

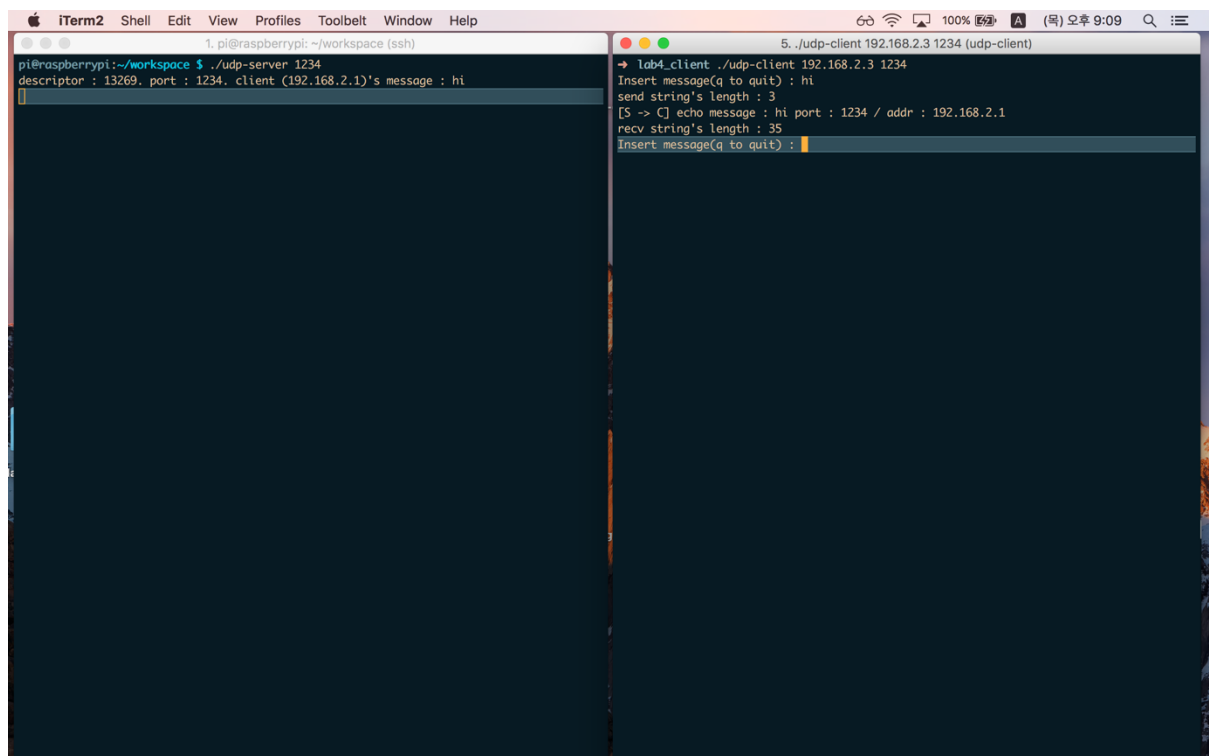
홈 네트워크(공유기 사용)에서 맥북과 라즈베리파이를 이용하여 발생한 결과와 스크린샷을 사용하였음.

## 1. 개요

세부적인 동작 결과 보고에 앞서, CL Echo Client, Server 이 대략적으로 어떻게 동작하는지를 먼저 설명하고자 한다. Client 에서는 UDP 로 데이터를 보낼 서버에 해당하는 IP 주소와, port 넘버를 입력하여 실행시킬 수 있다. 그리고 보낼 메시지를 입력하여 서버에 해당 메시지를 보낼 수 있다. 이때, 'q'를 입력하면 Client 의 socket close 가 발생하며, 'quit&quit'을 입력하면 Server 와 Client 의 소켓이 모두 close 된다. 서버는 해당 메시지를 받은 후, 문자열과 client 의 IP 주소, port 넘버를 클라이언트에게 다시 전송하게 된다. 앞에서도 설명하였듯이 만약 클라이언트에서 보낸 메시지가 quit&quit 인 경우, echo 를 정상적으로 실행시킨 후 socket close 를 시도한다.

### 1-1 CL Echo Client-Server

먼저, 라즈베리파이에서 CL 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 보내어 테스트를 진행하였다. 처음에는, 'hi'메시지를 보내었다. 동작 결과는 아래와 같다.

