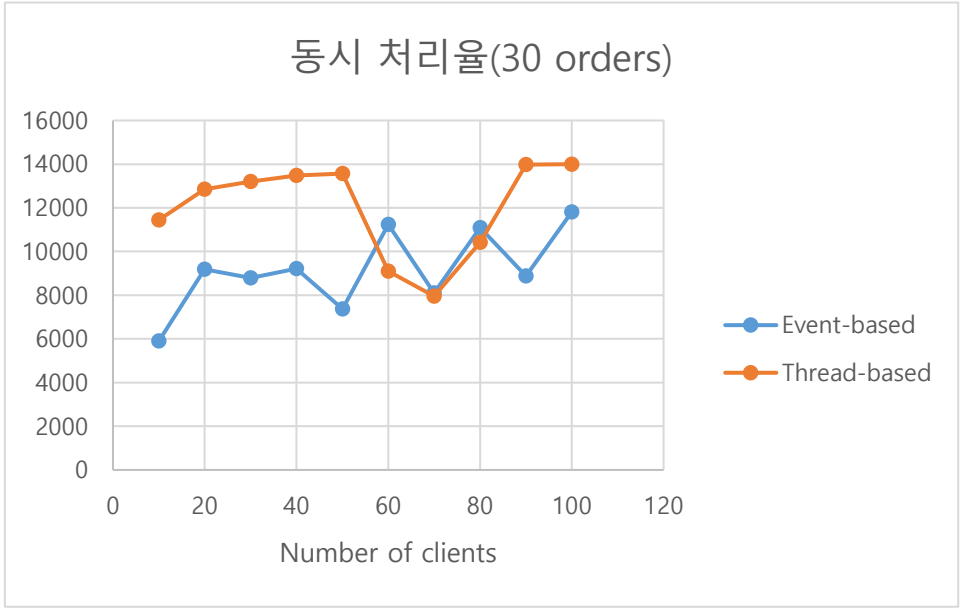
A. 확장성

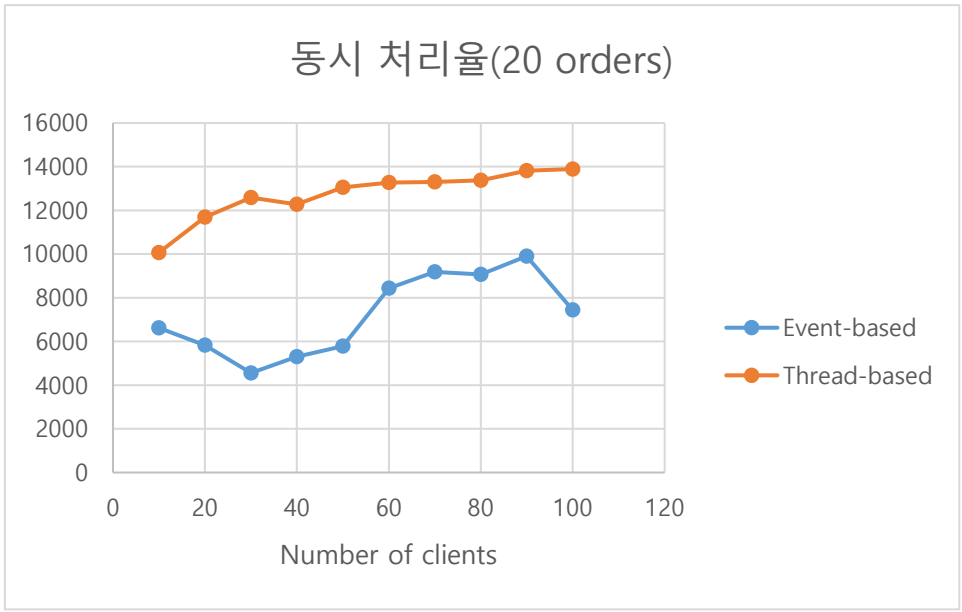
order per	client num	time	client num	time
50	100	12503.3	100	13840.2
50	90	12292.9	90	14084.2
50	80	12201.3	80	14044
50	70	11627	70	14022.3
50	60	12169.6	60	13962.2
50	50	11826.3	50	13887.4
50	40	11319.8	40	13858.6
50	30	9779.8	30	13599.5
50	20	10292.2	20	13161.5
50	10	9365.15	10	12276
40	100	12372.1	100	14053.3
40	90	11364.6	90	13993
40	80	11569.5	80	14034
40	70	11133.1	70	13937.8
40	60	11547.7	60	13929.6
40	50	10942.8	50	13767
40	40	10558.4	40	13703.6
40	30	9013.39	30	13468.1
40	20	6443.13	20	13106.5
40	10	10507.6	10	11980.7
30	100	11811	100	13997.8
30	90	8878.45	90	13984.5
30	80	11098.3	80	10413.7
30	70	8098.73	70	7956.21
30	60	11232.2	60	9104.34
30	50	7368.27	50	13569.5
30	40	9211.67	40	13479.7
30	30	8795.71	30	13202.4
30	20	9190.47	20	12843
30	10	5901.79	10	11448.9
20	100	7444	100	13895.5
20	90	9905.4	90	13820
20	80	9063.63	80	13381.5
20	70	9183.47	70	13298.5
20	60	8434.83	60	13267.5
20	50	5791.43	50	13047.3
20	40	5304.24	40	12276.3
20	30	4558.69	30	12582.1
20	20	5833.78	20	11694.3
20	10	6623.31	10	10070.2
10	100	6527.73	100	8452.69
10	90	6434.77	90	9156.28
10	80	6293.26	80	8567.26
10	70	5935.96	70	8210.78
10	60	5812.65	60	8965.58
10	50	5966.12	50	8756.84
10	40	6155.32	40	8211.09
10	30	6223.73	30	8543.5
10	20	5929.72	20	8092.78
10	10	5367.86	10	6391.08

