Multicore Programming Project 2

담당 교수 : 박성용

이름 : 김윤성

학번 : 20191576

1. 개발 목표

Multi-Client를 위한 주식 서버를 Event-driven Approach와 Thread-based Approach 두 가지 방식으로 구현해본다. 또한 두 가지 방식으로 구현한 주식 서버의 차이점을 수행 속도 측면에서 살펴본다.

2. 개발 범위 및 내용

A. 개발 범위

1. Task 1: Event-driven Approach

Event-driven Approach 기반 Server는 충분한 양의 pool을 생성하고 Client의 Connection 요청을 기다린다. 그리고 어떤 Client와의 Connection이 이루어졌을 때, 해당 Client들의 connfd와 rio를 모두 pool에 저장해 놓는다. 이후 pool의 read_set을 통해 Client의 요청을 인식하고 해당 Client의 요청을 수행하고 결과를 전송한다.

2. Task 2: Thread-based Approach

Thread-based Approach 기반 Server는 충분한 양의 Worker Thread를 생성하고 Client의 Connection 요청을 기다린다. 그리고 어떤 Client와의 Connection이 이루어졌을 때, 해당 Client들의 connfd를 sbuf에 저장해 놓는다. Worker Thread는 sbuf에 저장되어 있는 가장 오래된 connfd를 하나 꺼내어 Client의 요청을 인식하고 해당 Client의 요청을 수행하고 결과를 전송한다.

3. Task 3: Performance Evaluation

Performance Evaluation에서는 Task1과 Task2 모두 Client와의 Connection 요청을 받을 때부터 모든 Client와의 Connection이 끊어질 때까지의 수행 속도를 계산한다. Client의 개수, Client의 Request 개수에 차이를 두어 두 가지 방식의 Server에 어떤 차이가 존재하는지 확인한다.

B. 개발 내용

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

Server는 pool 구조체를 만들고 관리한다. 먼저 Server는 비어 있는 read_set을 적당한 크기로 만들고 listenfd를 열어 Client의 Connection 요청이 올 때까지 기다린다. Client으로부터 Connection 요청이 오면 Server는 Accept를 통해 Client의 Connection을 수락하고 read_set의 비어 있고 index가 가장 작은 곳의 비트를 1로 설정한다.

이후 read_set에 저장되어 있는 정보를 바탕으로 Client의 요청을 수행한다. 먼저 Server는 read_set을 차례로 순회하면서 1로 설정되어 있는 index의 connfd를 이용해 Client의 요청을

받아오고, 이에 맞는 결과를 Client에게 보내준다.

epoll은 select와 유사하게 모두 I/O Multiplexing을 위한 함수이다. 하지만 epoll은 거의 모든 OS에서 사용 가능한 select와는 다르게 LINUX에서만 사용이 가능하다. 또한 select는 사용자가 pool과 같은 구조체를 통해 파일 디스크립터를 관리하지만 epoll은 커널에서 파일 디스크립터를 관리한다. 마지막으로 select는 파일 디스크립터를 순차 순회하며 일일히 탐색하며 변화를 관찰하지만 epoll은 변화가 발생한 파일 디스크립터만을 탐색할 수 있으므로 select보다 효율이 좋다.

- Task2 (Thread-based Approach with pthread)

Master Thread는 sbuf_t 구조체를 만들고 관리한다. 먼저 Master Thread는 비어 있는 Worker Thread를 SBUFSIZE만큼 만들고 listenfd를 열어 Client의 Connection 요청이 올 때까지 기다린다. Client으로부터 Connection 요청이 오면, Master Thread는 connfd를 sbuf에 삽입한다. 이후에는 Worker Thread에서 Client의 요청을 수행한다.

Worker Thread는 sbuf_t 구조체에 저장되어 있는 정보를 바탕으로 Client의 요청을 수행한다. 먼저 Worker Thread는 sbuf_t에서 가장 오래된 connfd를 꺼낸다. 이후 connfd를 이용해 Client의 요청을 받아오고, 이에 맞는 결과를 Client에게 보내준다.

- Task3 (Performance Evaluation)

Performance Evaluation에서는 Task1과 Task2의 수행 속도를 비교한다. Client의 개수와 각각 Client마다의 Request 개수에 의해 수행 속도에 차이가 나기 때문에 본 Evaluation에서 얻을 수 있는 Metric은 Client 개수를 일정하게 통제하거나 Client의 Request 개수를 일정하게 통제함으로써 얻을 수 있다. Server와 이루어지는 Client의 첫 Connection이 이루어진 시점에서부터 모든 Client의 Connection이 끝나는 시점까지의 시간을 측정하여 수행 속도를 계산할 수 있다.

본 Evaluation에서 Client의 개수가 늘어날수록 수행 속도가 느려지고, Client의 Request 개수가 늘어날수록 수행 속도가 느려질 것으로 예상할 수 있다. 다만 변인이 늘어남에 따라 결과가 어떤 방식으로 증가($O(n)$, $O(n^2)$, $O(\log_2 n)$)할지는 예상하기 힘들다. 또한 두 방식으로 구현한 Server 간에 어떠한 차이를 보일지도 예상하기 힘들다.

