CS 실습 과제 보고서

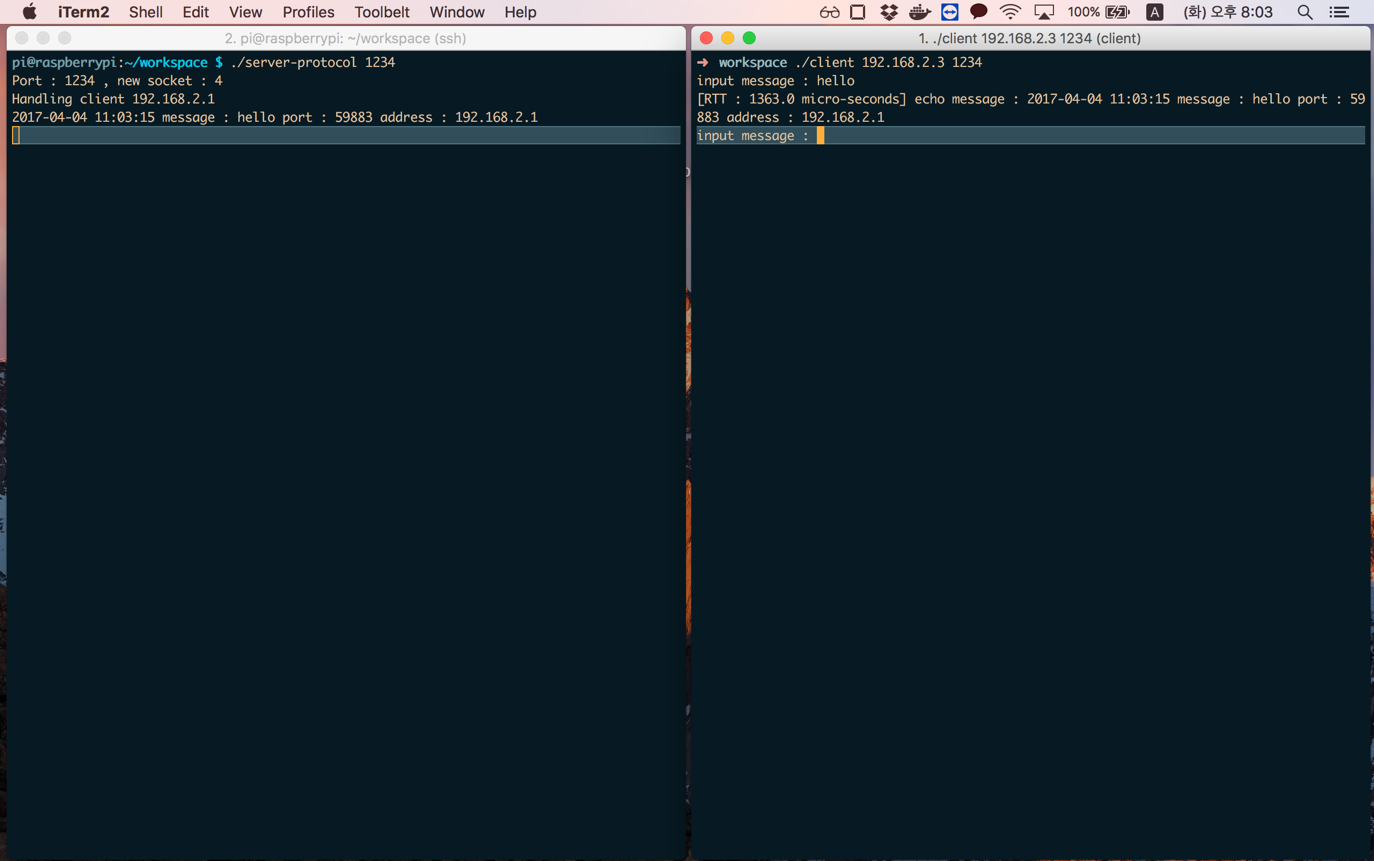
201121013 미디어학과 김유석

보고서의 내용을 작성하기에 앞서, 본고는 필자의 홈 네트워크(공유기 사용)에서 맥북과 라즈베리파이를 이용하여 발생한 결과와 스크린샷을 사용하였음을 밝힌다.

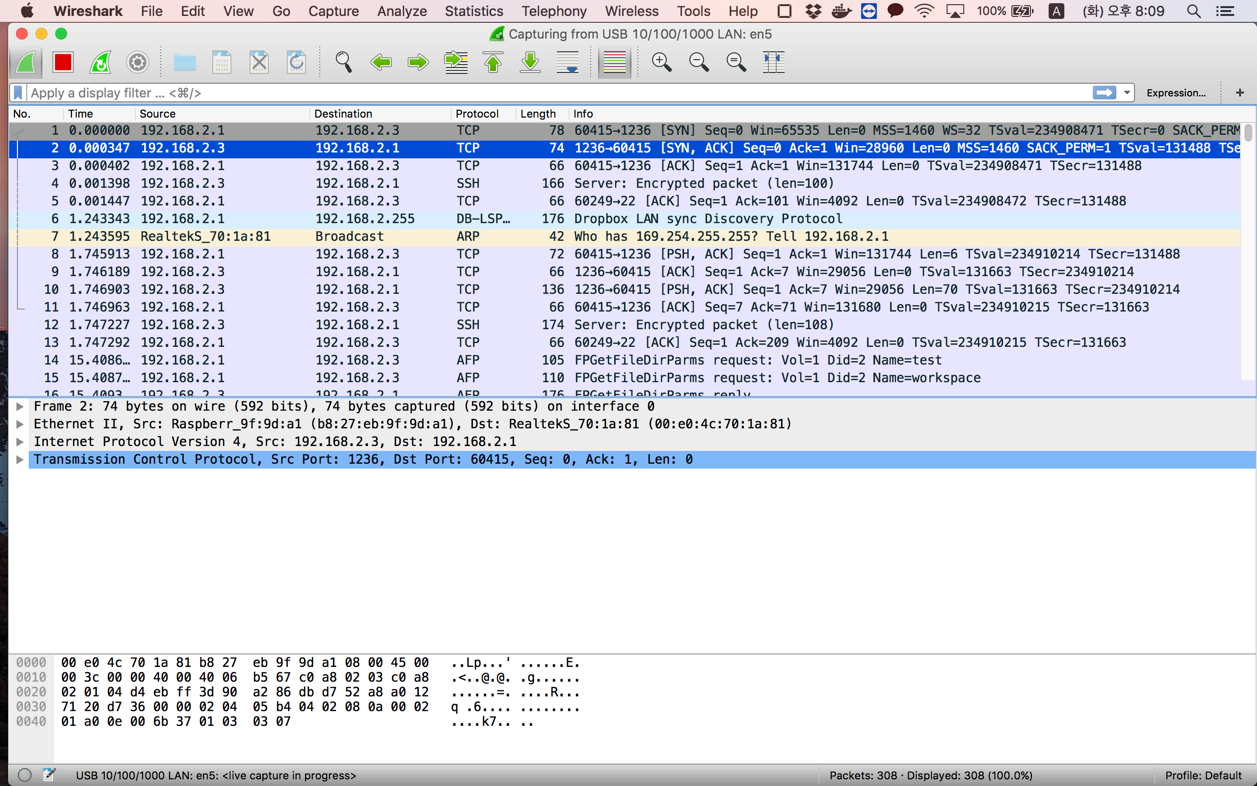
**1-1** iterative CO echo client-server 프로토콜 수정