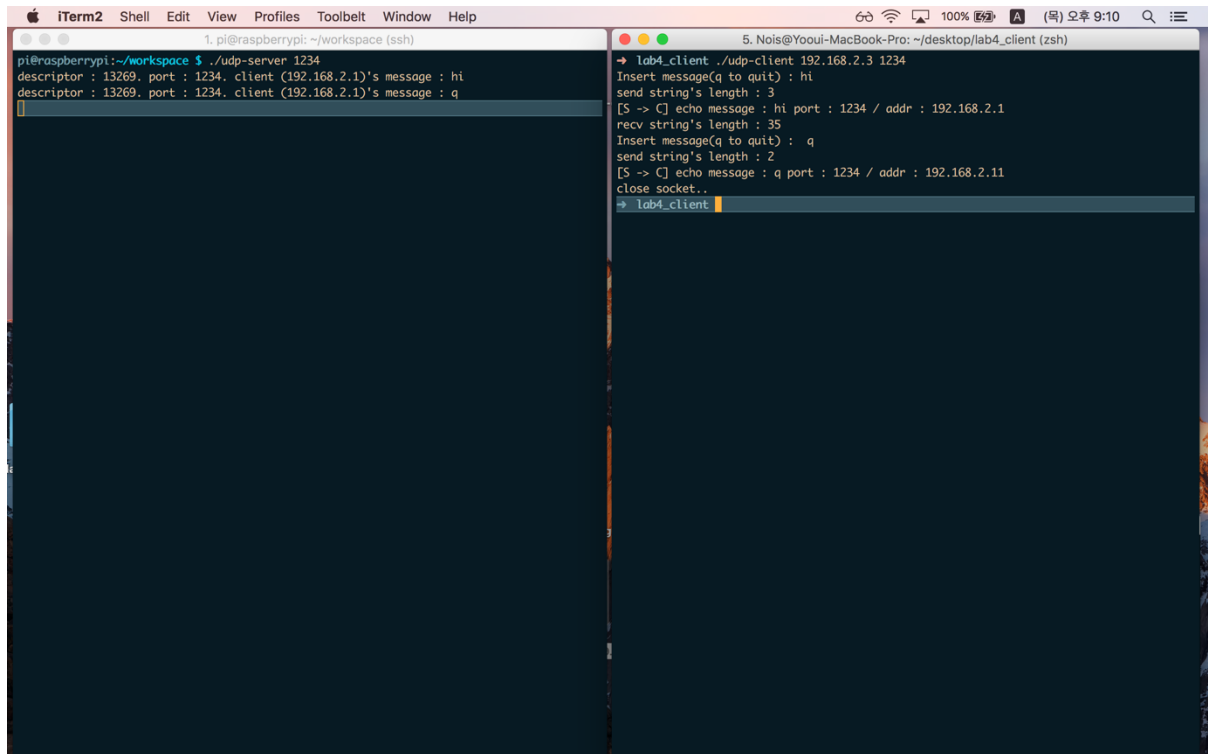


```
iTerm2 Shell Edit View Profiles Toolbelt Window Help
1. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./udp-server 1234
descriptor : 13269, port : 1234, client (192.168.2.1)'s message : hi

5. ./udp-client 192.168.2.3 1234 (udp-client)
→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : hi
send string's length : 3
[S -> C] echo message : hi port : 1234 / addr : 192.168.2.1
recv string's length : 35
Insert message(q to quit) :
```

그림의 왼쪽이 서버이고, 오른쪽이 클라이언트이다. 'hi'라는 메시지를 클라이언트에서 전송하자, 서버에서는 192.168.2.1. 즉, 맥북의 ip 주소에 해당하는 클라이언트에서 hi 라는 메시지가 전송되었음을 알리고, 메시지에 추가로 클라이언트의 정보를 추가하여 정상적으로 echo 메시지를