C. 개발 방법

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

```
int byte_cnt = 0;                       /* Counts total bytes received by server*/
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;
```

위 그림은 pool 구조체의 정의이다. Int형 배열 clientfd에 Client의 요청에 따른 connfd가 비어 있는 가장 빠른 index에 저장되고, rio_t형 배열 clientrio에 Client의 요청에 따른 rio가 비어 있는 가장 빠른 index에 저장된다.

```
listenfd = Open_listenfd(argv[1]);
init_pool(listenfd, &pool);

while (1) {
    pool.ready_set = pool.read_set;     /* Wait for listening / connected descriptor(s) to become ready */
    pool.nready = Select(pool.maxfd + 1, &pool.ready_set, NULL, NULL, NULL);

    /* If listening descriptor is ready, add new client to pool */
    if (FD_ISSET(listenfd, &pool.ready_set)) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Getnameinfo((SA *) &clientaddr, clientlen, client_hostname, MAXLINE,
                    client_port, MAXLINE, 0);
        printf("Connected to (%s, %s)\n", client_hostname, client_port);
        add_client(connfd, &pool);
    }
```

Server는 위의 코드와 같이 listenfd를 열어 놓고 pool 구조체를 초기화하고 대기한다. 이후 while문 안에서 Client의 Connection 요청이 오면 Accept를 통해 이를 수락하고, add_client를 이용해 해당 connfd를 pool에 저장한다.

```c
void add_client(int connfd, pool *p) {
    int i;
    p->nready--;
    /* Find an available slot */
    for (i = 0; i < FD_SETSIZE; i++) {
        /* If slot is empty */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);

            /* Update max descriptor and pool high water mark */
            if (connfd > p->maxfd) {
                p->maxfd = connfd;
            }
            if ( i > p->maxi) {
                p->maxi = i;
            }
            break;
        }
    }

    /* Couldn't find an empty slot */
    if (i == FD_SETSIZE) {
        app_error("add_client error: Too many clients");
    }
}
```

위 그림은 add_client의 정의이다. Server는 add_client를 통해 pool의 비어 있는 가장 작은 index에 connfd를 저장한다. 만약 더 이상 비어 있는 pool이 없다면 오류를 출력한다.

```
for(int i = 0; (i <= pool.maxi) && (pool.nready > 0); i++) {
    connfd = pool.clientfd[i];
    rio = pool.clientrio[i];

    /* If the descriptor is ready, echo a text line from it */
    if ((connfd > 0) && (FD_ISSET(connfd, &pool.ready_set))) {
        pool.nready--;

        if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
            byte_cnt += n;
            printf("server received %d bytes on fd %d\n", n, connfd);
            if (!strncmp(buf, "show", 4)) {···
            }
            else if(!strncmp(buf, "buy", 3)) {···
            }
            else if (!strncmp(buf, "sell", 4)) {···
            }
            Rio_writen(connfd, buf, MAXLINE);
        }
        /* EOF detected, remove descriptor from pool */
        else {
            sem_wait(&file_mutex);
            fp = fopen("stock.txt", "w");
            fprint_Node(root, fp);
            fclose(fp);
            sem_post(&file_mutex);

            Close(connfd);
            FD_CLR(connfd, &pool.read_set);
            pool.clientfd[i] = -1;
        }
    }
}
```

connfd를 pool에 저장한 이후, Server는 pool을 순회하며 각각의 index에 저장되어 있는 connfd와 rio를 꺼내 Client의 요청을 받게 된다. Client의 요청에 맞는 적절한 수행을 거친 뒤, Rio_writen을 통해 Client에게 해당 요청의 결과를 보내주게 된다. Client의 요청에 알맞은 수행을 하고 난 뒤에도 connfd와 rio는 해당 Client의 다음 요청을 수행하기 위해 삭제하지 않는다.

- Task2 (Thread-based Approach with pthread)

```
#define SBUFSIZE 1024
typedef struct {
    int *buf;                   /* Buffer array */
    int n;                      /* Maximum number of slots */
    int front;                  /* buf[(front + 1) % n] is the first item */
    int rear;                   /* buf[rear % n] is the last item */
    sem_t mutex;                /* Protects accesses to buf */
    sem_t slots;                /* Counts available slots */
    sem_t items;                /* Counts available items */
} sbuf_t;
```

위 그림은 sbuf 구조체의 정의이다. int*형 buf에 Client의 요청에 따른 connfd가 쌓이게 된다.

```
listenfd = Open_listenfd(argv[1]);
sbuf_init(&sbuf, SBUFSIZE);
Sem_init(&file_mutex, 0, 1);

// Create a pool of worker threads /
for (int i = 0; i < SBUFSIZE; i++) {
    Pthread_create(&tid, NULL, thread, NULL);
}

while (1) {
    // If listening descriptor is ready, add new client to pool /
    clientlen = sizeof(struct sockaddr_storage);
    connfd = (int*)malloc(sizeof(int));
    *connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    Getnameinfo((SA *) &clientaddr, clientlen, client_hostname, MAXLINE,
                client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);
    // Insert connfd in buffer /
    sbuf_insert(&sbuf, *connfd);
}
exit(0);
```

Master Thread는 위의 코드와 같이 listenfd를 열어 놓고 SBUFSIZE만큼 Worker Thread를 만들어 놓고 대기한다. 이후 while문 안에서 Client의 Connection 요청이 오면 Accept를 통해 이를 수락하고, sbuf에 해당 connfd를 저장한다.