**프로토콜 구현**  
 프로토콜 구현 방법은 다음과 같다. Client에서 Server로 메시지를 보내면 Server는 해당 메시지 내용을 [메시지를 받은 시간] [client address] [client port] [메시지]로 변경하여 이를 print하고, 해당 메시지를 Client에 echo한다. Client는 메시지를 보내기 전과 후의 시간을 비교하여, RTT를 계산하여 서버가 보낸 메시지와 RTT 값을 함께 출력한다.

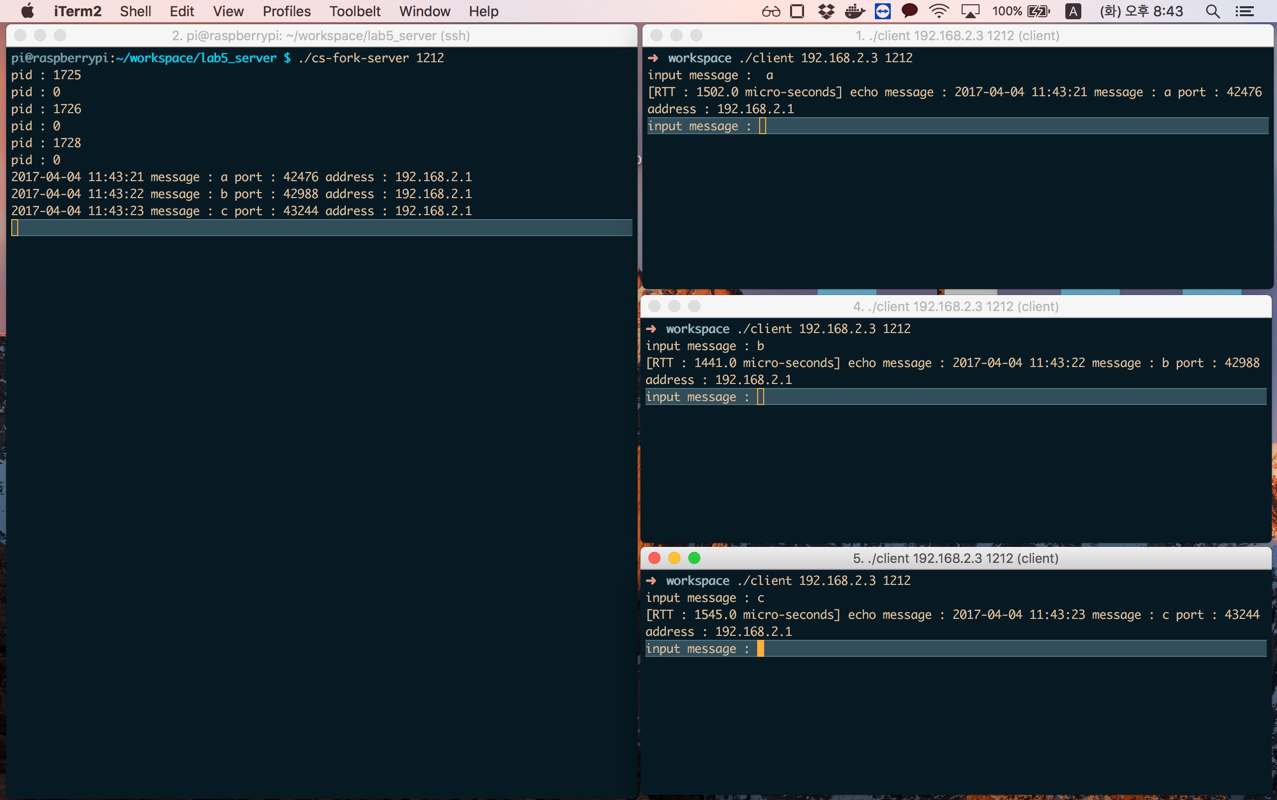
테스트를 위해 먼저, 라즈베리파이에서 CO Echo server를 port넘버 1234로 실행시켰다. 다음으로 맥북에서 클라이언트를 라즈베리파이의 IP주소(192.168.2.3)와 port넘버(1234)를 입력하여 실행시킨 후, 메시지를 보냈다. 동작 결과는 다음과 같다.