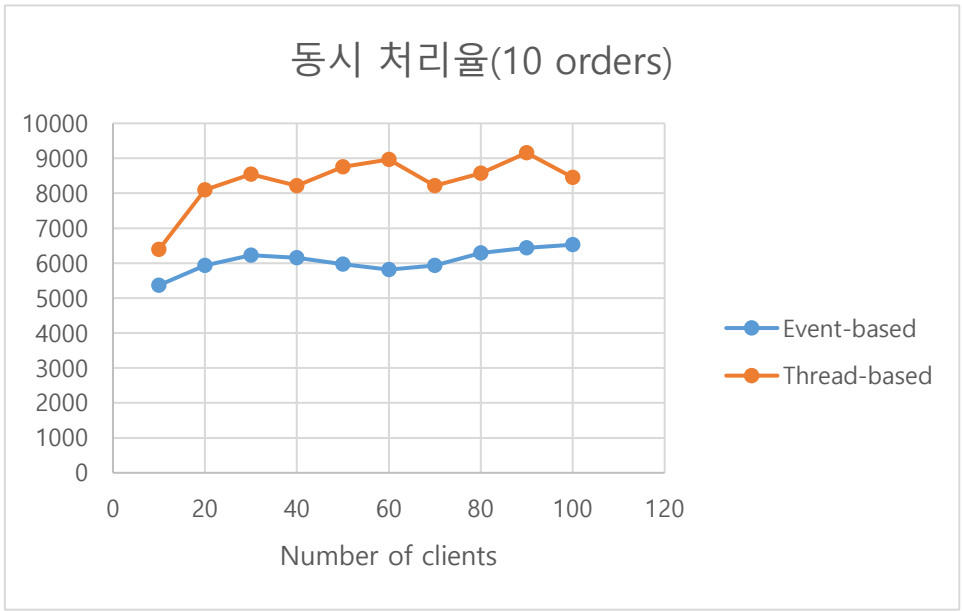
왼쪽이 event-based이고 오른쪽이 thread-based에 대한 자료이다.











thread-base의 경우가 일반적인 경우 event-based 경우보다 동시 처리율이 높다. 흥미로운 점은 order이 줄어들수록 동시 처리율이 줄어든다는 것이다. 50개에서는 thread의 경우 14000개를 1초에 처리하지만 10개의 경우 평균 9000개를 처리한다. Order의 개수가 줄어드는 것은 처리할 총 order의 개수가 적다는 것인데, 이에 대한 이유를 찾지 못하였다. 총 처리해야하는 개수가 줄어들면 당연히 overhead가 덜 걸려 빨리 처리 될 것이라고 생각했다. 또한 thread의 경우 client의 개수가 늘어날수록 동시 처리율이 늘어났다. Thread의 개수가 늘어날수록 동시 처리율이 늘어나는 것은 worker thread를 100개 만들었고 최대 client가 100개이기에 모든 thread가 working중이라 메모리와 필요없는 context switch 없이 요청을 처리하기 때문일 것 같다. 남는 thread가 있을 경우, 그 thread는 아무일도 하지 않고 있으며 그 thread로 context switching이 일어나는 것 자체가 동시 처리율에 영향을 미치는 것 같다. 하지만 event-base의 server가 더 낮은 동시 처리율을 보여주었다. 이는 select함수의 한계점때문인 것 같다. 아무리 thread-base의 semaphore가 lock을 잡고 푸는데 overhead가 걸리더라도, event-based의 select함수는 while문이 한번 돌때마다 모든 배열을 순회해야하기 때문에 시간이 더 오래 걸리는 것 같다. 동시 처리율은 걸린 시간에 반비례한다. 이로부터 알