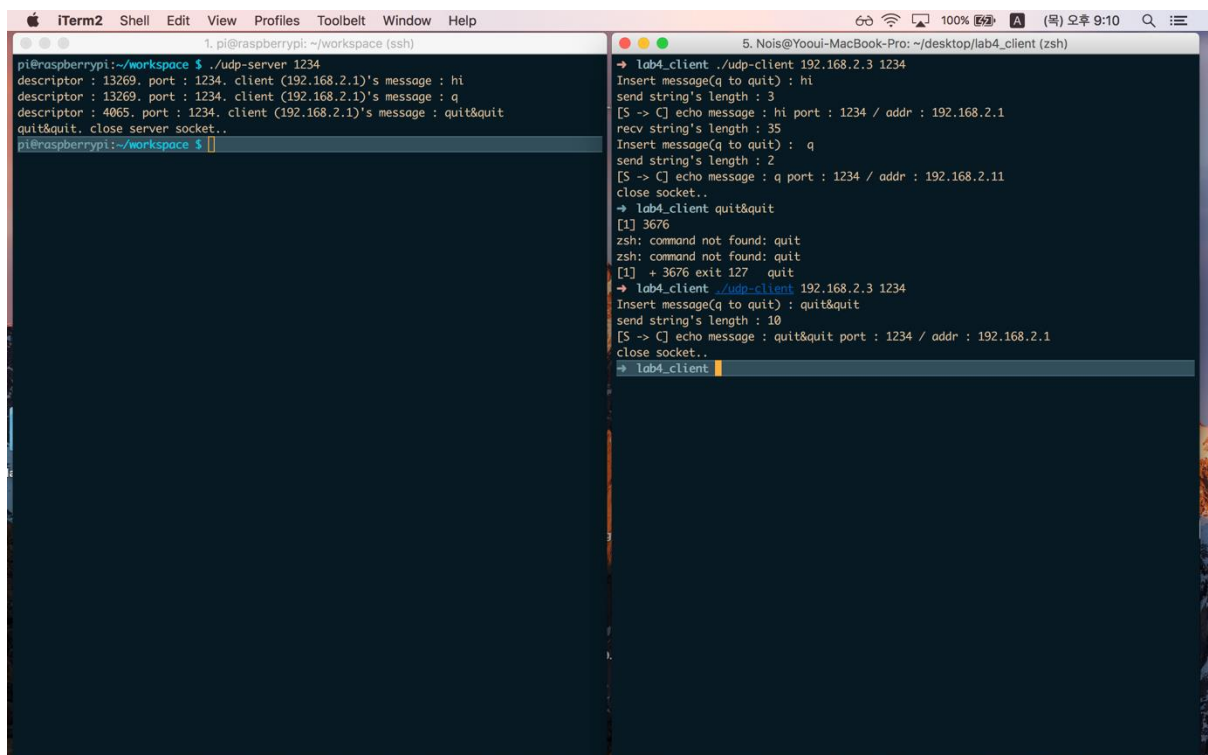
보낸다. 따라서 클라이언트에서는 제대로 메시지를 받음을 확인할 수 있다. 다음은, 'q' 메시지를 보내는 모습이다.



```
1. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./udp-server 1234
descriptor : 13269, port : 1234, client (192.168.2.1)'s message : hi
descriptor : 13269, port : 1234, client (192.168.2.1)'s message : q

5. Nois@Yooi-MacBook-Pro: ~/desktop/lab4_client (zsh)
→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : hi
send string's length : 3
[S -> C] echo message : hi port : 1234 / addr : 192.168.2.1
recv string's length : 35
Insert message(q to quit) : q
send string's length : 2
[S -> C] echo message : q port : 1234 / addr : 192.168.2.11
close socket..
→ lab4_client
```

역시 정상적으로 메시지를 주고 받은 후, Client 에서만 socket close 가 일어났다. 마지막으로, 'quit&quit' 메시지를 보내보았다.



