System Programming Project 3

담당 교수 : 김영재 교수님

이름 : 이 상 연

학번 :20201617

1. **개발 목표**

본 프로젝트의 목적은 네트워크 프로그래밍 이론을 실제로 적용해, 다수의 클라이언트가 동시에 접속하고 요청을 처리할 수 있는 동시 처리 주식 서버를 구현·분석하는 것이다. 클라이언트는 서버에 "주식 조회(show)", "매수(buy)", "매도(sell)" 명령을 전송하고, 서버는 이를 파싱해서 해당하는 동작을 수행하고, 적절한 응답을 클라이언트에게 돌려준다.

구체적으로는 아래 두 가지 방식으로 서버를 구현하고, 성능을 비교·분석한다.

- Task 1: Event-based Approach
  - 단일 프로세스·단일 스레드 내에서 select() 기반의 I/O 멀티플렉싱을 사용
  - 커넥션 풀에 등록된 모든 소켓을 순회하며, 준비된 파일 디스크립터만 처리
  - Non-blocking 방식의 이벤트 순차 처리 구조
- Task 2: Thread-based Approach
  - 메인 스레드가 워커 스레드 풀을 생성하고, 각 워커가 클라이언트 세션 전체(요청 수신 -> 응답 전송)를 담당
  - Blocking I/O를 활용해 실제 멀티코어 환경에서 병렬 처리 실현
  - 세마포어·뮤텍스를 이용한 동기화

서버 실행은 "./stockserver <Port>" 명령으로 시작하며, 실행 시점에 같은 디렉토리의 "stock.txt" 파일에서 주식 데이터를 읽어서 메모리에 로드한다. 주식 정보를 level-order 이진 트리 구조로 관리하여, 빠른 탐색·삽입과 동시성 제어를 지원한다.

클라이언트는 "./stockclient <IP> <Port>"로 서버에 접속 후, 표준 입력으로 명령어를 입력하면, 해당 요청이 소켓을 통해 전달되고, 서버는 내부 데이터 구조를 갱신하거나 조회한 뒤 응답을 다시 전송한다.

위의 두 가지 접근 방식을 구현하고, cspro 멀티코어 환경에서 동시 접속수, 처리량, 응답 지연 등을 측정·분석하여, 각 설계가 갖는 장단점을 심도 있게 이해하는 것이 최종적인 목표이다.

2. **개발 범위 및 내용**

A. 개발 범위

1. Task 1: Event-driven Approach

Task 1 에서 Event-driven 서버를 구현하면, 서버는 하나의 메인 스레드만으로 select() 기반 방식을 사용해 클라이언트의 show, buy, sell, exit 명령을 처리할 수 있다. 서버를 시작하면 같은 디렉토리 내의 "stock.txt" 파일에서 주식 데이터를 로드하여 level-ordered 이진 트리 자료구조로 메모리 내에 보관하고, 무한 루프 내에서 select() 호출을 통해 입출력 준비가 완료된 파일 디스크립터만 검출한다. 신규 연결이 감지되면 accept()와 add_client()로 커넥션 풀에 추가하고, 기존 연결에 데이터가 수신되면 Rio_readlineb()로 요청 라인을 읽어 적절하게 파싱한 후, 트리 탐색 또는 노드 갱신 작업을 수행한다. 처리에 대한 결과는 Rio_writen()으로 클라이언트에게 응답하며, 논블로킹 이벤트 루프가 반복 실행되므로 다수의 클라이언트를 semi-concurrent하게 관리할 수 있다.

2. Task 2: Thread-based Approach

Task 2 에서는 pthread 라이브러리를 이용해 고정 수의 워커 스레드 풀을 구성하고, 워커 스레드들이 진정한 병렬 처리로 클라이언트 요청을 처리하도록 설계했다. 서버 초기화 시에 sbuf_init()로 연결 큐를 준비하고, 메인 스레드는 Accept()로 받아낸 소켓을 sbuf_insert()로 큐에 저장한다. 각 워커 스레드는 sbuf_remove()로 큐에서 소켓을 꺼내 Rio_readinitb()와 Rio_readlineb()로 요청을 수신한 뒤, 적절하게 파싱한다. 그 후 요청에 해당하는 동작을 수행하고, 이 때 특히 여러 클라이언트가 같은 노드를 변경하는 동작을 수행할 때 동기화를 위해서 mutex 변수를 이용했다.

