

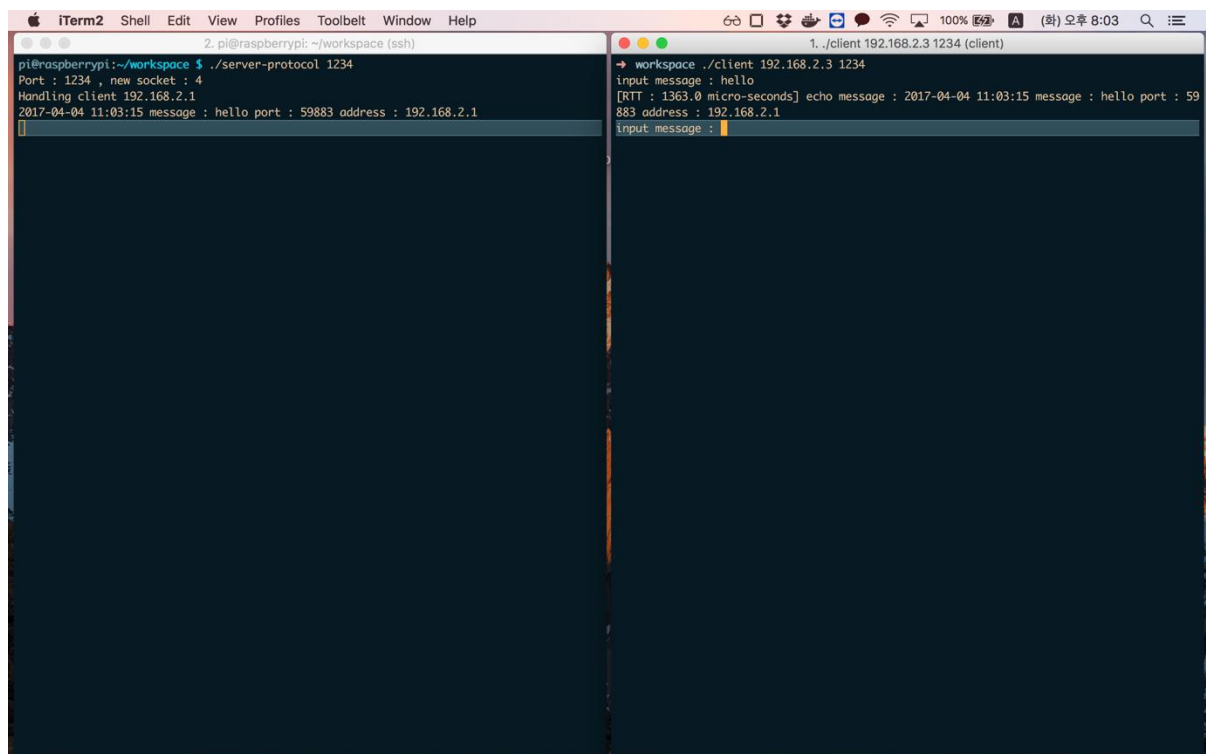
홈 네트워크(공유기 사용)에서 맥북과 라즈베리파이를 이용하여 발생한 결과와 스크린샷을 사용하였음을 밝힌다.

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

### 프로토콜 구현

프로토콜 구현 방법은 다음과 같다. Client 에서 Server 로 메시지를 보내면 Server 는 해당 메시지 내용을 [메시지를 받은 시간] [client address] [client port] [메시지]로 변경하여 이를 print 하고, 해당 메시지를 Client 에 echo 한다. Client 는 메시지를 보내기 전과 후의 시간을 비교하여, RTT 를 계산하여 서버가 보낸 메시지와 RTT 값을 함께 출력한다.