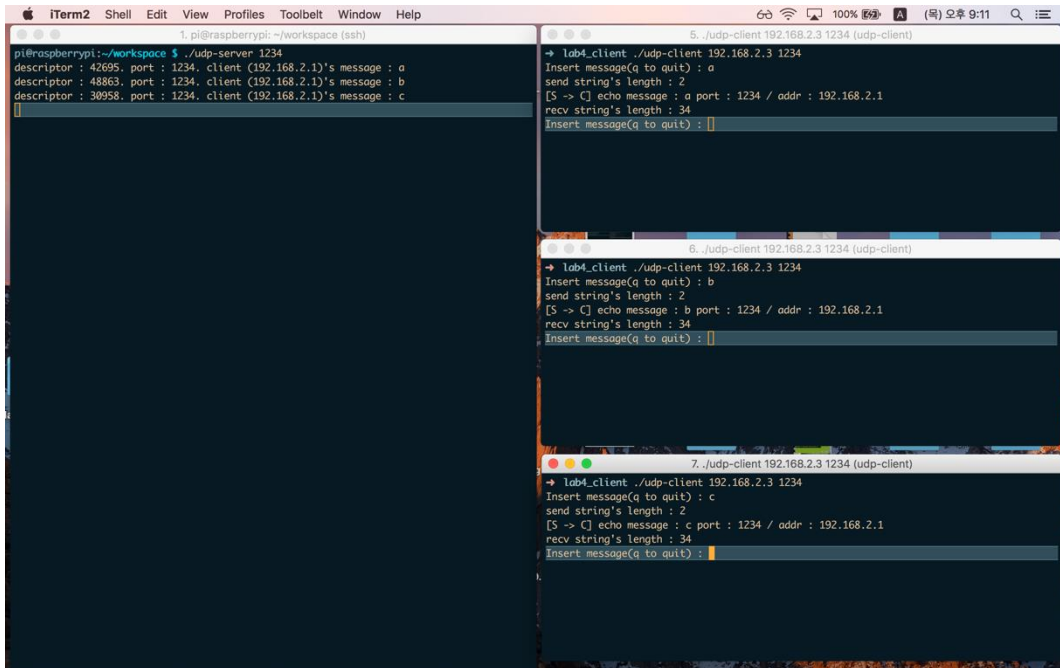
```
1. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./udp-server 1234
descriptor : 13269, port : 1234, client (192.168.2.1)'s message : hi
descriptor : 13269, port : 1234, client (192.168.2.1)'s message : q
descriptor : 4065, port : 1234, client (192.168.2.1)'s message : quit&quit
quit&quit, close server socket..
pi@raspberrypi:~/workspace $

5. Nois@Yooi-MacBook-Pro: ~/desktop/lab4_client (zsh)
→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : hi
send string's length : 3
[S -> C] echo message : hi port : 1234 / addr : 192.168.2.1
recv string's length : 35
Insert message(q to quit) : q
send string's length : 2
[S -> C] echo message : q port : 1234 / addr : 192.168.2.11
close socket..
→ lab4_client quit&quit
[1] 3676
zsh: command not found: quit
zsh: command not found: quit
[1] + 3676 exit 127 quit
→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : quit&quit
send string's length : 10
[S -> C] echo message : quit&quit port : 1234 / addr : 192.168.2.1
close socket..
→ lab4_client
```

이번에는 서버와 클라이언트에서 모두 socket close 가 일어남을 확인할 수 있다.

## 1-2 CL Echo Client-Server

이번에는, 1-1과 서버는 동일한 조건에서 클라이언트만 여럿 실행시켜 진행하였다. CO의 경우, 처음 연결된 클라이언트만 메시지를 주고받을 수 있었는데, CO의 특성상 모든 클라이언트가 메시지를 주고받을 수 있을 것이라 예상했다. 그리고 실제로는 과연 어떤 결과가 나올지 궁금했다.