3. Task 3: Performance Evaluation

Event-driven 서버와 Thread-based 서버의 성능을 비교·분석한다. <time.h>의 struct timespec 과 clock_gettime(CLOCK_MONOTONIC, ...)을 사용하여 높은 해상도로 시간 측정을 수행한다. 클라이언트가 요청을 보내는 시점에 clock_gettime()으로 start를 찍고, 더 이상 연결된 클라이언트가 없을 때마다 end를 찍은뒤 elapsed time 을 계산해서 출력한다. 가장 마지막에 찍힌 시간이 구하고자 하는 elapsed time 이다. 측정 실험은 클라이언트 수(예: 10, 100, 500, 1000)을 달리하

며 각각의 평균·최대 지연과 처리량을 기록한다. 또한 이를 기반으로 처리율(Throughput)을 구해 비교·분석한다.

B. 개발 내용

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

✓ Multi-client 요청에 따른 I/O Multiplexing 설명

I/O 멀티플렉싱은 이벤트 기반 서버에서 하나의 스레드로 여러 클라이언트의 입출력 요청을 동시에 처리할 수 있도록 돕는 기법이다. '멀티플렉싱(Multiplexing)'이란 여러 신호 중 하나를 선택해 분배하는 다중화 기술을 뜻하며, 서버에서는 커넥션 풀(Connection Pool)을 통해 활성화된 소켓을 관리한다.

1. select() 호출

메인 스레드는 select() 시스템 콜을 이용해 준비된 파일 디스크립터(fd)가 나타날 때까지 블록 상태로 대기한다.

2. FD_ISSET 검사

select() 에서 리턴하면 FD_ISSET(listenfd, &pool.ready_set)으로 리스닝 소켓에 새로운 연결 요청이 감지되었는지 확인한다.

3. 새 연결 수락

새로운 연결 요청이 존재하면 accept()로 해당 클라이언트 연결을 수락하고, add_client()로 해당 소켓을 커넥션 풀에 추가한다.

4. 기존 클라이언트 요청 처리

이후 check_clients()를 호출해 커넥션 풀에 등록된 모든 소켓을 순회하면서, 읽기 준비가 된 소켓이 있으면 그 소켓의 요청을 처리한다.

이처럼 I/O 멀티플렉싱을 통해 서버는 멀티스레드 없이도 높은 동시성을 효율적으로 달성할 수 있다.

✓ epoll과의 차이점 서술

select 함수는 사용자 영역에서 파일 디스크립터 집합(fd_set)을 비트맵 형태로 관리하며, 매 I/O 멀티플렉싱 호출 시 최대 번호(maxfd)까지 모든 비트맵 비트를 순차적으로 확인한다. 이로 인해 감시 대상 소켓 수가 늘어날수록 검사 비용이 선형적으로 증가하고, 특히 수천~수만 개의 소켓을 다룰 때에는 전체 순회 오버헤드가 성능 저하의 주된 원인이 될 수도 있다.

반면 epoll은 커널 내부에서 대상 소켓을 등록(epoll_ctl)한 뒤, 실제 이벤트가 발생한 소켓 정보만을 사용자 영역으로 전달(epoll_wait)한다. 따라서 등록된 소켓을 매번 순회하지 않고, 준비된(ready) 소켓 수(k)에 비례하는 비용만 발생하기 때문에, 대규모 동시 접속 처리에 월등히 유리하다. 또한 epoll은 엣지 트리거(edge-triggered) 모드와 레벨 트리거(level-triggered) 모드를 지원하여, 이벤트 발생 패턴에 맞춰 호율적인 I/O 처리가 가능하도록 설계되어 있다.