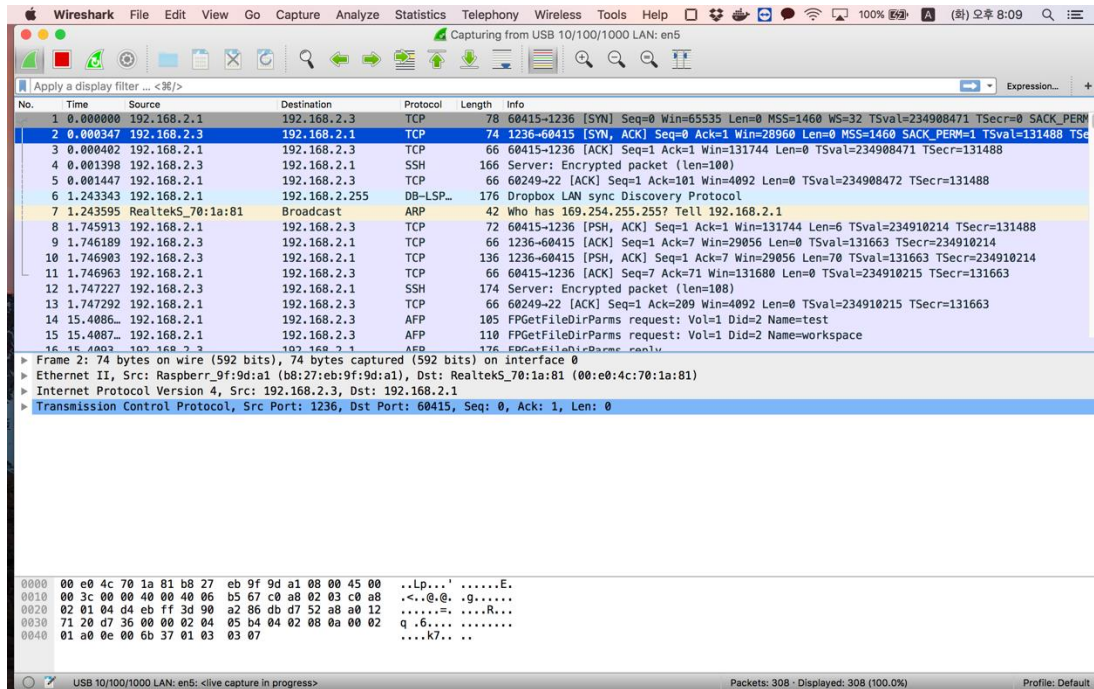
테스트를 위해 먼저, 라즈베리파이에서 CO Echo server 를 port 넘버 1234 로 실행시켰다. 다음으로 맥북에서 클라이언트를 라즈베리파이의 IP 주소(192.168.2.3)와 port 넘버(1234)를 입력하여 실행시킨 후, 메시지를 보냈다. 동작 결과는 다음과 같다.



```
2. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./server-protocol 1234
Port : 1234 , new socket : 4
Handling client 192.168.2.1
2017-04-04 11:03:15 message : hello port : 59883 address : 192.168.2.1

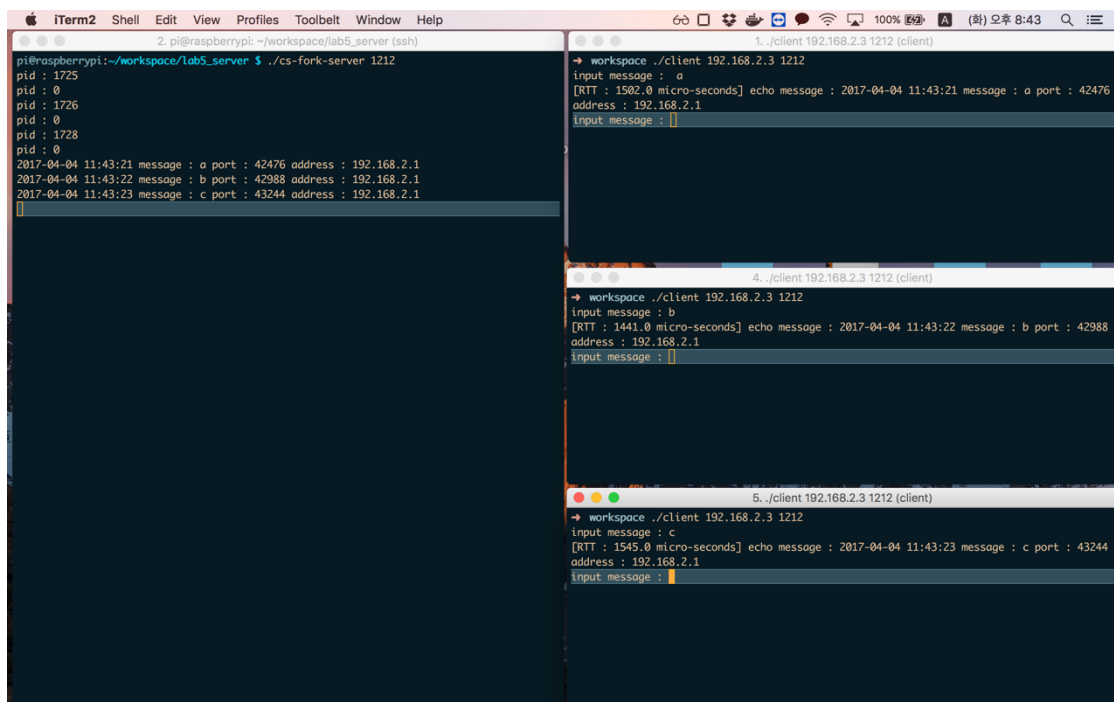
1. ./client 192.168.2.3 1234 (client)
workspace ./client 192.168.2.3 1234
input message : hello
[RTT : 1363.0 micro-seconds] echo message : 2017-04-04 11:03:15 message : hello port : 59883 address : 192.168.2.1
input message :
```

서버가 왼쪽, 오른쪽이 클라이언트이다. 'hello'라는 메시지를 클라이언트에서 전송하자, 서버에서는 메시지에 추가로 클라이언트의 정보와 시간 등을 추가하여 print 한다. 이후 echo 메시지 클라이언트에게 보내면 클라이언트에서 메시지를 받음을 확인할 수 있다. 또한 추가로 RTT 가 micro-second 단위로 출력됨을 확인할 수 있다. '와이어샹크'에서도 정상적으로 connection 이 이루어짐을 확인할 수 있었다.