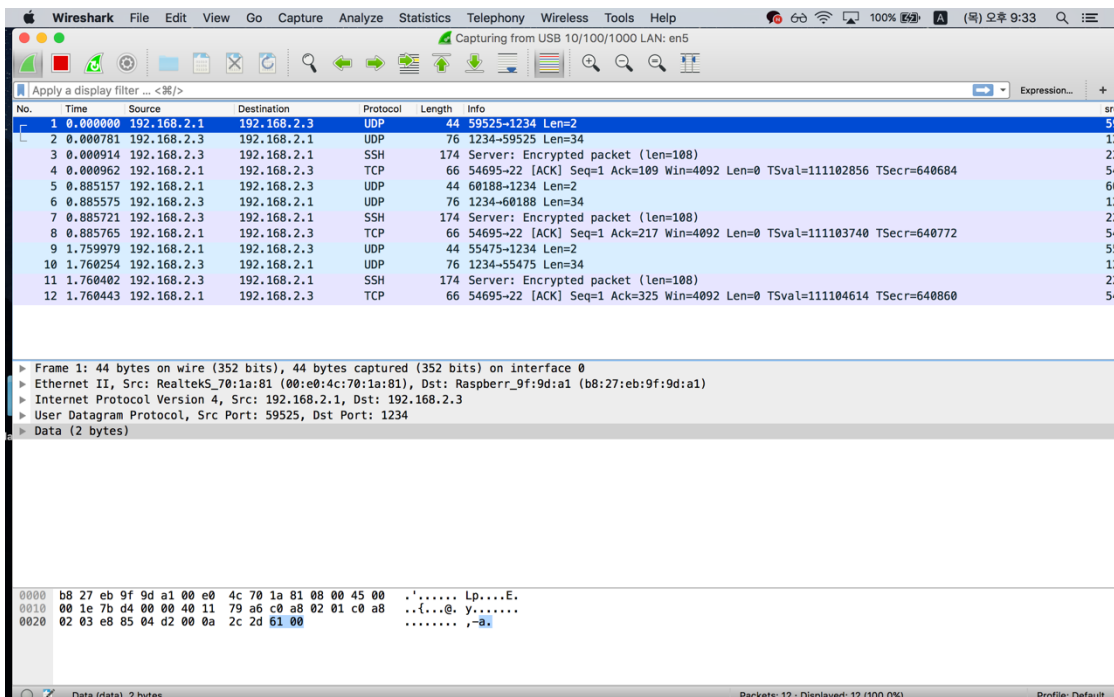
```
pi@raspberrypi:~/workspace $ ./udp-server 1234
descriptor : 42695, port : 1234, client (192.168.2.1)'s message : a
descriptor : 48863, port : 1234, client (192.168.2.1)'s message : b
descriptor : 30958, port : 1234, client (192.168.2.1)'s message : c

→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : a
send string's length : 2
[S -> C] echo message : a port : 1234 / addr : 192.168.2.1
recv string's length : 34
Insert message(q to quit) : []

→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : b
send string's length : 2
[S -> C] echo message : b port : 1234 / addr : 192.168.2.1
recv string's length : 34
Insert message(q to quit) : []

→ lab4_client ./udp-client 192.168.2.3 1234
Insert message(q to quit) : c
send string's length : 2
[S -> C] echo message : c port : 1234 / addr : 192.168.2.1
recv string's length : 34
Insert message(q to quit) : []
```

1-1과 같은 조건에서, 먼저 각 클라이언트에서 한 번씩 메시지를 보내었다. (왼쪽이 서버이고, 오른쪽이 클라이언트이다.) 예상대로 모든 클라이언트에서 echo 메시지를 정상적으로 받을 수 있었다. 또한 여기서, 각 프로세스마다 다른 socket descriptor값을 가지고 있는 점을 확인할 수 있었다. 와이어샤크에서는 다음과 같이 패킷이 캡쳐되었다.