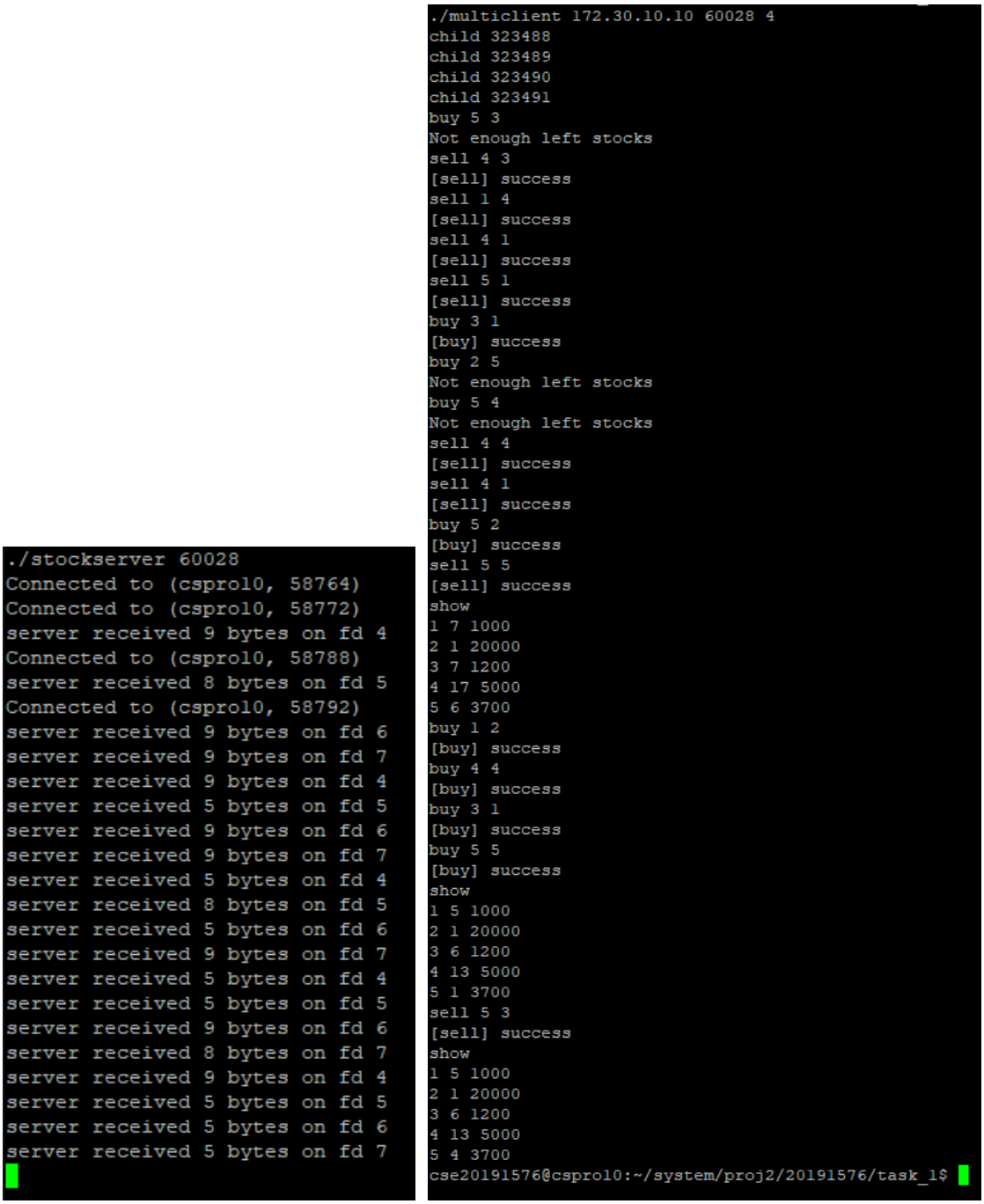
```
void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    while(1) {
        /* Remove connfd from buf */
        int connfd = sbuf_remove(&sbuf);
        /* Service Client */
        //echo_cnt(connfd);
        int n;
        char buf[MAXLINE];
        rio_t rio;
        itemNode* ptr;

        Rio_readinitb(&rio, connfd);
        while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
            printf("thread %d received %d bytes on fd %d\n", (int)pthread_self(), n, connfd);
            if (!strncmp(buf, "show", 4)) {···
            }
            else if(!strncmp(buf, "buy", 3)) {···
            }
            else if (!strncmp(buf, "sell", 4)) {···
            }
            Rio_writen(connfd, buf, MAXLINE);
        }
        /* EOF detected, remove descriptor from pool */
        P(&file_mutex);
        fp = fopen("stock.txt", "w");
        fprint_Node(root, fp);
        fclose(fp);
        V(&file_mutex);

        Close(connfd);
    }
}
```