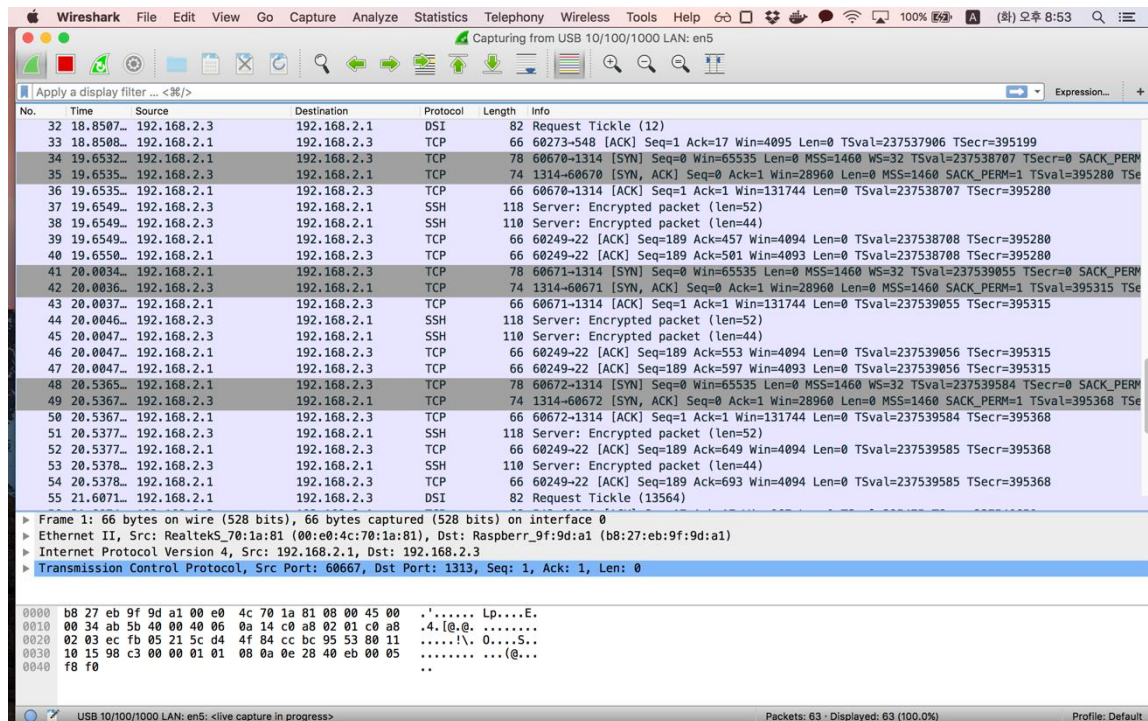
## 1-2 Fork CS 구현

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,

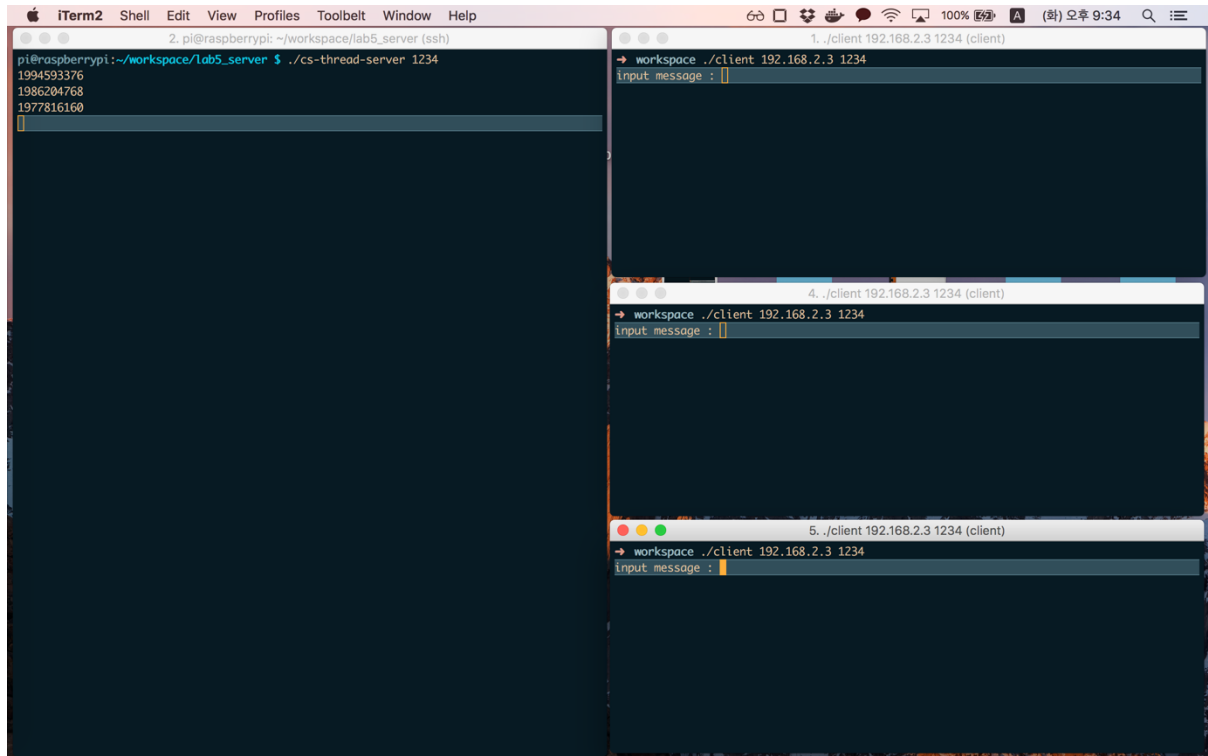
pid를 출력한다. 그림에서는 3개의 Client가 connect요청을 하였기 때문에 3번의 fork가 이루어졌음을 확인할 수 있다. 