수 있는 것은 event-based가 더 시간이 걸린다는 점이고 이는 위의 결과와 분석과 일치한다.

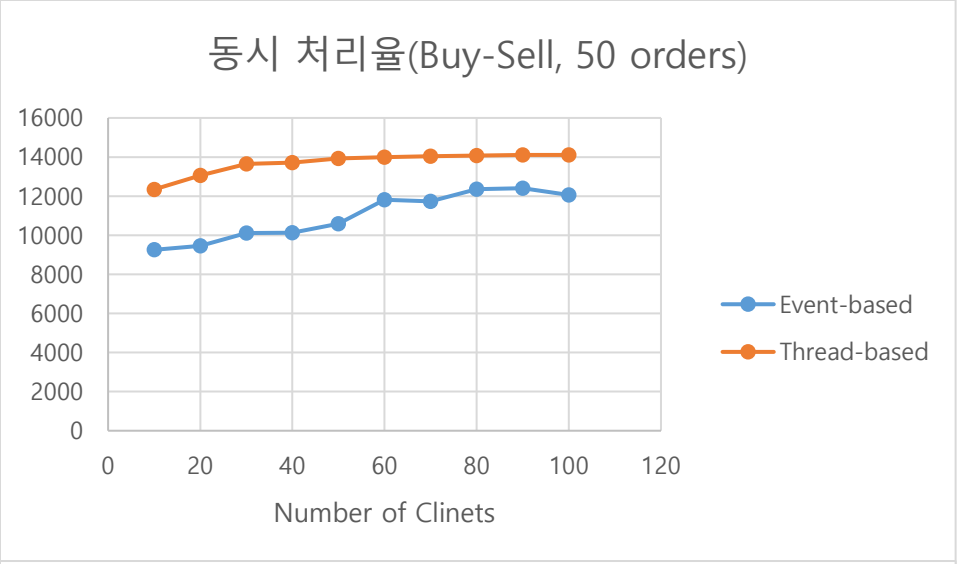
B. 워크로드에 따른 분석

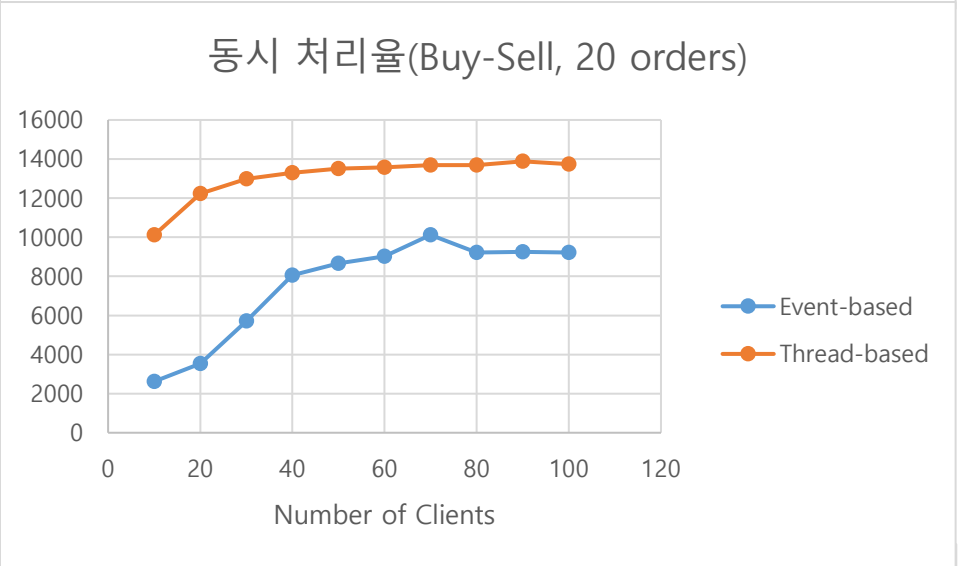
Buy/Sell

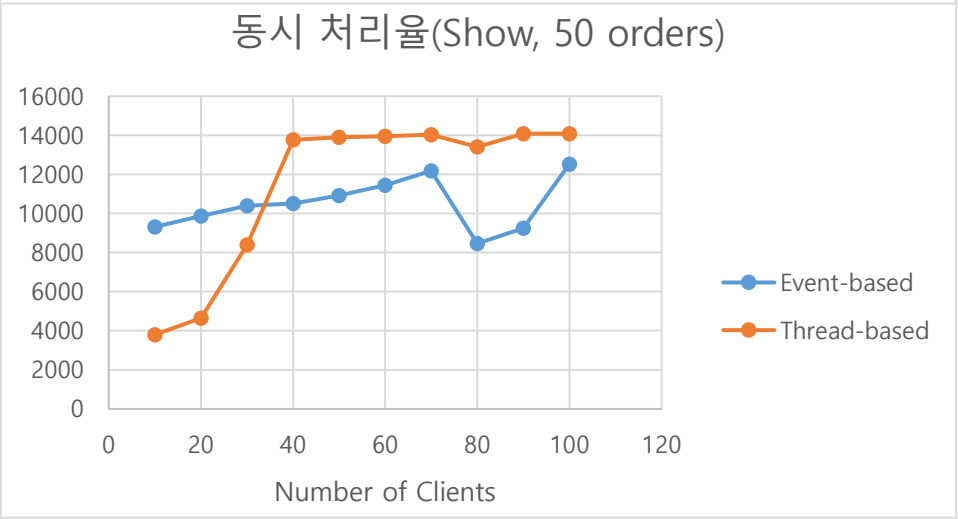
order per	client num	time	client num	time
50	100	12056.8	100	14107
50	90	12405.1	90	14104.7
50	80	12358.3	80	14069.7
50	70	11734.2	70	14040
50	60	11819.8	60	13992.2
50	50	10584.4	50	13930.4
50	40	10125.1	40	13721.5
50	30	10105.7	30	13648.5
50	20	9458	20	13058.5
50	10	9252.85	10	12343.2
40	100	12042.7	100	14021.4
40	90	12179.6	90	14074.6
40	80	12308.3	80	14040.7
40	70	11820.9	70	13995.8
40	60	11337.3	60	9309.54
40	50	11078	50	7107.69
40	40	11276.6	40	6419.42
40	30	9989	30	5418.5
40	20	9533.41	20	6959.83
40	10	9400.13	10	12028.4
30	100	11758.9	100	14033.4
30	90	11152.3	90	13968.4
30	80	11689.8	80	13969.6
30	70	10985.1	70	13921.1
30	60	10295.4	60	13832.9
30	50	9867.58	50	13740.7
30	40	9407.77	40	13612.1
30	30	5152.27	30	13377.4
30	20	3929.06	20	12651.8
30	10	3064.66	10	11547.7
20	100	9222.11	100	13746.9
20	90	9244.07	90	13883.3
20	80	9215.22	80	13695
20	70	10118.5	70	13692.5
20	60	9022.15	60	13580.4
20	50	8658.82	50	13513.6
20	40	8060.68	40	13288.6
20	30	5713.34	30	12982.8
20	20	3537.56	20	12229.7
20	10	2620.54	10	10112
10	100	5884.62	100	8324.72
10	90	6165.97	90	8534.49
10	80	6489.28	80	8630.23
10	70	7879.44	70	8774.39
10	60	6143.46	60	8748.86
10	50	6106.54	50	8901.33
10	40	5674.63	40	7468.93
10	30	5973.95	30	6930.62
10	20	5674.82	20	7269.61
10	10	5522.6	10	5780.55