요약하면, select는 단순하고 호환성이 높으나 소켓 수 증가에 따른 선형 스캔 비용과 디스크립터 수 제한이 문제인 반면, epoll은 커널 수준에서 이벤트를 관리하여 고도로 확장 가능한 비동기 I/O를 제공하며, 특히 수천~수만 개의 동시 연결을 처리할 때 select 보다 더 뛰어난 성능을 발휘한다.

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

  ✓ Master Thread의 Connection 관리

  Thread-based 서버에서는 Master Thread는 클라이언트 연결을 관리한다. 서버가 실행되면 미리 정의된 개수(NTHREADS)의 Worker Thread들을 생성한 뒤, 무한루프에서 Accept()를 호출하여 클라이언트 연결을 기다린다. 이때 Accept()는 클라이언트의 연결 요청이 들어올 때까지 block된 상태로 대기한다. 클라이언트의 연결 요청이 도착하면, 해당 클라이언트와 연결된 소켓의 파일 디스크립터를 sbuf_insert() 함수를 이용해 shared buffer에 삽입하고, 다음 연결 요청을 계속해서 기다린다.

  ✓ Worker Thread Pool 관리하는 부분에 대해 서술

  앞서 설명한 것과 같이 서버가 초기화 될 때 Master Thread는 미리 정의된 개수(NTHREADS) 만큼의 Worker Thread를 생성한다. Worker Thread들을 생성 직후 각자 pthread_detach()를 호출하여, 작업이 종료되었을 때 Master Thread가 별도의 reaping 과정을 수행할 필요 없이 자동으로 자원을 반환(detach)하도록 설정

된다.

각 Worker Thread는 무한루프를 돌며, sbuf_remove() 함수를 호출하여 shared buffer에서 클라이언트와 연결된 소켓의 파일 디스크립터를 하나씩 꺼내 작업을 처리한다. 결과적으로, Thread-based 서버의 구조는 Master Thread가 연결 요청을 받아 shared buffer에 파일 디스크립터를 삽입하면, Worker Thread들이 이를 즉시 꺼내어 병렬적으로 클라이언트의 요청을 처리하는 방식으로 concurrent하게 동작한다.

sbuf_insert() 및 sbuf_remove() 함수 내부에서는 mutex lock과 semaphore를 이용해서 critical section을 보호한다. 이를 통해 Master Thread와 Worker Thread 사이에 shared buffer 접근과 관련된 race condition을 방지하고 안전한 데이터 전달을 보장한다.

- **Task3 (Performance Evaluation)**

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

1. 확장성(Scalability): 클라이언트 수 변화에 따른 동시 처리율 비교

Event-based 서버와 Thread-based 서버의 성능 차이를 평가하기 위해 설정하였다. 동시 접속한 클라이언트의 수가 증가함에 따라, 두 서버의 동시 처리율(Throughput) 변화를 분석한다. 이를 통해 각 설계 방식의 확장성을 평가할 수 있다.

2. 워크로드(Workload): 클라이언트 요청 유형(buy, show, sell)에 따른 동시 처리율 비교

Thread-based 서버에서 semaphore와 mutex를 이용한 동기화로 인해 요청 유형에 따른 성능 차이가 발생할 수 있다. 특히 read 요청(show)와 write 요청(buy, sell)은 동기화 방식에 따라 병렬 처리 가능 여부가 다르다. 따라서 클라이언트 수가 증가할 때 아래 세 가지 상황으로 나누어 동시 처리율을 측정하고 비교한다.

- ◆ Read 요청(show)만 수행하는 경우
- ◆ Write 요청(buy, sell)만 수행하는 경우
- ◆ Read 요청과 Write 요청이 혼합된 경우(일반적인 경우)

✓ Configuration 변화에 따른 예상 결과 서술

1. 확장성: 클라이언트 수 변화에 따른 동시 처리율 비교

적은 수의 클라이언트가 동시에 접속했을 때는 Event-based 방식과 Thread-based 방식의 성능 차이가 크지 않을 것으로 예상된다. 그러나 클라이언트 수가 많아질수록 병렬 처리 능력을 갖춘 Thread-based 방식이 Event-based 방식보다 같은 수의 요청을 더 빠르게 처리할 것으로 예상된다.