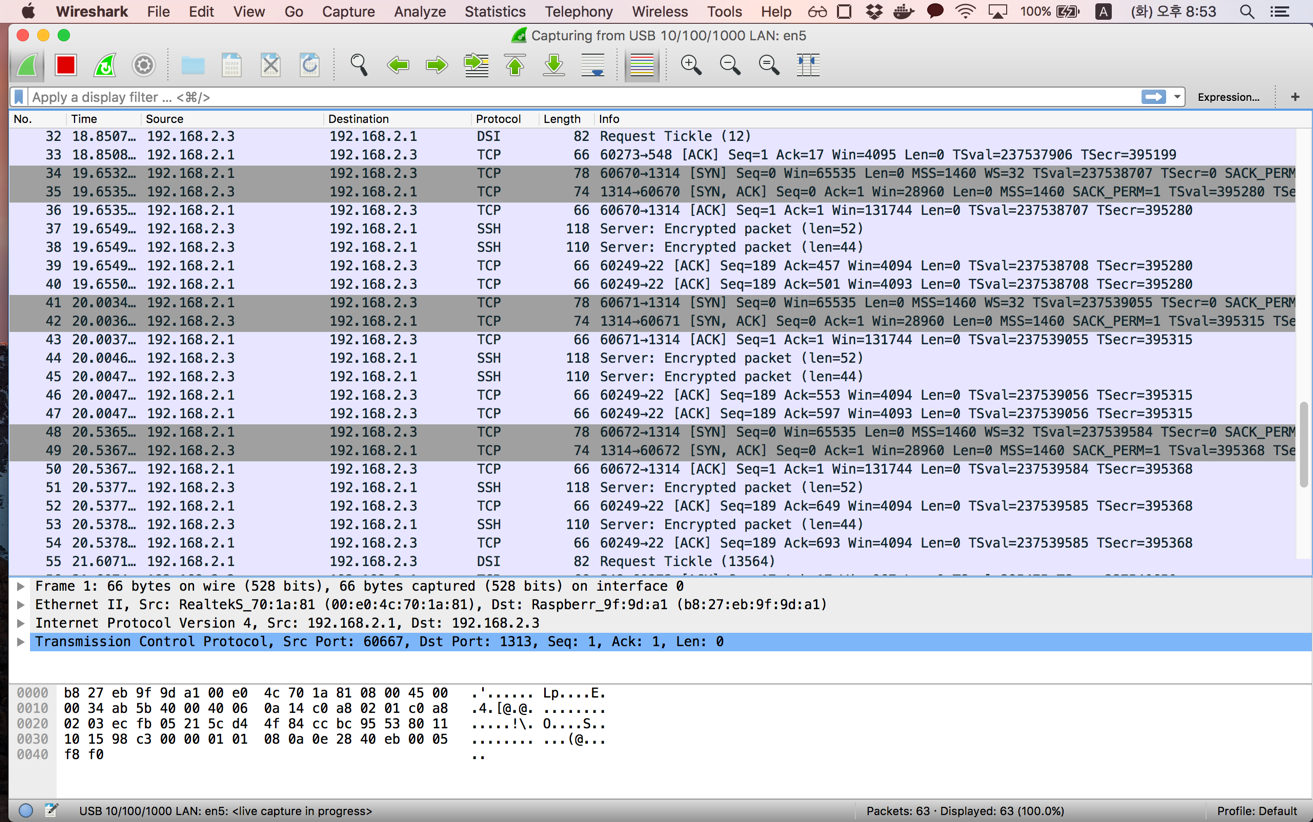
서버가 왼쪽, 오른쪽이 클라이언트이다. ‘hello’라는 메시지를 클라이언트에서 전송하자, 서버에서는 메시지에 추가로 클라이언트의 정보와 시간 등을 추가하여 print한다. 이후 echo 메시지 클라이언트에게 보내면 클라이언트에서 메시지를 받음을 확인할 수 있다. 또한 추가로 RTT가 micro-second단위로 출력됨을 확인할 수 있다. ‘와이어샤크’에서도 정상적으로 connection이 이루어짐을 확인할 수 있었다.



**1-2** Fork CS 구현

Fork CS는 Parent는 listen만 담당하고, Client의 Connect 요청이 있을 때마다 fork를 통해 새로운 프로세스를 생성하여 해당 프로세스가 accept하여 통신하도록 구현하였다. 때문에 기능상 가장 큰 차이점은 여러 클라이언트와 Connection을 맺을 수 있었다는 점이다. 아래 그림을 보자.

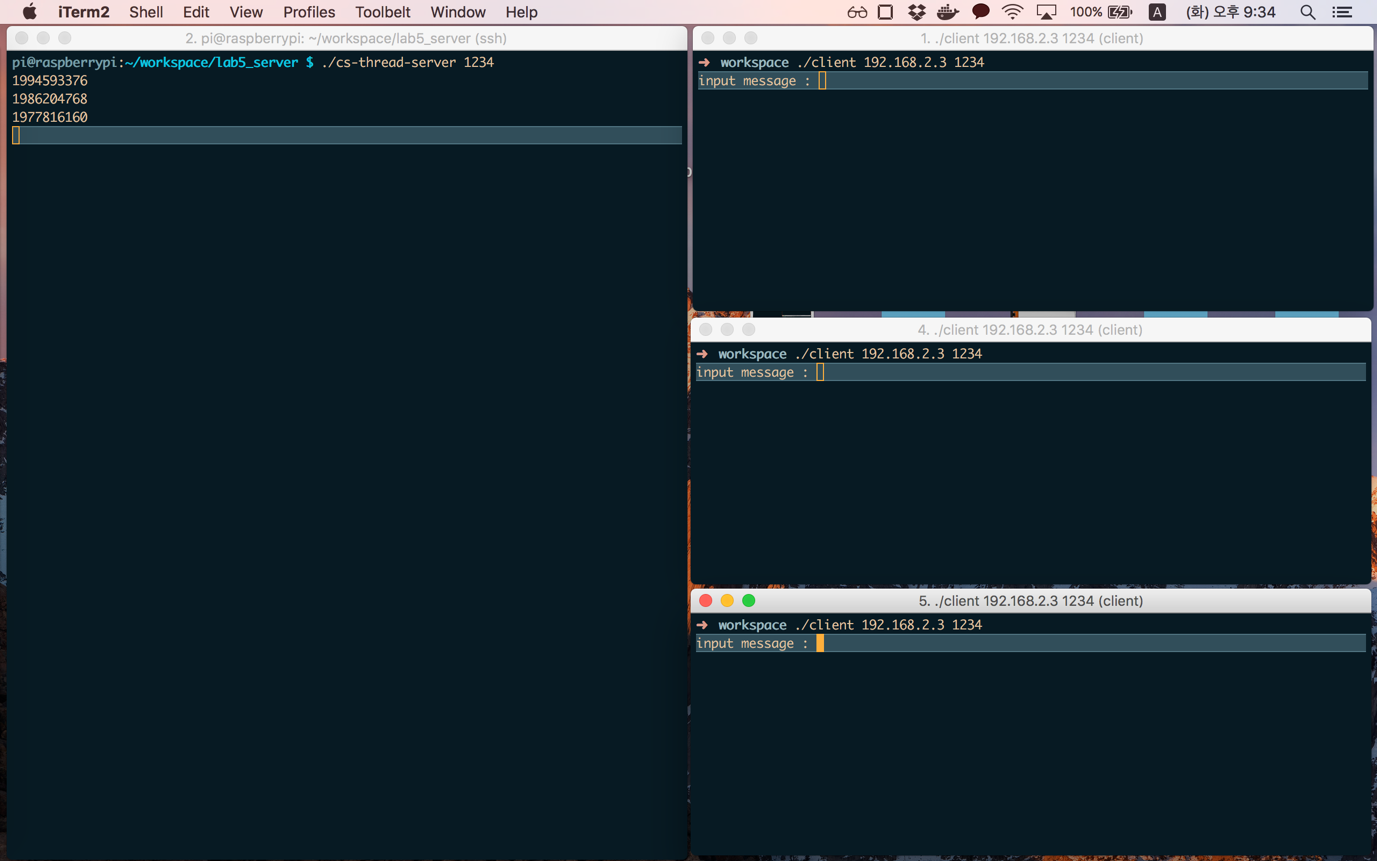
왼쪽 서버에서는, 새로운 connect요청이 생길 때마다 Fork()를 통해 새로운 프로세스를 만들고, pid를 출력한다. 그림에서는 3개의 Client가 connect요청을 하였기 때문에 3번의 fork가 이루어졌음을 확인할 수 있다. 그리고 각 클라이언트 역시 전달한 메시지에 대한 응답을 서버로부터 받을 수 있음을 확인할 수 있다. ‘와이어샤크’의 결과는 다음과 같다.