그리고 각 클라이언트 역시 전달한 메시지에 대한 응답을 서버로부터 받을 수 있음을 확인할 수 있다. '와이어샤크'의 결과는 다음과 같다.



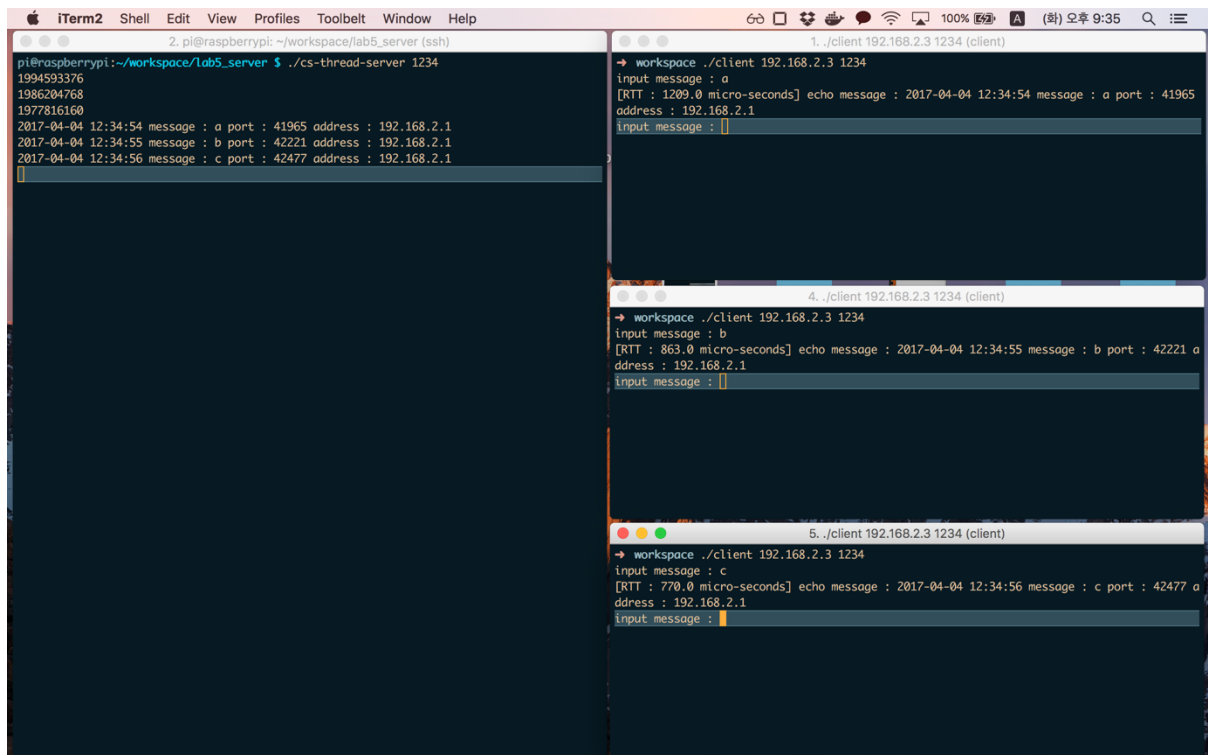
CO Server의 경우, LimitQueue를 '1'로 설정하면 TCP retransmission이 발생하였지만 Fork Server의 LimitQueue Size를 '1'으로 설정하여 테스트하여도, 위와 같은 결과를 얻을 수 있었다. 이는 Listen과 Accept가 분리되어있기 때문에, 이러한 결과가 도출되었다고 생각한다.