Worker Thread는 sbuf_remove를 통해 sbuf의 가장 오래된 connfd를 꺼내고 rio를 통해 Client의

요청을 받게 된다. Client의 요청에 맞는 적절한 수행을 거친 뒤, Rio_writen을 통해 Client에게 해당 요청의 결과를 보내주게 된다. 이후 Worker Thread는 수행이 끝난 connfd를 삭제함으로써 Client의 요청을 마무리 짓는다.

3. 구현 결과

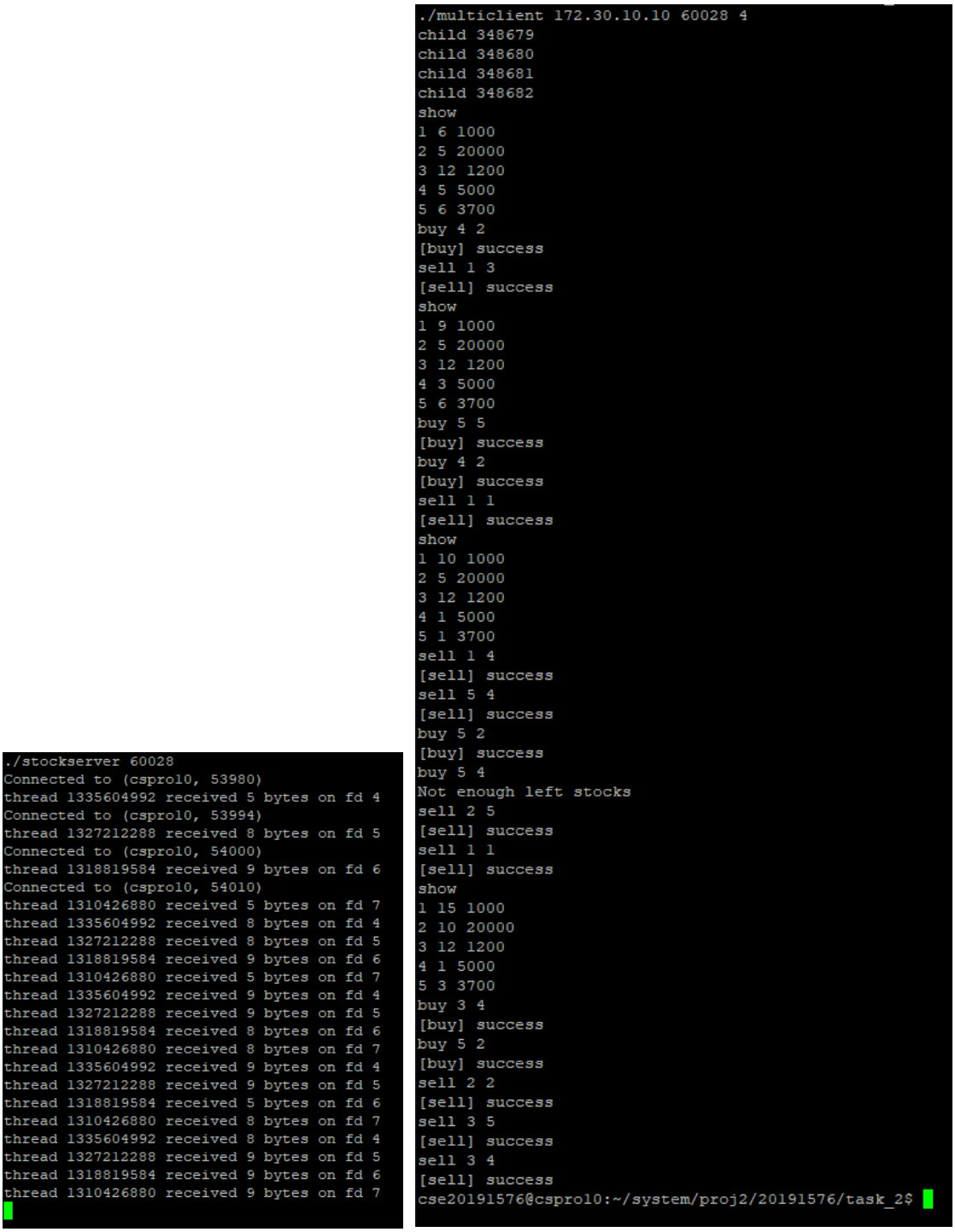
- Task1 (Event-driven Approach with select())



위 그림은 4개의 Client가 동시에 Server에 접속하여 각각 5개의 Request를 보낸 결과이다. 왼쪽은 Server의 화면, 오른쪽은 Client의 화면이다.