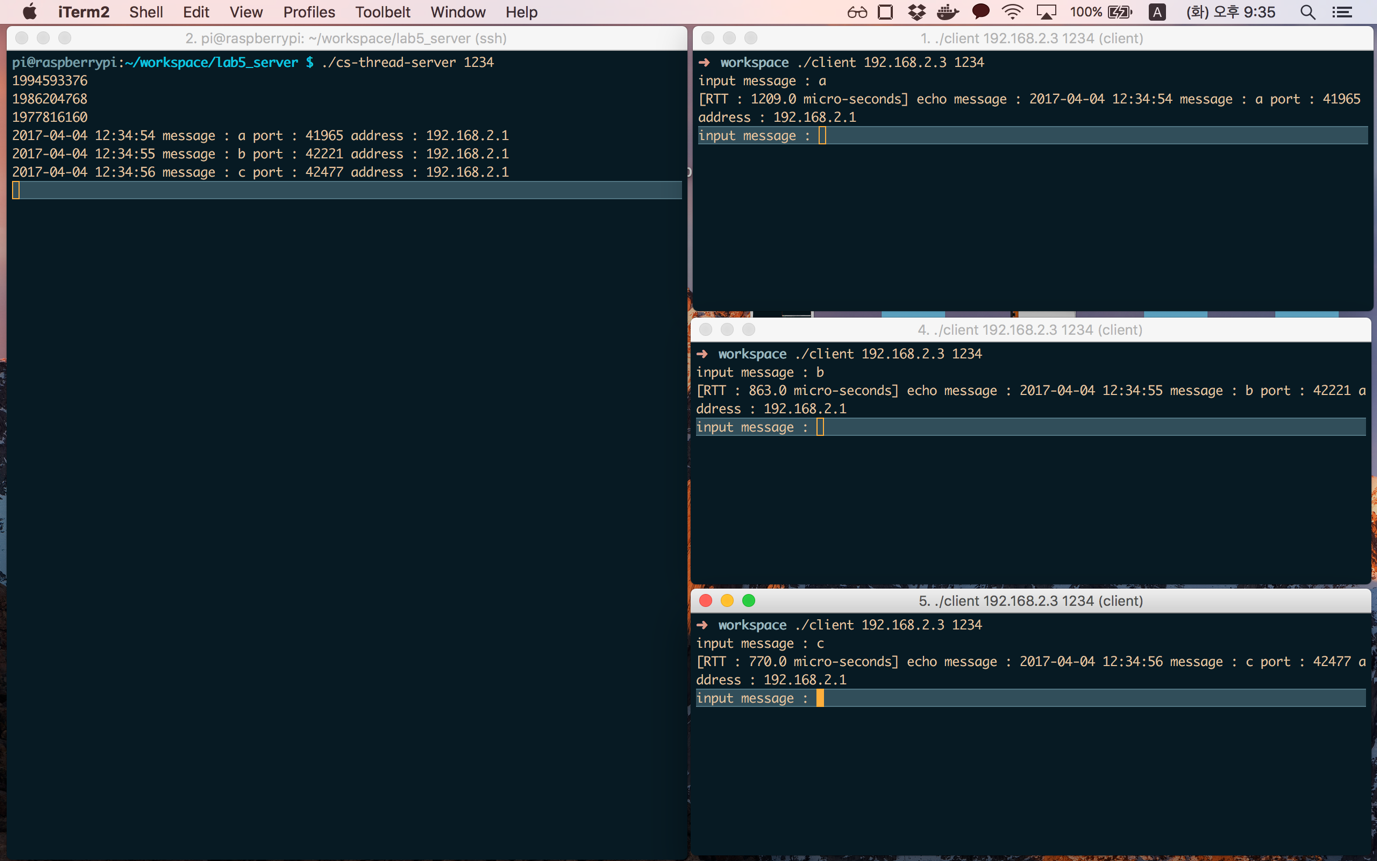
CO Server의 경우, LimitQueue를 ‘1’로 설정하면 TCP retransmission이 발생하였지만 Fork Server의 LimitQueue Size를 ‘1’으로 설정하여 테스트하여도, 위와 같은 결과를 얻을 수 있었다. 이는 Listen과 Accept가 분리되어있기 때문에, 이러한 결과가 도출되었다고 생각한다.

**2-1** Thread CS 구현

Thread CS는 Client의 Connect 요청이 있을 때마다 새로운 Thread를 생성하고, 새롭게 생성된Thread가 accept하여 통신하도록 구현하였다. Fork CS와 마찬가지로, 여러 클라이언트의 Connect 요청을 받을 수 있다.