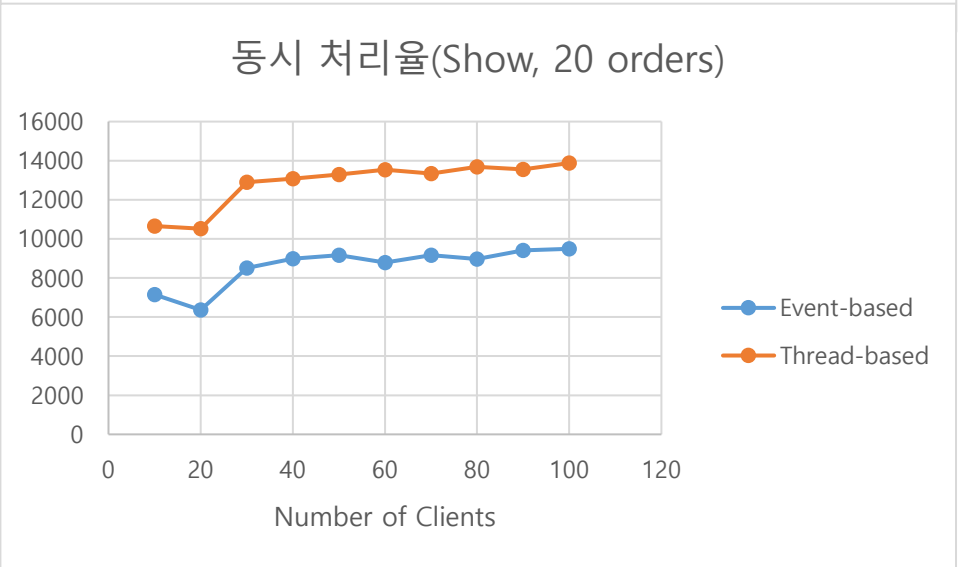
Show

order per	client num	time	client num	time
50	100	12531	100	14071.5
50	90	9251.22	90	14082.4
50	80	8458.93	80	13410.7
50	70	12177.9	70	14028.1
50	60	11449.8	60	13952.8
50	50	10921.5	50	13896.7
50	40	10515.1	40	13773.7
50	30	10397.5	30	8399.02
50	20	9874.65	20	4638.43
50	10	9316.3	10	3798.8
40	100	11522.2	100	12899.1
40	90	11226.2	90	14019.9
40	80	11244.3	80	14028.5
40	70	11000.5	70	13281.4
40	60	8314.28	60	13900.8
40	50	6063.31	50	13863
40	40	7804.27	40	13715
40	30	10061.5	30	13020.1
40	20	9570.96	20	12621.3
40	10	8320.82	10	11239.6
30	100	11610.3	100	13976.1
30	90	10796.8	90	13973.5
30	80	10405.5	80	13927.1
30	70	10517.6	70	13899
30	60	10538	60	13751.2
30	50	10249.4	50	13689.3
30	40	9483.69	40	7992.6
30	30	8782.29	30	5452.5
30	20	8621.73	20	4415.23
30	10	7420.75	10	3417.21
20	100	9490.55	100	13878.3
20	90	9407.08	90	13562
20	80	8966.09	80	13689.5
20	70	9165.76	70	13343
20	60	8788.93	60	13541.1
20	50	9164.59	50	13297.2
20	40	8982.71	40	13074.9
20	30	8512.21	30	12894.6
20	20	6363.19	20	10520.6
20	10	7147.45	10	10651.7
10	100	6022.83	100	8652.36
10	90	6223.02	90	8378.91
10	80	5780.05	80	8747.53
10	70	5819.03	70	8223.63
10	60	5992.51	60	8093.78
10	50	5706.13	50	8191.54
10	40	5785.58	40	10391.8
10	30	5519.62	30	9009.44
10	20	5587.28	20	8389.19
10	10	4920.24	10	7426









50개와 20개의 order의 그래프만 가져왔다. 다른 그래프도 비슷한 성향을 보인다

*위의 그래프가 buy와 sell만 실행한 경우이고 아래가 show만 처리한 경우이다. 두 경우 다 50개, 20개의 order을 받았다. 아래 show만 있는 경우 outlier가 조금 있다. 하지만 평균적으로 둘다 초당 event는 50개일때 12000개, 20개일 때 10000개이며 thread는 14000개이다. 두 실험의 다른점은 buy/sell은 writer's lock을 추가로 걸어준다는 점이다. Writer's lock이 걸리는 시간에 크게 영향을 미치지 않는다는 것을 알 수 있다.

결론적으로, event-based가 더욱 시간이 많이 걸린 다는 것을 알 수 있다. Thread가 lock을 거는데 overhead가 걸리지만 event 의 select 함수가 더 시간이 많이 걸리는 것 같다. 하지만 메모리 관점에서 thread-based는 thread를 생성해야하기에 메모리 소모가 큰 반면, event-based는 하나의 process에서 하나의 thread로 실행되기 때문에 메모리 소모가 thread-based보다 훨씬 적을 것이라 생각한다. 두 방법을 적절히 섞어 사용하면 보다 효율적인 방법으로 요청을 처리할 수 있을 것이다.