Server는 Client와의 Connection이 이루어지면 Client의 host name과 port를 출력한다. 또한 Client에서 Request를 받으면 해당 Request의 byte와 descriptor를 출력한다.
Client는 자신의 process id를 출력하고, 보낼 Request를 출력한다. 이어서 Server에서 해당 Request에 대응하는 Response를 받고 이를 출력한다.

- Task2 (Thread-based Approach with pthread)



위 그림은 4개의 Client가 동시에 Server에 접속하여 각각 5개의 Request를 보낸 결과이다.

왼쪽은 Server의 화면, 오른쪽은 Client의 화면이다.

Server는 Client와의 Connection이 이루어지면 Client의 host name과 port를 출력한다. 또한 Client에서 Request를 받으면 해당 Request의 byte와 descriptor를 출력한다.

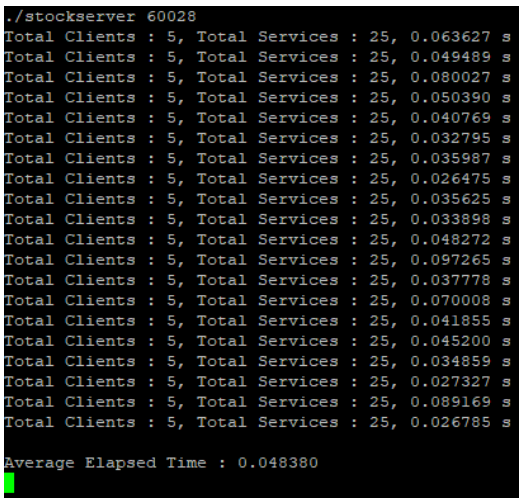
Client는 자신의 process id를 출력하고, 보낼 Request를 출력한다. 이어서 Server에서 해당 Request에 대응하는 Response를 받고 이를 출력한다.

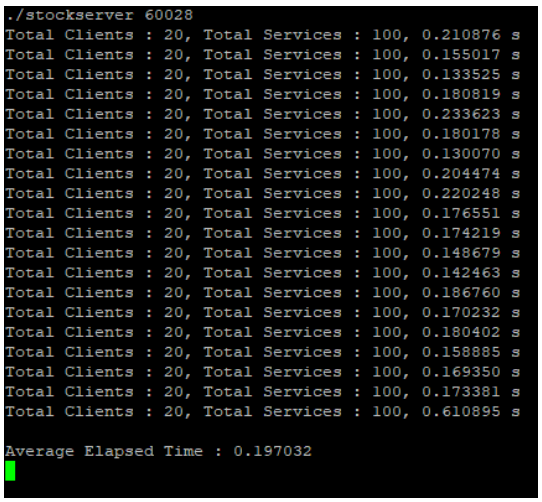
4. 성능 평가 결과 (Task 3)

성능 평가를 위해 Client의 개수는 5, 20, 100, 500개, Client Request의 개수는 5, 20, 100, 500개로 설정하여 실험했다. Client Request의 개수를 고정하고 Client의 개수를 조절하며 실험했다.