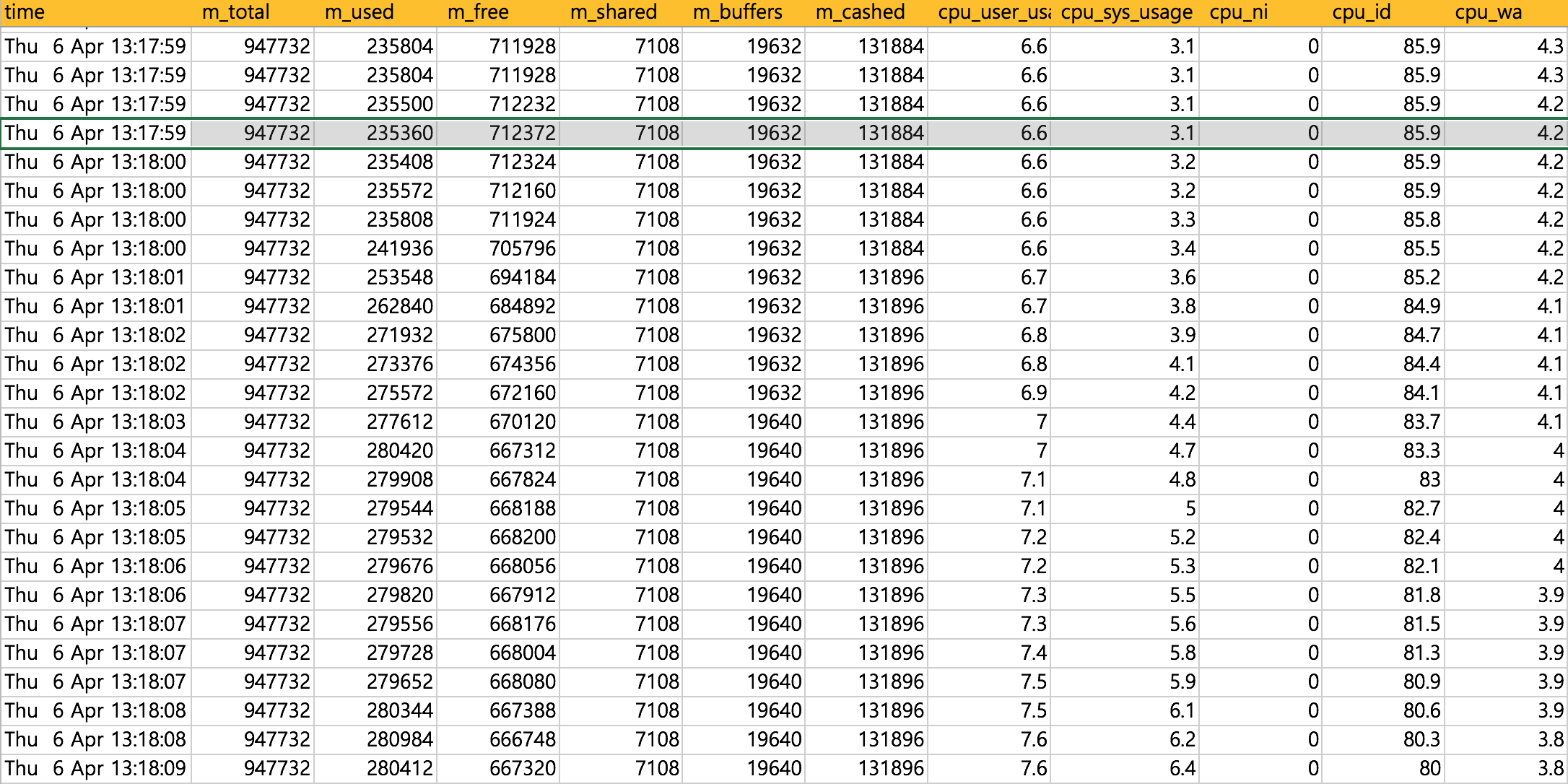
그림의 왼쪽은 Server이다. 오른쪽 Client가 connect요청을 하여 새로운 쓰레드를 생성하면, Server는 생성된 쓰레드의 id를 출력한다. 그리고 각 클라이언트에서 메시지를 전송하면, 서버의 각 쓰레드가 echo메시지를 보냄을 확인할 수 있다.



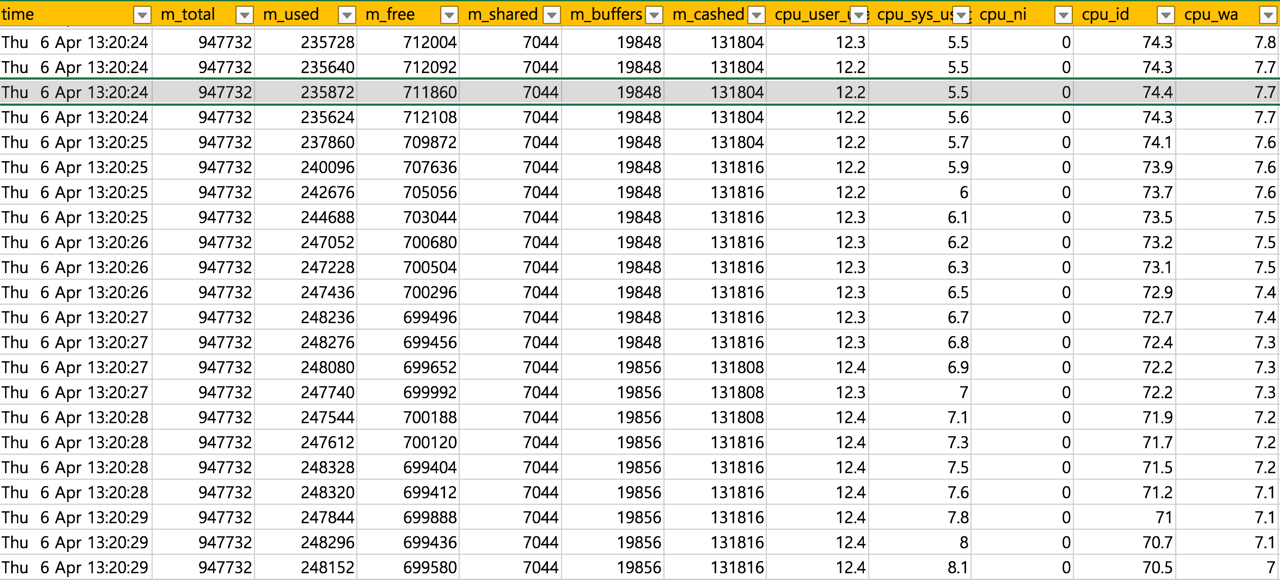
**2-2** fork-CS와 thread-CS 성능 비교

두 서버 간의 비교를 위해, 여러 사전작업이 필요했다. 먼저, 클라이언트를 특정 문자열을 입력하지 않아도 일정 문자열을 계속해서 전송하도록 수정하였다. 또한 쉘 스크립트를 작성하여, 해당 클라이언트를 n개 실행시킬 수 있는 스크립트를 생성하였다. 이때, 첫 번째로 실행시킨 클라이언트는, 매 RTT값을 파일에 쓰고, 이를 저장한다. 서버의 경우, 별도의 ssh-client를 실행시켜, 0.5초 간격으로 현재 memory와 cpu사용량을 측정하도록 하고 이를 csv파일로 저장하였다. 본고에서는 3인이 조를 구성할 수 없어 필자 혼자서 클라이언트 **500**개를 실행시켜 테스트하였다.

먼저 Fork서버 실행중에는 다음과 같은 결과를 보였다.