2. 워크로드

Thread-based 서버는 특정 노드를 변경하는 write 요청(buy, sell)을 수행할 때 semaphore를 통해 배타적 접근을 보장하므로, 동시에 여러 쓰레드가 같은 node에 대한 write 작업을 할 수 없다. 반면, 단순 조회를 수행하는 read 요청(show)의 경우에는 여러 쓰레드가 동시에 작업을 처리할 수 있으므로, 병렬성이 높아 처리 속도가 더 빠를 것으로 예상된다.

C. 개발 방법

Task 1

1. 소켓 관리

pool 구조체 내에 클라이언트 연결 상태를 관리하기 위해 clientfd 배열과 clientrio 배열을 활용한다. 새로운 클라이언트 연결을 수락하면 add_client() 함수가 이를 처리하고, 연결이 끊어지면 close_client() 함수가 호출된다.

```
/* server pool */
typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;                  /* Largest descriptor in read_set */
    fd_set read_set;             /* Set of all active descriptors */
    fd_set ready_set;              /* Subset of descriptors ready for
reading */
    int nready;                   /* Number of ready descriptors from
select */
    int maxi;                   /* High water index into client array
*/
    int clientfd[FD_SETSIZE];      /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE];    /* Set of active read buffers */
} pool;
```

2. I/O Multiplexing 구현

main 함수의 반복문에서 select() 시스템 콜을 사용하여 준비된 파일 디스크립터를 감지하고 처리할 수 있도록 유지한다. 소켓 준비 여부를 확인하기 위해 FD_ISSET()을 이용한다.

3. 클라이언트 요청 처리

클라이언트 요청을 처리하는 handle_request() 함수는 show, buy, sell, exit 명령을 처리하도록 구현되어 있으며, 향후 명령어 추가 시에는 이 함수 내의 명령 파싱 로직의 확장이 필요한다.

4. 성능 측정 코드 추가

성능 측정을 위해 <time.h>의 timespec 구조체를 활용하여 첫 번째 클라이언트 연결 부터 마지막 요청 처리 까지의 시간을 측정했다.

Task 2

1. 스레드 관리

메인 스레드가 고정된 개수(NTHREADS)만큼의 워커 스레드를 생성한다. 생성된 워커 스레드는 각자 독립적으로 클라이언트 요청 처리를 수행하며, pthread_detach()를 이용해 메인 스레드가 별도의 reaping 작업을 하지 않아도 되도록 설정한다.

2. 클라이언트 연결 관리

메인 스레드는 클라이언트의 연결 요청을 accept 한 뒤, 이 요청의 소켓 파일 디스크립터를 공유 버퍼(sbuf_t)에 삽입한다.

```
/* sbuf_t: Bounded buffer used by the SBUF package */
typedef struct {
    int *buf;              /* Buffer array */
    int n;                /* Maximum number of slots */
    int front;             /* buf[(front+1)%n] is first item */
    int rear;              /* buf[rear%n] is last item */
    sem_t mutex;           /* Protects accesses to buf */
    sem_t slots;           /* Counts available slots */
    sem_t items;           /* Counts available items */
```

```
} sbuf_t;
```

3. 공유 버퍼 동기화

공유 버퍼(sbuf_t)를 통해 메인 스레드와 워커 스레드 사이에서 클라이언트 소켓 디스크립터를 주고받는다. semaphore(mutex, slotx, items)를 사용하여 critical section을 보호하고 race condition을 방지한다.

4. 클라이언트 요청 처리

각 워커 스레드는 handle_request()를 호출하여 클라이언트의 요청(show, buy, sell, exit)을 처리하며, 특히 주식 데이터를 변경하는 buy, sell 요청의 경우 각 노드별로 semaphore를 이용하여 동기화를 수행한다.

5. 성능 측정 코드 추가

Task 1 에서의 변경사항과 같다.