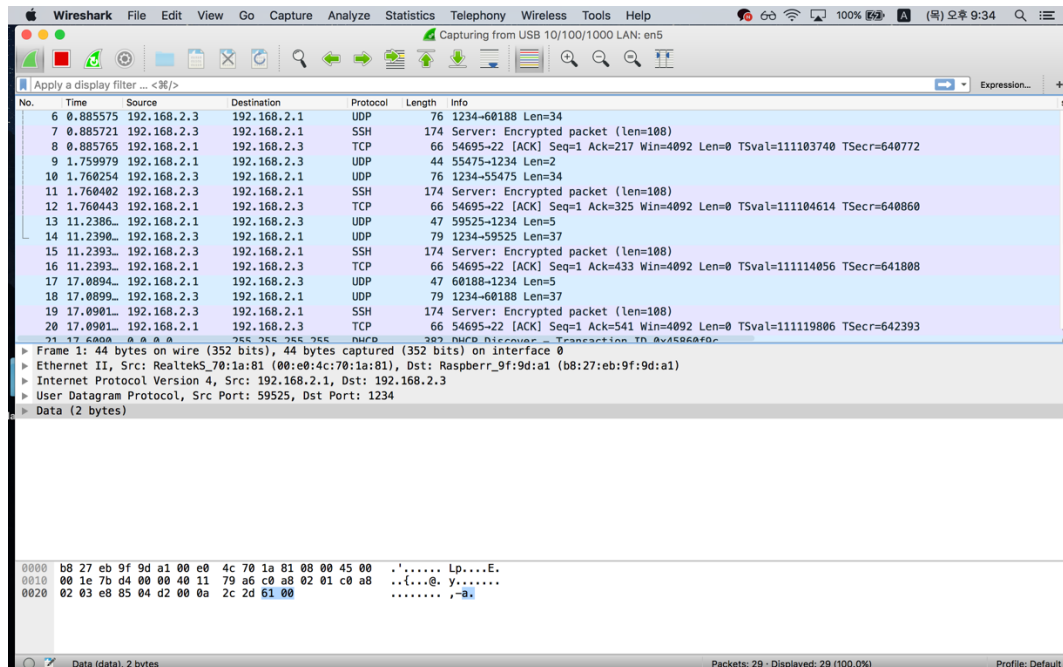


No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.2.1	192.168.2.3	UDP	44	59525-1234 Len=2
2	0.000781	192.168.2.3	192.168.2.1	UDP	76	1234-59525 Len=34
3	0.000914	192.168.2.1	192.168.2.3	SSH	174	Server: Encrypted packet (len=108)
4	0.000962	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=109 Win=4092 Len=0 TSval=111102856 TSecr=640684
5	0.085157	192.168.2.1	192.168.2.3	UDP	44	60188-1234 Len=2
6	0.085575	192.168.2.3	192.168.2.1	UDP	76	1234-60188 Len=34
7	0.085721	192.168.2.3	192.168.2.1	SSH	174	Server: Encrypted packet (len=108)
8	0.085765	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=217 Win=4092 Len=0 TSval=111103740 TSecr=640772
9	1.759979	192.168.2.1	192.168.2.3	UDP	44	55475-1234 Len=2
10	1.760254	192.168.2.3	192.168.2.1	UDP	76	1234-55475 Len=34
11	1.760402	192.168.2.3	192.168.2.1	SSH	174	Server: Encrypted packet (len=108)
12	1.760443	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=325 Win=4092 Len=0 TSval=111104614 TSecr=640860

Frame 1: 44 bytes on wire (352 bits), 44 bytes captured (352 bits) on interface 0  
Ethernet II, Src: RealtekS\_70:1a:81 (00:e0:4c:70:1a:81), Dst: Raspberr\_9f:9d:a1 (b8:27:eb:9f:9d:a1)  
Internet Protocol Version 4, Src: 192.168.2.1, Dst: 192.168.2.3  
User Datagram Protocol, Src Port: 59525, Dst Port: 1234  
Data (2 bytes)

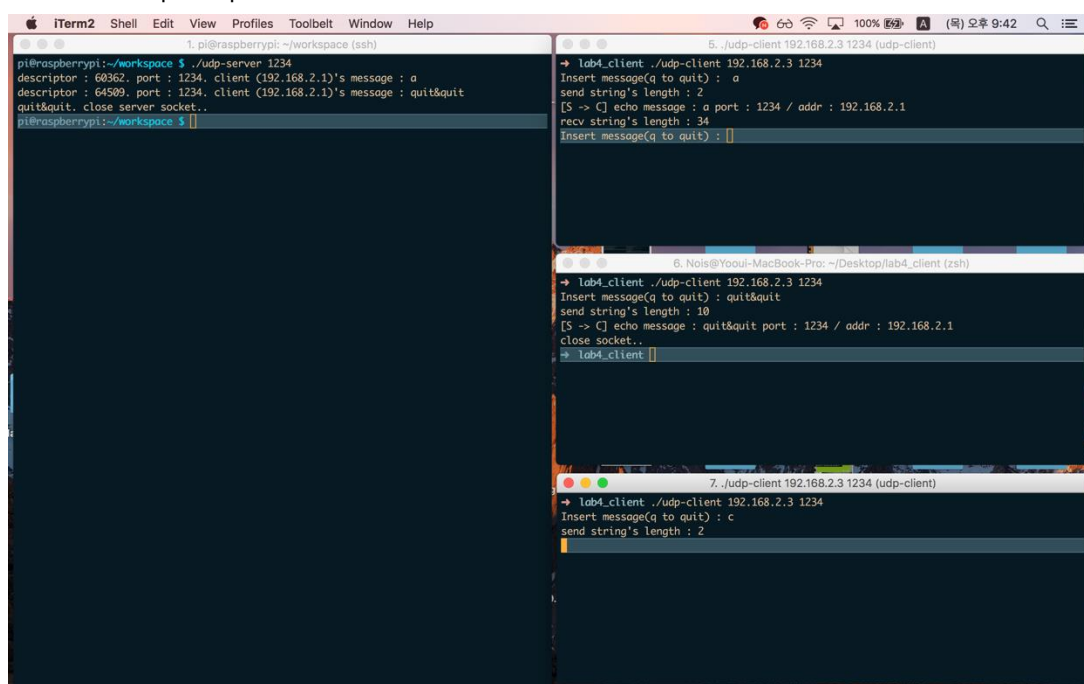
0000 b8 27 eb 9f 9d a1 00 e0 4c 70 1a 81 08 00 45 00 .'. . . . . Lp . . . E.  
0010 00 1e 7b d4 00 00 40 11 79 a6 c0 a8 02 01 c0 a8 . . . . . @. y . . . . .  
0020 02 03 e8 85 04 d2 00 0a 2c 2d 61 00 . . . . . - 8 .