## 2-1 Thread CS 구현

Thread CS는 Client의 Connect 요청이 있을 때마다 새로운 Thread를 생성하고, 새롭게 생성된 Thread가 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 보낼 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 3인이조를 구성할 수 없어 필자 혼자서 클라이언트 500개를 실행시켜 테스트하였다.

먼저 Fork서버 실행중에는 다음과 같은 결과를 보였다.

time	m_total	m_used	m_free	m_shared	m_buffers	m_cached	cpu_user_usi	cpu_sys_usage	cpu_ni	cpu_id	cpu_wa
Thu 6 Apr 13:17:59	947732	235804	711928	7108	19632	131884	6.6	3.1	0	85.9	4.3
Thu 6 Apr 13:17:59	947732	235804	711928	7108	19632	131884	6.6	3.1	0	85.9	4.3
Thu 6 Apr 13:17:59	947732	235500	712232	7108	19632	131884	6.6	3.1	0	85.9	4.2
Thu 6 Apr 13:17:59	947732	235360	712372	7108	19632	131884	6.6	3.1	0	85.9	4.2
Thu 6 Apr 13:18:00	947732	235408	712324	7108	19632	131884	6.6	3.2	0	85.9	4.2
Thu 6 Apr 13:18:00	947732	235572	712160	7108	19632	131884	6.6	3.2	0	85.9	4.2
Thu 6 Apr 13:18:00	947732	235808	711924	7108	19632	131884	6.6	3.3	0	85.8	4.2
Thu 6 Apr 13:18:00	947732	241936	705796	7108	19632	131884	6.6	3.4	0	85.5	4.2
Thu 6 Apr 13:18:01	947732	253548	694184	7108	19632	131896	6.7	3.6	0	85.2	4.2
Thu 6 Apr 13:18:01	947732	262840	684892	7108	19632	131896	6.7	3.8	0	84.9	4.1
Thu 6 Apr 13:18:02	947732	271932	675800	7108	19632	131896	6.8	3.9	0	84.7	4.1
Thu 6 Apr 13:18:02	947732	273376	674356	7108	19632	131896	6.8	4.1	0	84.4	4.1
Thu 6 Apr 13:18:02	947732	275572	672160	7108	19632	131896	6.9	4.2	0	84.1	4.1
Thu 6 Apr 13:18:03	947732	277612	670120	7108	19640	131896	7	4.4	0	83.7	4.1
Thu 6 Apr 13:18:04	947732	280420	667312	7108	19640	131896	7	4.7	0	83.3	4
Thu 6 Apr 13:18:04	947732	279908	667824	7108	19640	131896	7.1	4.8	0	83	4
Thu 6 Apr 13:18:05	947732	279544	668188	7108	19640	131896	7.1	5	0	82.7	4
Thu 6 Apr 13:18:05	947732	279532	668200	7108	19640	131896	7.2	5.2	0	82.4	4
Thu 6 Apr 13:18:06	947732	279676	668056	7108	19640	131896	7.2	5.3	0	82.1	4
Thu 6 Apr 13:18:06	947732	279820	667912	7108	19640	131896	7.3	5.5	0	81.8	3.9
Thu 6 Apr 13:18:07	947732	279556	668176	7108	19640	131896	7.3	5.6	0	81.5	3.9
Thu 6 Apr 13:18:07	947732	279728	668004	7108	19640	131896	7.4	5.8	0	81.3	3.9
Thu 6 Apr 13:18:07	947732	279652	668080	7108	19640	131896	7.5	5.9	0	80.9	3.9
Thu 6 Apr 13:18:08	947732	280344	667388	7108	19640	131896	7.5	6.1	0	80.6	3.9
Thu 6 Apr 13:18:08	947732	280984	666748	7108	19640	131896	7.6	6.2	0	80.3	3.8
Thu 6 Apr 13:18:09	947732	280412	667320	7108	19640	131896	7.6	6.4	0	80	3.8

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.