3. 구현 결과

- stockclient 테스트

stockclient.c 프로그램을 실행해서 2번 항목에서 언급했던 각 명령어들이 정상적으로 동작하는 것을 확인했다.

```
cse20201617@cspro:~/cse4100/lab/prj3/20201617/task_1$ ./stockserver 60029
Connected to (172.30.10.9, 37450)
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 11 bytes
server received 5 bytes
server received 5 bytes
```

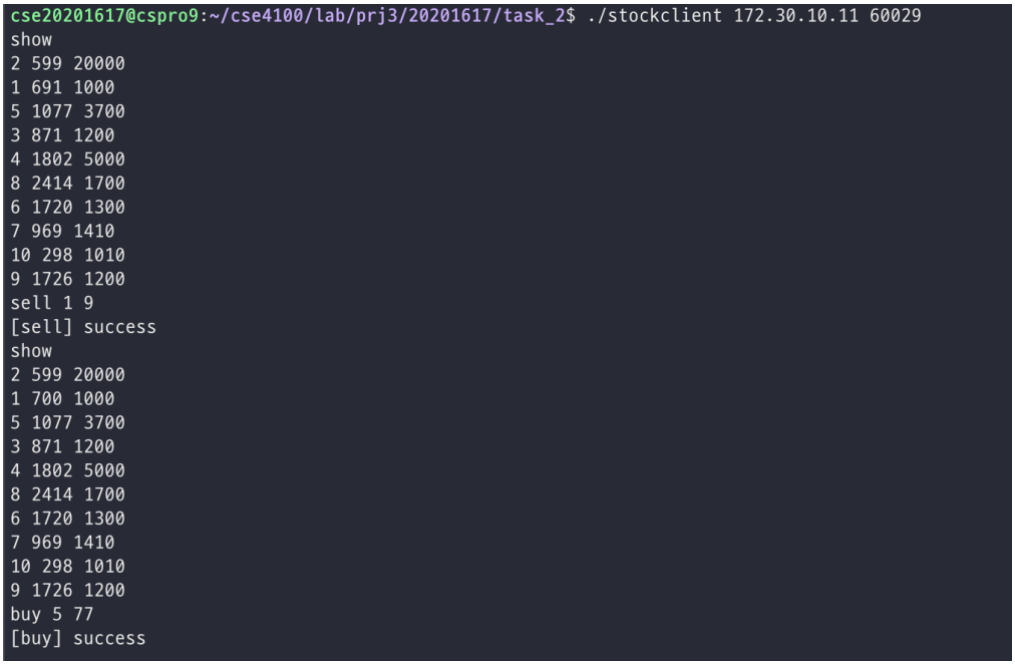
task_1 stockserver.c 실행 결과

```
cse20201617@cspro9:~/cse4100/lab/prj3/20201617/task_1$ ./stockclient 172.30.10.11 60029
show
2 586 20000
1 1200 1000
5 956 3700
3 576 1200
4 721 5000
8 797 1700
6 876 1300
7 604 1410
10 1029 1010
9 1672 1200
buy 3 10
[buy] success
show
2 586 20000
1 1200 1000
5 956 3700
3 566 1200
4 721 5000
8 797 1700
6 876 1300
7 604 1410
10 1029 1010
9 1672 1200
sell 2 100
[sell] success
```

task_1 stockclient.c 실행 결과

```
cse20201617@cspro:~/cse4100/lab/prj3/20201617/task_2$ ./stockserver 60029
start timer!!
Conncected to (172.30.10.9, 43988)
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 9 bytes
server received 5 bytes
server received 5 bytes
no client!!
>> elapsed time: 29.012
```

task_2 stockserver.c 실행 결과



task_2 stockclient.c 실행 결과

본 프로젝트에서 추가적으로 고려해 볼 만한 점은 주식 데이터를 저장하는 자료구조와 관련된 부분이다. 프로젝트 명세에 따라 현재 구현된 자료구조는 level-order 이진 트리이다. 이 자료구조 대신 ID를 기준으로 정렬해서 저장하는 BST를 이용하면 노드 탐색에 걸리는 시간을 줄일 수 있다. 물론 현재 저장되는 주식 데이터(node)의 수가 작기 때문에 실제 성능에는 큰 영향을 미치지 않는다. 그러나 대규모 서버 등 확장성을 고려했을 때에는 더 효율적인 자료구조를 사용하는 것이 필요하다.