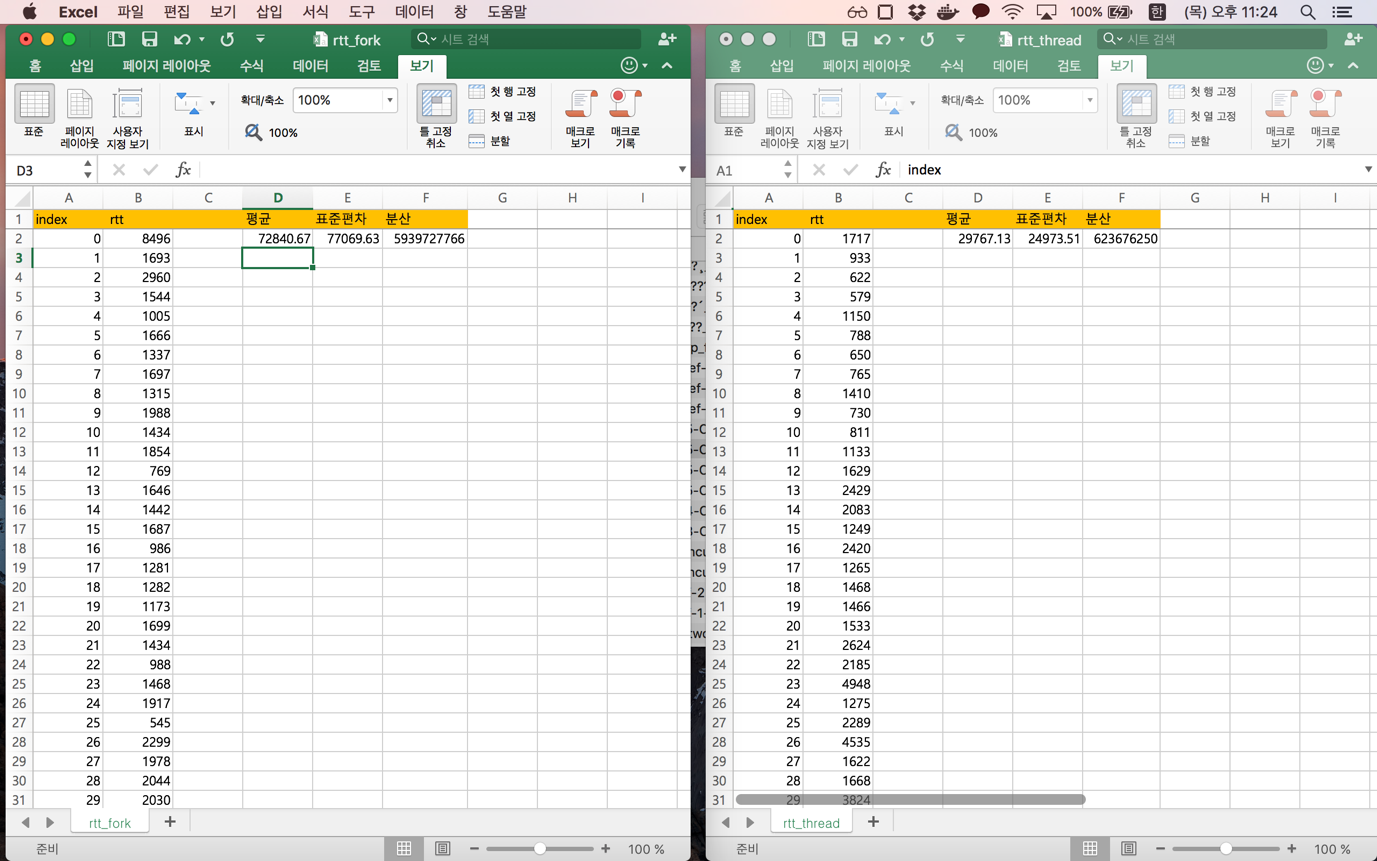


위 그림은, Fork서버를 실행시킨 후, 500개의 클라이언트가 요청을 계속해서 보내기 시작할 때의 표이다. cpu\_sys\_usage의 값이 ‘3.1’보다 커지는 지점이 쉘스크립트를 실행하는 부분인데, 아래에서 살펴볼 Thread서버에 비해 메모리 사용량이 크게 늘어남을 확인할 수 있었다. 또한 유저 프로세스 CPU 사용율 역시 훨씬 더 빠른 속도로 증가하는 것을 확인할 수 있었다. 다음은 Thread 서버의 결과이다.