time	m_total	m_used	m_free	m_shared	m_buffers	m_cached	cpu_user_	cpu_sys_u	cpu_ni	cpu_id	cpu_wa
Thu 6 Apr 13:20:24	947732	235728	712004	7044	19848	131804	12.3	5.5	0	74.3	7.8
Thu 6 Apr 13:20:24	947732	235640	712092	7044	19848	131804	12.2	5.5	0	74.3	7.7
Thu 6 Apr 13:20:24	947732	235872	711860	7044	19848	131804	12.2	5.5	0	74.4	7.7
Thu 6 Apr 13:20:24	947732	235624	712108	7044	19848	131804	12.2	5.6	0	74.3	7.7
Thu 6 Apr 13:20:25	947732	237860	709872	7044	19848	131804	12.2	5.7	0	74.1	7.6
Thu 6 Apr 13:20:25	947732	240096	707636	7044	19848	131816	12.2	5.9	0	73.9	7.6
Thu 6 Apr 13:20:25	947732	242676	705056	7044	19848	131816	12.2	6	0	73.7	7.6
Thu 6 Apr 13:20:25	947732	244688	703044	7044	19848	131816	12.3	6.1	0	73.5	7.5
Thu 6 Apr 13:20:26	947732	247052	700680	7044	19848	131816	12.3	6.2	0	73.2	7.5
Thu 6 Apr 13:20:26	947732	247228	700504	7044	19848	131816	12.3	6.3	0	73.1	7.5
Thu 6 Apr 13:20:26	947732	247436	700296	7044	19848	131816	12.3	6.5	0	72.9	7.4
Thu 6 Apr 13:20:27	947732	248236	699496	7044	19848	131816	12.3	6.7	0	72.7	7.4
Thu 6 Apr 13:20:27	947732	248276	699456	7044	19848	131816	12.3	6.8	0	72.4	7.3
Thu 6 Apr 13:20:27	947732	248080	699652	7044	19856	131808	12.4	6.9	0	72.2	7.3
Thu 6 Apr 13:20:27	947732	247740	699992	7044	19856	131808	12.3	7	0	72.2	7.3
Thu 6 Apr 13:20:28	947732	247544	700188	7044	19856	131808	12.4	7.1	0	71.9	7.2
Thu 6 Apr 13:20:28	947732	247612	700120	7044	19856	131816	12.4	7.3	0	71.7	7.2
Thu 6 Apr 13:20:28	947732	248328	699404	7044	19856	131816	12.4	7.5	0	71.5	7.2
Thu 6 Apr 13:20:28	947732	248320	699412	7044	19856	131816	12.4	7.6	0	71.2	7.1
Thu 6 Apr 13:20:29	947732	247844	699888	7044	19856	131816	12.4	7.8	0	71	7.1
Thu 6 Apr 13:20:29	947732	248296	699436	7044	19856	131816	12.4	8	0	70.7	7.1
Thu 6 Apr 13:20:29	947732	248152	699580	7044	19856	131816	12.4	8.1	0	70.5	7

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.

index	rtt	평균	표준편차	분산
1	0	8496	72840.67	77069.63
2	1	1693		5939727766
3	2	2960		
4	3	1544		
5	4	1005		
6	5	1666		
7	6	1337		
8	7	1697		
9	8	1315		
10	9	1988		
11	10	1434		
12	11	1854		
13	12	769		
14	13	1646		
15	14	1442		
16	15	1687		
17	16	986		
18	17	1281		
19	18	1282		
20	19	1173		
21	20	1699		
22	21	1434		
23	22	988		
24	23	1468		
25	24	1917		
26	25	545		
27	26	2299		
28	27	1978		
29	28	2044		
30	29	2030		

index	rtt	평균	표준편차	분산
1	0	1717	29767.13	24973.51
2	1	933		623676250
3	2	622		
4	3	579		
5	4	1150		
6	5	788		
7	6	650		
8	7	765		
9	8	1410		
10	9	730		
11	10	811		
12	11	1133		
13	12	1629		
14	13	2429		
15	14	2083		
16	15	1249		
17	16	2420		
18	17	1265		
19	18	1468		
20	19	1466		
21	20	1533		
22	21	2624		
23	22	2185		
24	23	4948		
25	24	1275		
26	25	2289		
27	26	4535		
28	27	1622		
29	28	1668		
30	29	3824		

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.

index	rtt	평균	표준편차	분산
252	250	149733		
253	251	16421		
254	252	146229		
255	253	5906		
256	254	506		
257	255	173019		
258	256	114880		
259	257	98836		
260	258	33147		
261	259	93972		
262	260	19529		
263	261	19789		
264	262	122615		
265	263	180147		
266	264	17516		
267	265	14256		
268	266	132057		
269	267	1681		
270	268	248682		
271	269	97817		
272	270	98535		
273	271	101561		
274	272	181790		
275	273	112528		
276	274	80666		
277	275	31943		
278	276	125824		
279	277	784		
280	278	159583		
281	279	129721		