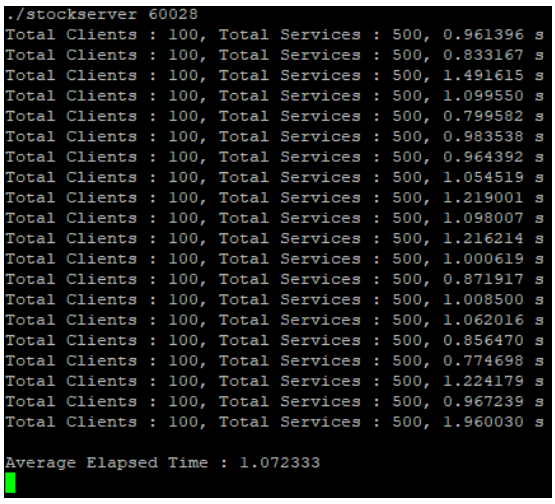
A. Event-driven Approach

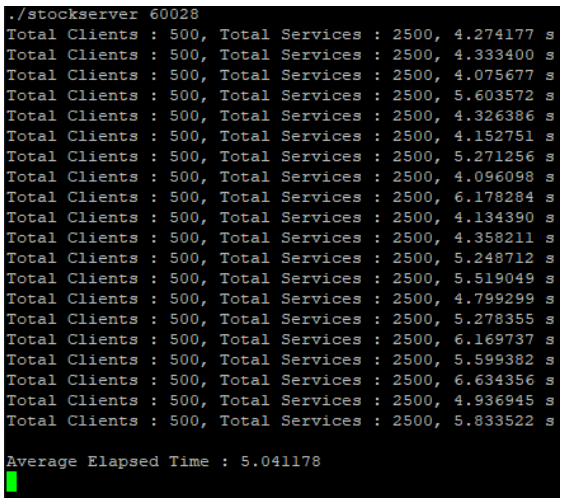
- Client Request를 5로 고정





좌: Client의 개수가 5 우: Client의 개수가 20





좌: Client의 개수가 100 우: Client의 개수가 500

- Client Request를 20으로 고정

```
./stockserver 60028
Total Clients : 5, Total Services : 100, 0.096559 s
Total Clients : 5, Total Services : 100, 0.047026 s
Total Clients : 5, Total Services : 100, 0.043549 s
Total Clients : 5, Total Services : 100, 0.058006 s
Total Clients : 5, Total Services : 100, 0.056498 s
Total Clients : 5, Total Services : 100, 0.054793 s
Total Clients : 5, Total Services : 100, 0.050569 s
Total Clients : 5, Total Services : 100, 0.029120 s
Total Clients : 5, Total Services : 100, 0.087554 s
Total Clients : 5, Total Services : 100, 0.061560 s
Total Clients : 5, Total Services : 100, 0.055971 s
Total Clients : 5, Total Services : 100, 0.090800 s
Total Clients : 5, Total Services : 100, 0.041227 s
Total Clients : 5, Total Services : 100, 0.036821 s
Total Clients : 5, Total Services : 100, 0.040416 s
Total Clients : 5, Total Services : 100, 0.054882 s
Total Clients : 5, Total Services : 100, 0.027483 s
Total Clients : 5, Total Services : 100, 0.038353 s
Total Clients : 5, Total Services : 100, 0.030873 s
Total Clients : 5, Total Services : 100, 0.055603 s

Average Elapsed Time : 0.052883
```

```
./stockserver 60028
Total Clients : 20, Total Services : 400, 0.239994 s
Total Clients : 20, Total Services : 400, 0.301534 s
Total Clients : 20, Total Services : 400, 0.147960 s
Total Clients : 20, Total Services : 400, 0.213130 s
Total Clients : 20, Total Services : 400, 0.187706 s
Total Clients : 20, Total Services : 400, 0.224997 s
Total Clients : 20, Total Services : 400, 0.183580 s
Total Clients : 20, Total Services : 400, 0.214682 s
Total Clients : 20, Total Services : 400, 0.219516 s
Total Clients : 20, Total Services : 400, 0.199793 s
Total Clients : 20, Total Services : 400, 0.178826 s
Total Clients : 20, Total Services : 400, 0.189154 s
Total Clients : 20, Total Services : 400, 0.199961 s
Total Clients : 20, Total Services : 400, 0.161625 s
Total Clients : 20, Total Services : 400, 0.135425 s
Total Clients : 20, Total Services : 400, 0.185772 s
Total Clients : 20, Total Services : 400, 0.213761 s
Total Clients : 20, Total Services : 400, 0.212854 s
Total Clients : 20, Total Services : 400, 0.241662 s
Total Clients : 20, Total Services : 400, 0.234900 s

Average Elapsed Time : 0.204342
```

좌: Client의 개수가 5 우: Client의 개수가 20

```
./stockserver 60028
Total Clients : 100, Total Services : 2000, 0.709677 s
Total Clients : 100, Total Services : 2000, 0.786501 s
Total Clients : 100, Total Services : 2000, 0.900961 s
Total Clients : 100, Total Services : 2000, 1.311448 s
Total Clients : 100, Total Services : 2000, 0.864497 s
Total Clients : 100, Total Services : 2000, 1.930997 s
Total Clients : 100, Total Services : 2000, 1.231408 s
Total Clients : 100, Total Services : 2000, 0.883951 s
Total Clients : 100, Total Services : 2000, 0.966257 s
Total Clients : 100, Total Services : 2000, 0.931445 s
Total Clients : 100, Total Services : 2000, 0.851431 s
Total Clients : 100, Total Services : 2000, 1.002675 s
Total Clients : 100, Total Services : 2000, 0.783075 s
Total Clients : 100, Total Services : 2000, 0.872771 s
Total Clients : 100, Total Services : 2000, 0.886903 s
Total Clients : 100, Total Services : 2000, 1.007999 s
Total Clients : 100, Total Services : 2000, 1.128884 s
Total Clients : 100, Total Services : 2000, 0.815600 s
Total Clients : 100, Total Services : 2000, 0.969610 s
Total Clients : 100, Total Services : 2000, 0.997847 s

Average Elapsed Time : 0.991697
```

```
./stockserver 60028
Total Clients : 500, Total Services : 10000, 4.404403 s
Total Clients : 500, Total Services : 10000, 4.523865 s
Total Clients : 500, Total Services : 10000, 5.618697 s
Total Clients : 500, Total Services : 10000, 6.433036 s
Total Clients : 500, Total Services : 10000, 5.286607 s
Total Clients : 500, Total Services : 10000, 4.684193 s
Total Clients : 500, Total Services : 10000, 4.686142 s
Total Clients : 500, Total Services : 10000, 4.668484 s
Total Clients : 500, Total Services : 10000, 6.221697 s
Total Clients : 500, Total Services : 10000, 4.491853 s
Total Clients : 500, Total Services : 10000, 4.517231 s
Total Clients : 500, Total Services : 10000, 6.105253 s
Total Clients : 500, Total Services : 10000, 5.474881 s
Total Clients : 500, Total Services : 10000, 7.547679 s
Total Clients : 500, Total Services : 10000, 5.300706 s
Total Clients : 500, Total Services : 10000, 5.065076 s
Total Clients : 500, Total Services : 10000, 5.542380 s
Total Clients : 500, Total Services : 10000, 6.458140 s
Total Clients : 500, Total Services : 10000, 6.338982 s
Total Clients : 500, Total Services : 10000, 4.959825 s

Average Elapsed Time : 5.416456
```