4. 성능 평가 결과 (Task 3)

서버 프로그램은 cspro, 클라이언트 프로그램은 cspro9 서버에서 실행된다.

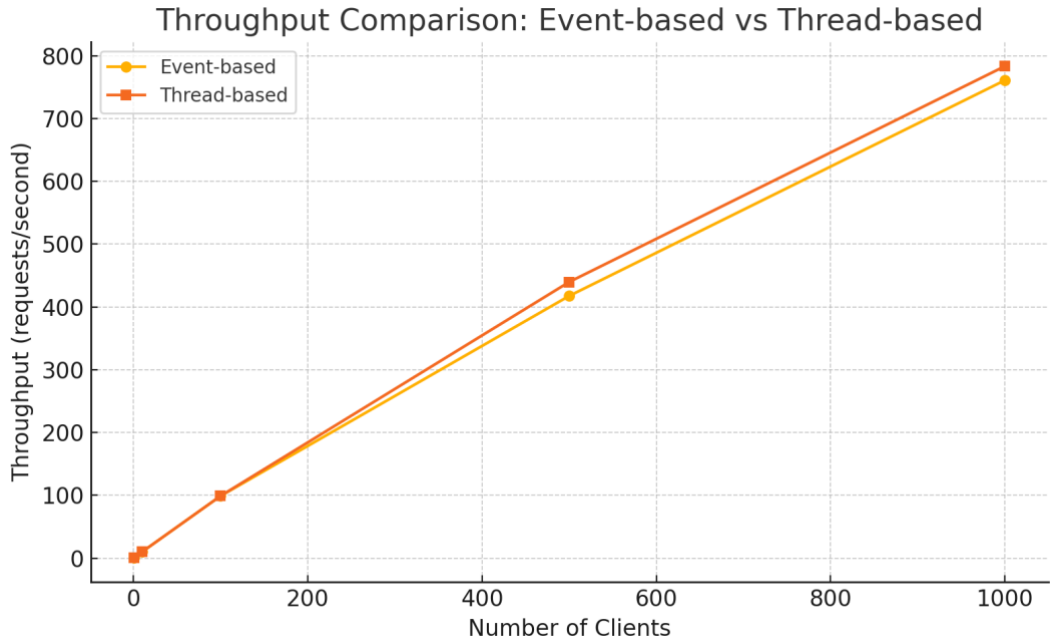
Thread-based 서버의 경우 NTHREADS 값을 변경함에 따라 Elapsed Time이 급격하게 달라지는 경향이 있었다. 그 이유는 Thread-based 서버는 각 워커 스레드는 하나의 클라이언트를 처리할 때에 그 클라이언트의 연결이 끝날 때까지 계속 같은 클라이언트를 처리한다. 그렇기 때문에 실험 시에 클라이언트의 수가 NTHREADS 보다 더 많은 경우에는 대략적으로 floor(CLINTES / NTHREADS) * 10초 정도로 Elapsed Time 이 관찰되었다. 이 문제를 해결하고 일반적인 비교·분석

을 위해 NTHREADS를 실험 시에 사용할 최대 클라이언트 수인 1000으로 설정해 둔 상태에서 실험을 진행했다.

처리율(Throughput)은 아래와 같은 공식을 통해 구했다. 실험 결과값은 3회 실행 시 결과값들의 평균값으로 얻었다.