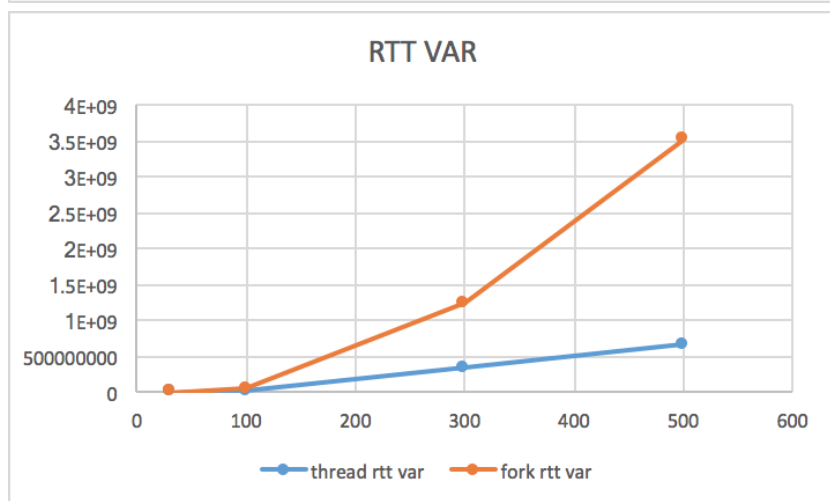
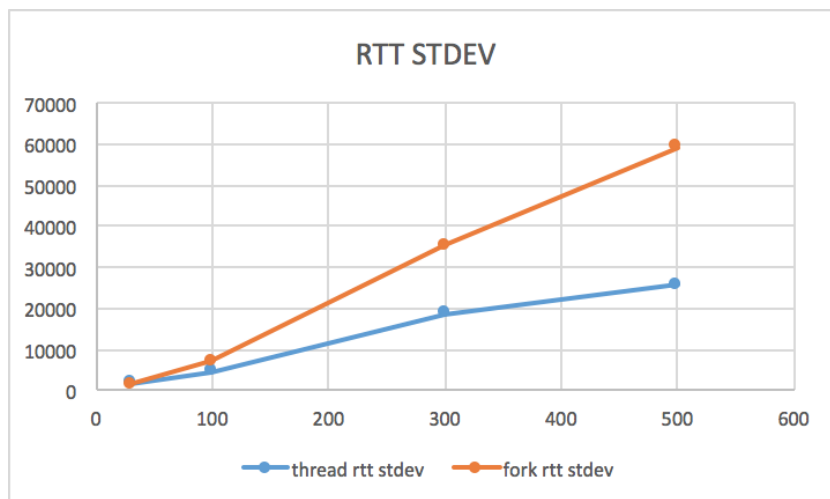
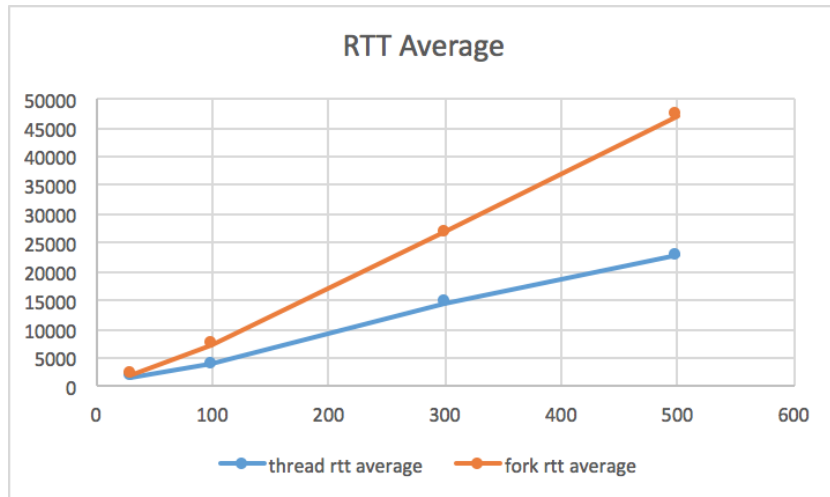
그리고 가장 통신이 활발하게 이루어지는 중간 지점이다. 확실히 초반부에 비해 RTT가 전체적으로 확 올라갔다. 하지만, fork서버에서는 100000, 200000을 넘는 값도 많이 보이는 반면, Thread 기반 서버에서는 100000을 넘는 값도 찾기 힘들다.

index	rtt	평균	표준편차	분산
470	240132			
471	500			
472	44679			
473	3146			
474	237144			
475	410180			
476	65446			
477	130552			
478	127467			
479	75284			
480	95736			
481	56012			
482	54222			
483	60390			
484	50508			
485	42860			
486	37635			
487	23059			
488	107155			
489	154240			
490	226933			
491	8378			
492	224444			
493	352849			
494	199633			
495	5202			
496	165710			
497	844			
498	298604			