Client(192.168.2.1)에서 Server(192.168.2.3)으로 UDP 패킷을 보내고 있음을 확인할 수 있었다. 해당 패킷의 Data는 'a'임을 확인할 수 있다. 바로 아래의 UDP패킷은 Server에서 Client로 보내는 ECHO메시지에 해당하며, 보다 아래에 위치한 UDP 패킷들은 다른 프로세스에서 보낸 데이터에 해당한다. 다음으로는 'q'메시지를 보내어, 각각 Client의 소켓을 닫았다. 결과는 다음과 같다.



## 1-3 CL Echo Client-Server

1-2와 동일한 조건으로 진행하였고, 각 클라이언트에서 차례로 메시지를 전송하되, 두 번째 Client에서만 'quit&quit' 메시지를 전송하였다.



세 번째 클라이언트에서는 메시지를 전송한 후 1-2와는 달리, echo 메시지를 받지 못함을 확인할 수 있다. 와어샤크에서는 다음은 결과를 확인할 수 있었다.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	0.0.0.0	255.255.255.255	DHCP	382	DHCP Discover - Transaction ID 0x45860f9c
2	0.446094	192.168.2.1	192.168.2.3	UDP	44	51947-1234 Len=2
3	0.447055	192.168.2.3	192.168.2.1	UDP	76	1234-51947 Len=34
4	0.447075	192.168.2.3	192.168.2.1	SSH	174	Server: Encrypted packet (len=108)
5	0.447163	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=109 Win=4092 Len=0 TSval=111822938 TSecr=712860
6	4.828430	192.168.2.1	192.168.2.3	UDP	52	65335-1234 Len=10
7	4.828998	192.168.2.3	192.168.2.1	UDP	84	1234-65335 Len=42
8	4.829575	192.168.2.3	192.168.2.1	SSH	214	Server: Encrypted packet (len=148)
9	4.829656	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=257 Win=4091 Len=0 TSval=111827319 TSecr=713299
10	4.832820	192.168.2.3	192.168.2.1	SSH	190	Server: Encrypted packet (len=124)
11	4.832878	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=381 Win=4092 Len=0 TSval=111827322 TSecr=713299
12	6.313427	192.168.2.1	192.168.2.3	UDP	44	58597-1234 Len=2
13	6.313783	192.168.2.3	192.168.2.1	ICMP	72	Destination unreachable (Port unreachable)