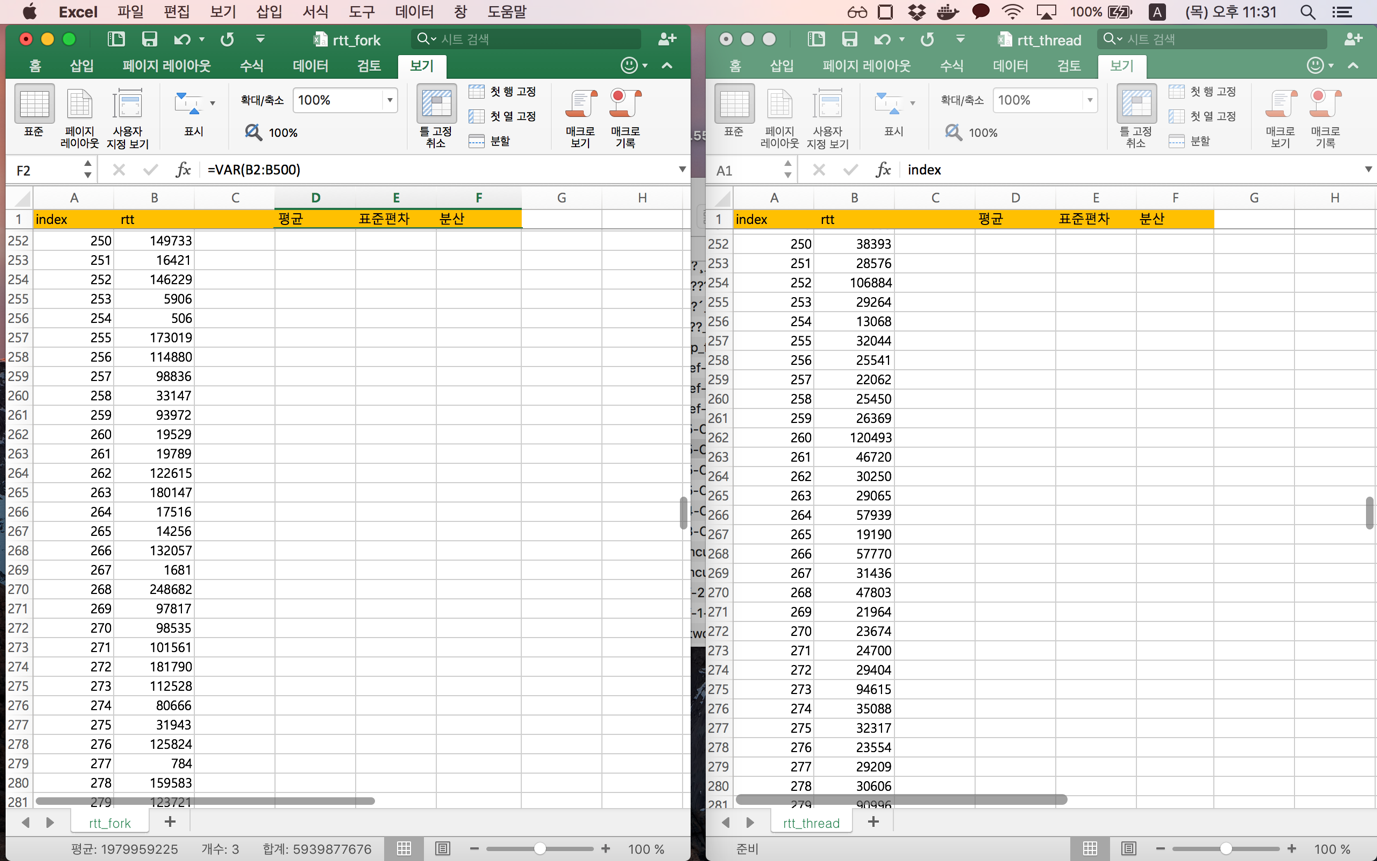


그림에서 cpu\_sys\_usage가 5.5를 넘어가는 부분부터 클라이언트의 요청이 시작되는데, fork서버와 비교하였을 때 확실히 메모리 사용량의 변화가 크지 않음을 확인할 수 있다. 또한 cpu\_user\_usage값. 즉 유저 프로세스 사용율의 증가도 fork에 비해 훨씬 느리다.

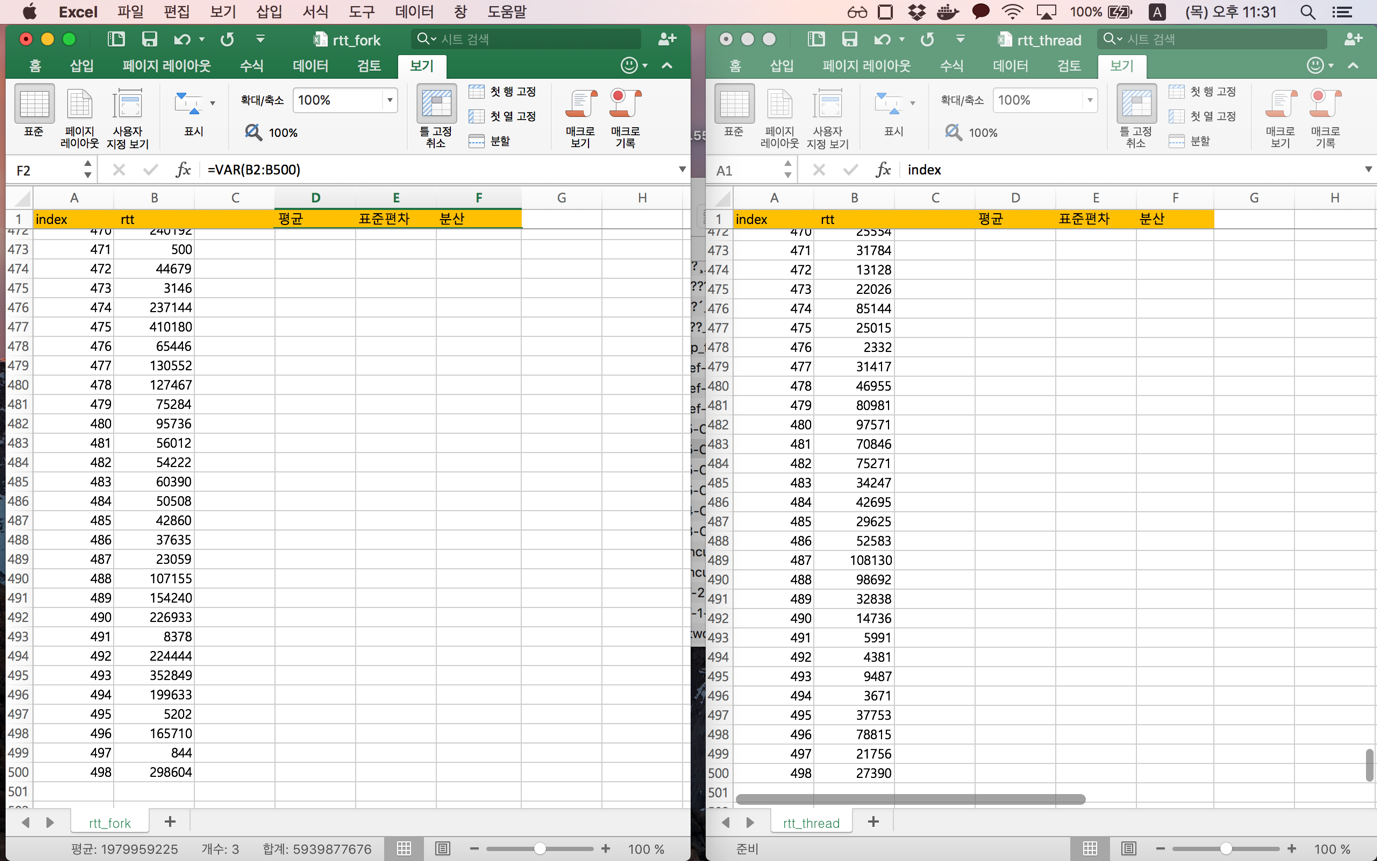
다음으로, 각 서버의 RTT를 비교해보겠다. RTT는 처음, 중간, 끝으로 나누어 각 서버의 RTT를 확인하였다. 먼저, 처음 20개의 RTT와 평균을 보자.



왼쪽은 fork서버, 오른쪽은 thread서버에 대한 RTT이다. Fork서버의 RTT 평균은 72840(micro-seconds)으로, Thread 서버의 평균 RTT인 29767에 비해 약 2.4배가 높다. 또한, 처음 20개 RTT는 비교적 낮음을 확인할 수 있다.