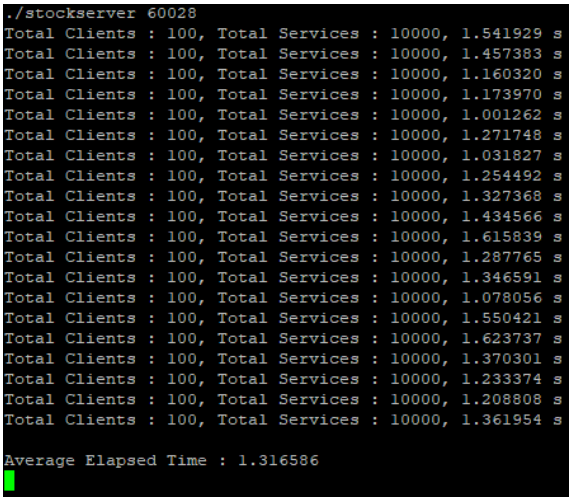
좌: Client의 개수가 100 우: Client의 개수가 500

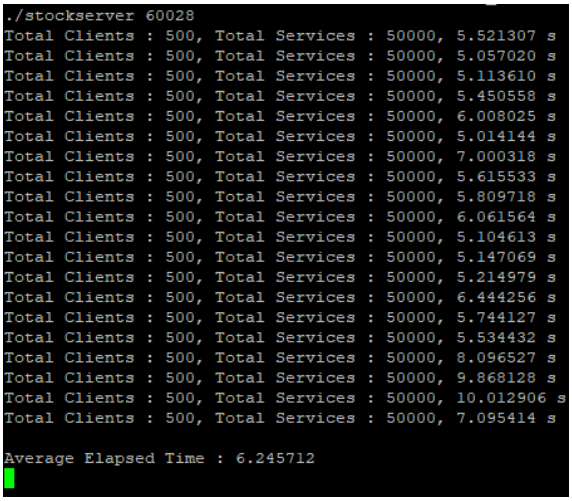
- Client Request를 100으로 고정

```
./stockserver 60028
Total Clients : 5, Total Services : 500, 0.169183 s
Total Clients : 5, Total Services : 500, 0.051944 s
Total Clients : 5, Total Services : 500, 0.036580 s
Total Clients : 5, Total Services : 500, 0.069605 s
Total Clients : 5, Total Services : 500, 0.123978 s
Total Clients : 5, Total Services : 500, 0.049070 s
Total Clients : 5, Total Services : 500, 0.069593 s
Total Clients : 5, Total Services : 500, 0.059717 s
Total Clients : 5, Total Services : 500, 0.035566 s
Total Clients : 5, Total Services : 500, 0.044359 s
Total Clients : 5, Total Services : 500, 0.076858 s
Total Clients : 5, Total Services : 500, 0.082886 s
Total Clients : 5, Total Services : 500, 0.113990 s
Total Clients : 5, Total Services : 500, 0.090923 s
Total Clients : 5, Total Services : 500, 0.113432 s
Total Clients : 5, Total Services : 500, 0.134912 s
Total Clients : 5, Total Services : 500, 0.106490 s
Total Clients : 5, Total Services : 500, 0.076348 s
Total Clients : 5, Total Services : 500, 0.100004 s
Total Clients : 5, Total Services : 500, 0.102810 s

Average Elapsed Time : 0.085412
```

```
./stockserver 60028
Total Clients : 20, Total Services : 2000, 0.334883 s
Total Clients : 20, Total Services : 2000, 0.237832 s
Total Clients : 20, Total Services : 2000, 0.228914 s
Total Clients : 20, Total Services : 2000, 0.245211 s
Total Clients : 20, Total Services : 2000, 0.258053 s
Total Clients : 20, Total Services : 2000, 0.290565 s
Total Clients : 20, Total Services : 2000, 0.211896 s
Total Clients : 20, Total Services : 2000, 0.221272 s
Total Clients : 20, Total Services : 2000, 0.200786 s
Total Clients : 20, Total Services : 2000, 0.211791 s
Total Clients : 20, Total Services : 2000, 0.247924 s
Total Clients : 20, Total Services : 2000, 0.250107 s
Total Clients : 20, Total Services : 2000, 0.276595 s
Total Clients : 20, Total Services : 2000, 0.293526 s
Total Clients : 20, Total Services : 2000, 0.265818 s
Total Clients : 20, Total Services : 2000, 0.249176 s
Total Clients : 20, Total Services : 2000, 0.218010 s
Total Clients : 20, Total Services : 2000, 0.244676 s
Total Clients : 20, Total Services : 2000, 0.237049 s
Total Clients : 20, Total Services : 2000, 0.162635 s

Average Elapsed Time : 0.244336
```