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

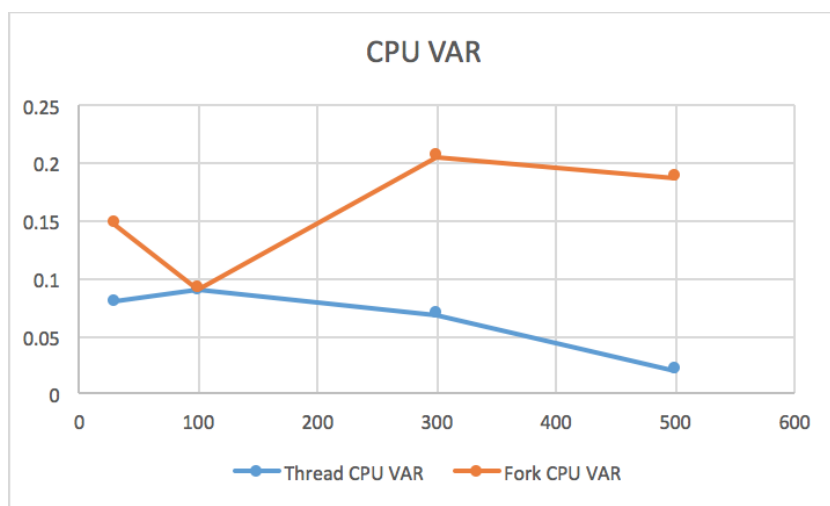
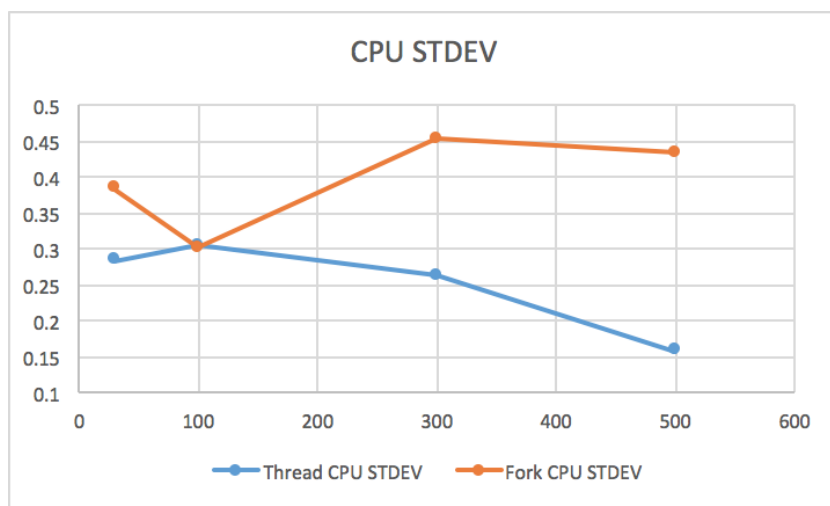
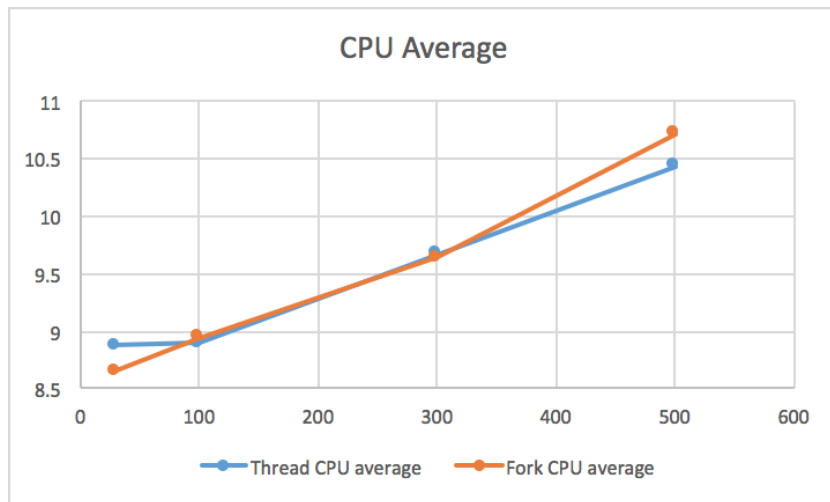
다음은 클라이언트 수에 따른 RTT, CPU 사용량, 메모리 사용량 각각에 대한 평균, 표준편차, 분산 그래프이다. X 축은 클라이언트의 수이며, 필자는 각각 30, 100, 300, 500개의 클라이언트로 실험을 진행하였다.

- RTT





- CPU 사용량



- 메모리 사용량