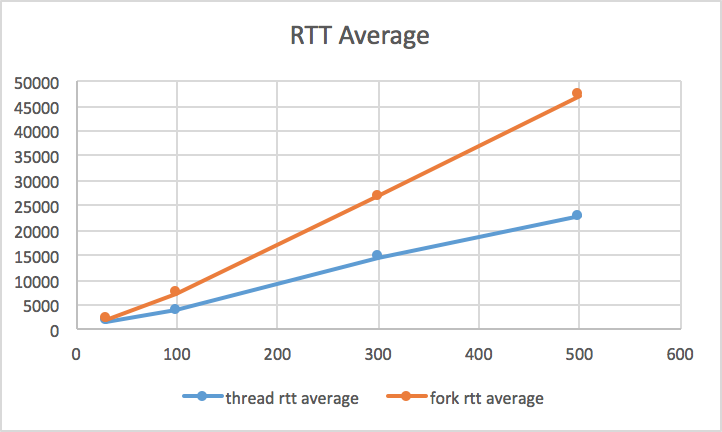
그리고 가장 통신이 활발하게 이루어지는 중간 지점이다. 확실히 초반부에 비해 RTT가 전체적으로 확 올라갔다. 하지만, fork서버에서는 100000, 200000을 넘는 값도 많이 보이는 반면, Thread기반 서버에서는 100000을 넘는 값도 찾기 힘들다.

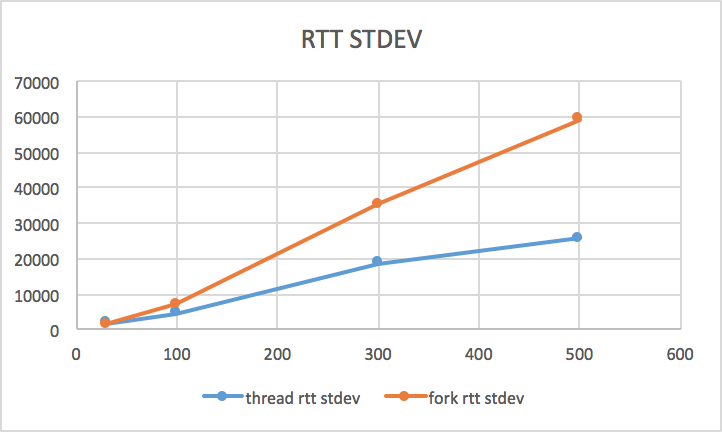


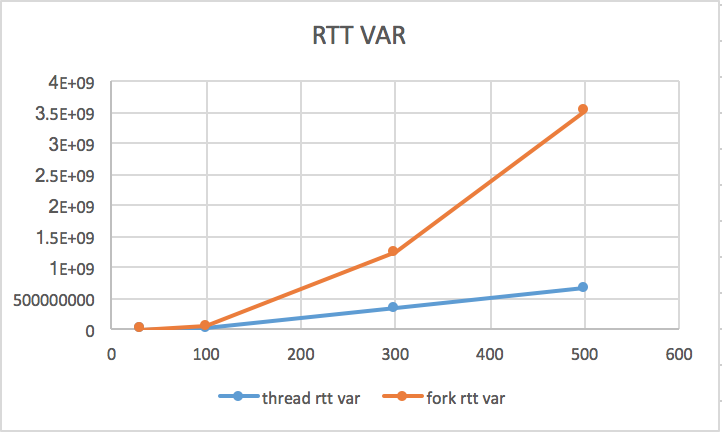
마지막으로, 끝부분이다. 중간부분과 마찬가지로, 둘간의 수 차이가 확연히 드러난다. 때문에, Fork보다는 Thread로 Concurrent Server를 구현하는 것이 이점이 많다고 판단된다.

다음은 클라이언트 수에 따른 RTT, CPU 사용량, 메모리 사용량 각각에 대한 평균, 표준편차, 분산그래프이다. X 축은 클라이언트의 수이며, 필자는 각각 30, 100, 300, 500개의 클라이언트로 실험을 진행하였다.

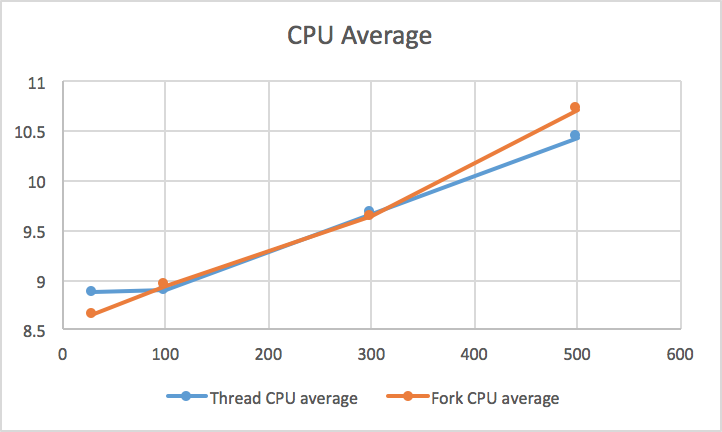
* RTT

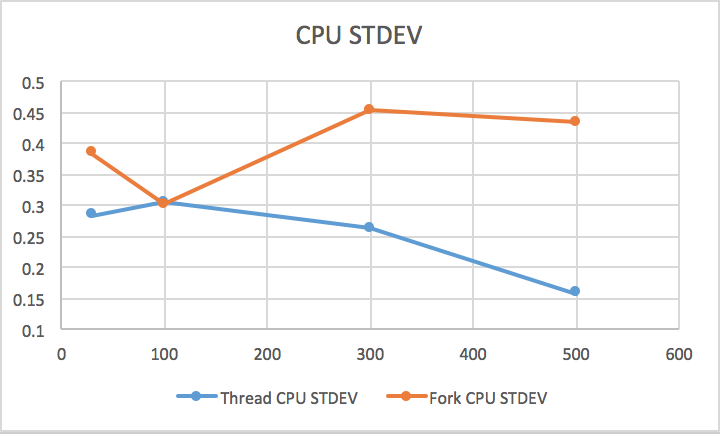


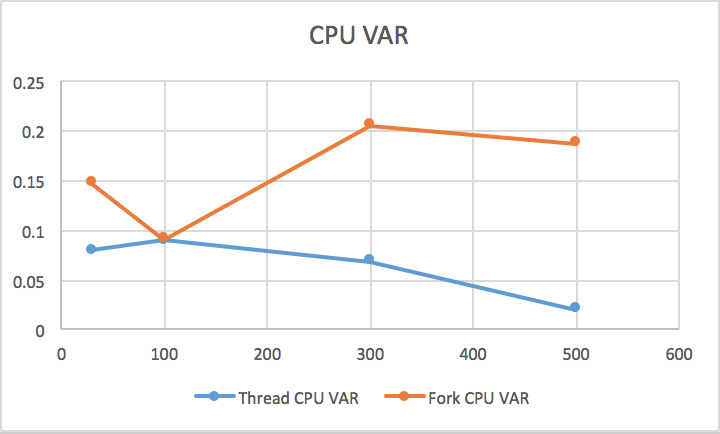




* CPU 사용량







* 메모리 사용량