$$Throughput = \frac{Total\ Number\ of\ Requests}{Elapsed\ Time}$$

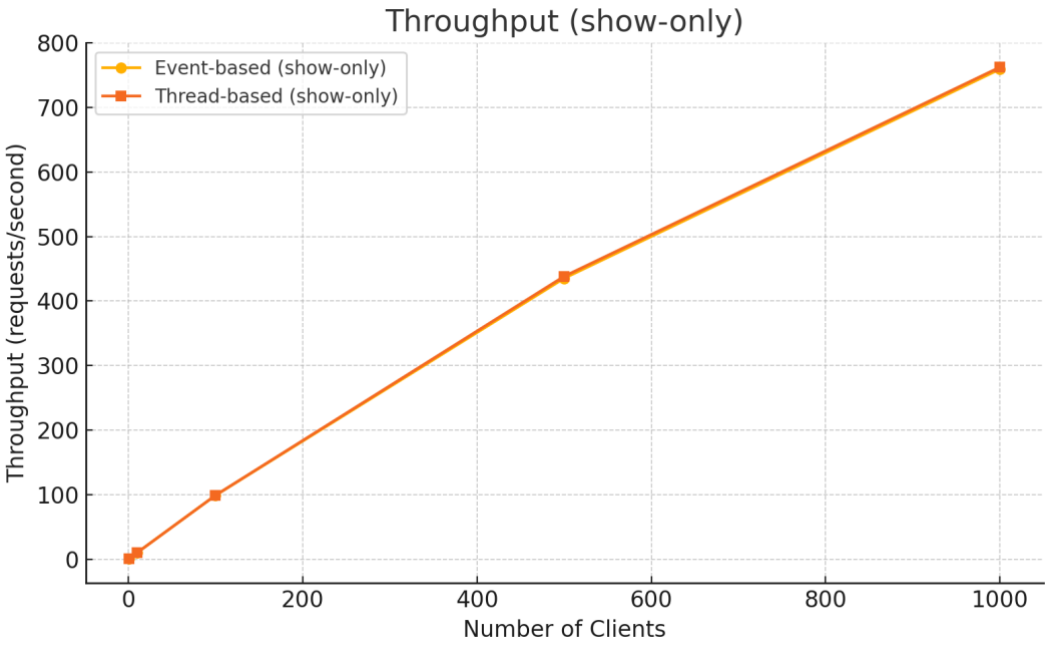
- 확장성(Scalability): 클라이언트 수 변화에 따른 처리율 비교



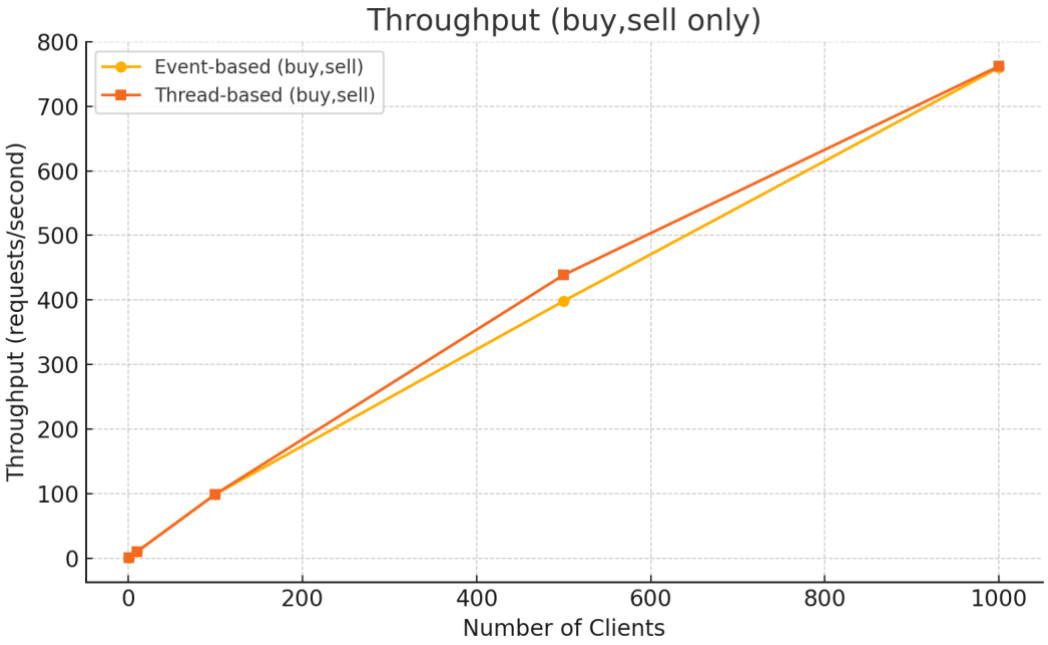
Event-based 서버와 Thread-based 서버의 클라이언트 수 변화에 따른 처리율 그래프이다. 두 서버의 처리율을 비교해보면 거의 동일하게 선형적으로 증가하는 것을 확인할 수 있다. 이런 식의 결과가 도출되는 이유는 현재 서버에서 클라이언트의 요청에 대해 처리하는 로직이 매우 단순하다. show 명령을 받으면, 주식 데이터를 저장하고 있는 이진 트리를 한 번 순회해서 그 내용들을 출력한다. buy, sell 명령을 받으면 역시 이진 트리를 순회하며 특정 노드를 찾아 그 노드에 대한 간단한 변경만 이루어진다. 그런데 multiclient.c 에서는 각 요청 사이에 usleep()을 호출해서 1초 간격으로 요청을 전송하도록 되어 있다. 실제 명령을 처리 하는 데에는 이보다 매우 적은 시간만 소요되지만 각 명령 사이에 1초의 지연이 있기 때문에 이 지연에 전체 elapsed time에서 Bottleneck 이 된다. 이 지연