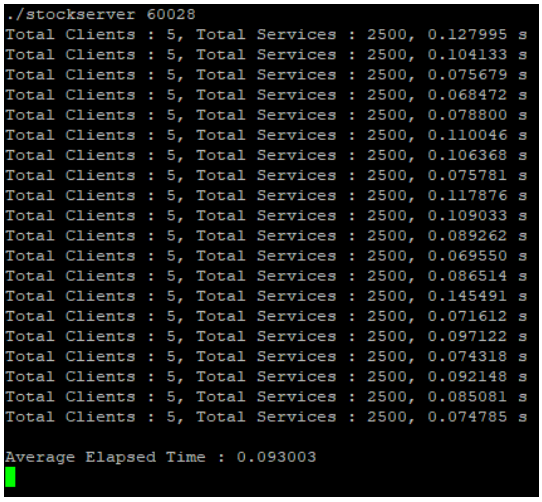
좌: Client의 개수가 5 우: Client의 개수가 20

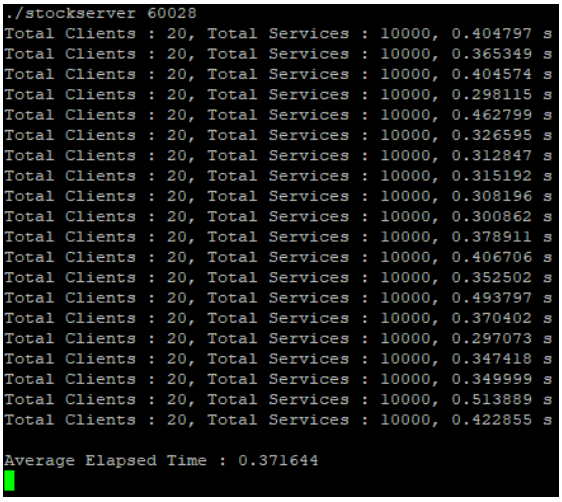




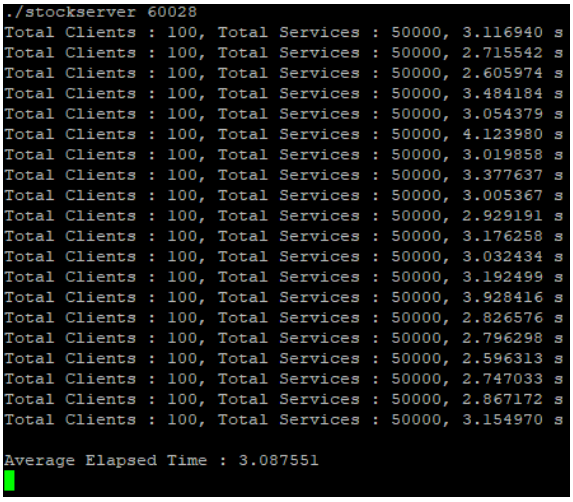
좌: Client의 개수가 100　　우: Client의 개수가 500

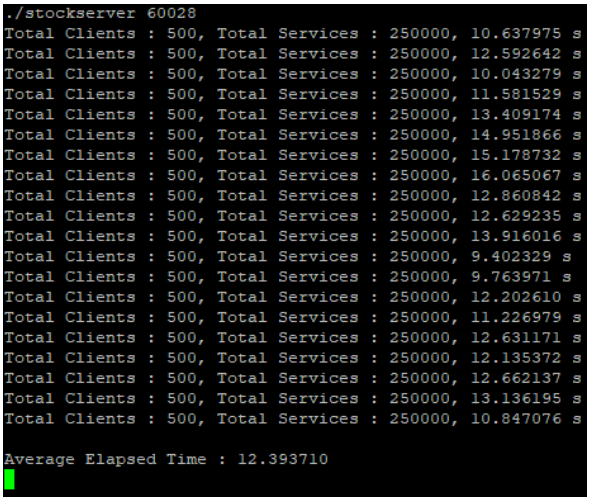
- Client Request를 500으로 고정





좌: Client의 개수가 5　　우: Client의 개수가 20





좌: Client의 개수가 100　　우: Client의 개수가 500

즉, Event-driven Approach의 Client의 개수와 Client Request의 개수에 따른 시간 변화를 표로 나타내면 다음과 같다.

Client \ Request	5	20	100	10000
5	0.04838	0.052883	0.085412	0.093003
20	0.197032	0.204342	0.244336	0.371644
100	1.072333	0.991697	1.316586	3.087551
10000	53041178	5.416456	6.245712	12.39371