없이 측정한다면 더 정확한 결과를 얻을 수 있을 것으로 예상된다.

- 워크로드(Workload): 클라이언트 요청 유형(buy, show, sell)에 따른 처리율 비교



클라이언트가 show 요청만 보내는 경우



클라이언트가 buy, sell 요청만 보내는 경우

위 그래프는 각각 클라이언트가 show 명령만 보내는 경우와 buy, sell 명령만 보

내는 경우에 처리율을 비교한 그래프이다. 우선 각 경우에 따른 처리율 차이는 유의미하다고 할 수 있을 만큼 나지 않았다. 이 원인은 앞서 설명한 것처럼 현재 각 요청을 처리하는 데에 걸리는 시간은 매우 적고, usleep() 의 지연이 bottleneck 이기 때문이다.

각 경우마다 두 개의 다른 접근 방식으로 구현된 서버의 처리율을 분석한 결과, 본 프로젝트의 클라이언트의 요청과 1000개 까지의 클라이언트 까지는 큰 차이가 나지 않는 것을 확인할 수 있었다. Thread-based 서버의 처리율이 약간씩 더 높은 경향을 보이고 있고, 이는 더 복잡한 동작을 수행하는 서버와 더 대용량 서버에서 더 많은 차이를 띌 것으로 예상된다.

아래는 위의 그래프에서 사용한 실제 측정값 데이터 표이다. 측정값의 신뢰도를 높이기 위해 각 Elapsed Time은 실험 3회 진행 후 평균값으로 설정했다.

Event-based						Thread-based						
Clients	1	10	100	500	1000	Clients	1	10	100	500	1000	
Elapsed Time	10.005461	10.01530133	10.11291067	11.97449567	13.140452	Elapsed Time	10.007	10.01666667	10.099	11.36933333	12.764	
Throughput	0.999454198	9.984722044	98.88349981	417.55412	761.0088298	Thrroughput(requests/second)	0.99930049	9.983361065	99.01970492	439.7795239	783.4534629	
show only Event-based						show only Thread-based						
Clients	1	10	100	500	1000	Clients	1	10	100	500	1000	
Elapsed Time	10.00527833	10.0153315	10.11576533	11.486077	13.16724433	Elapsed Time	10.00766667	10.01633333	10.10166667	11.40633333	13.115	
Throughput(requests/second)	0.999472445	9.98469197	98.85559491	435.3096362	759.4603508	Throughput(requests/second)	0.999233921	9.983693301	98.99356542	438.3529618	762.4857034	
buy, sell only Event-based						buy, sell only Thread-based						
Clients	1	10	100	500	1000	Clients	1	10	100	500	1000	
Elapsed Time	10.005553	10.01473467	10.114293	12.55434633	13.156916	Elapsed Time	10.00766667	10.017	10.09666667	11.39633333	13.12	
Throughput(request/second)	0.999445008	9.985287012	98.86998528	398.2684456	760.056536	Throughput(request/second)	0.999233921	9.983028851	99.04258831	438.7376057	762.195122	