여기서 주목할만한 부분은, 바로 12번 패킷과 13번 패킷이다. 12번 패킷은 세 번째 Client에서 Server에게 데이터를 전송하는 패킷이다. 그리고, 13번 패킷은 ICMP(Internet Control Management Protocol)인데, 설명하기에 앞서 먼저 해당 패킷을 자세히 보도록 하겠다.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	0.0.0.0	255.255.255.255	DHCP	382	DHCP Discover - Transaction ID 0x45860f9c
2	0.446094	192.168.2.1	192.168.2.3	UDP	44	51947-1234 Len=2
3	0.447055	192.168.2.3	192.168.2.1	UDP	76	1234-51947 Len=34
4	0.447075	192.168.2.3	192.168.2.1	SSH	174	Server: Encrypted packet (len=108)
5	0.447163	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=109 Win=4092 Len=0 TSval=111822938 TSecr=712860
6	4.828430	192.168.2.1	192.168.2.3	UDP	52	65335-1234 Len=10
7	4.828998	192.168.2.3	192.168.2.1	UDP	84	1234-65335 Len=42
8	4.829575	192.168.2.3	192.168.2.1	SSH	214	Server: Encrypted packet (len=148)
9	4.829656	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=257 Win=4091 Len=0 TSval=111827319 TSecr=713299
10	4.832820	192.168.2.3	192.168.2.1	SSH	190	Server: Encrypted packet (len=124)
11	4.832878	192.168.2.1	192.168.2.3	TCP	66	54695->22 [ACK] Seq=1 Ack=381 Win=4092 Len=0 TSval=111827322 TSecr=713299
12	6.313427	192.168.2.1	192.168.2.3	UDP	44	58597-1234 Len=2
13	6.313783	192.168.2.3	192.168.2.1	ICMP	72	Destination unreachable (Port unreachable)

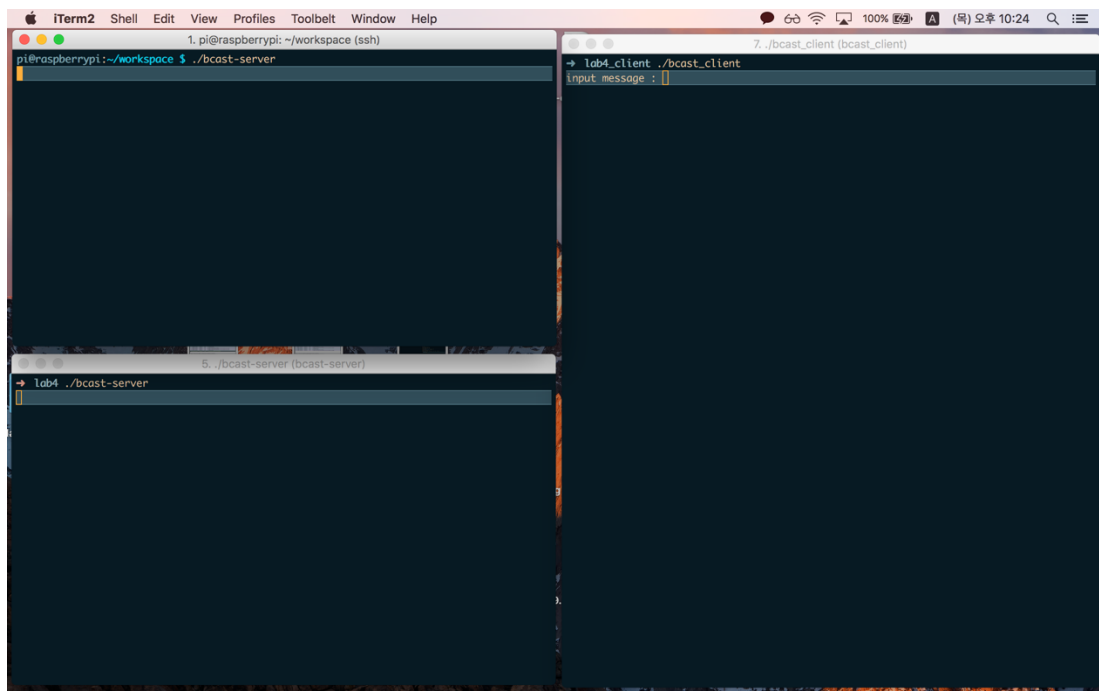
해당 ICMP패킷의 헤더에는 Type과 Code값이 존재하는데 Type은 오류의 원인을, Code는 구체적인 이유를 설명한다. 수집된 패킷의 type, code값은 각각 3, 3이다. "type = 3"은 "목적지 미도착 (Destination unreachable)"을 의미하고 "Code = 3"은 지정된 Transport layer protocol(TCP 혹은 UDP)가 데이터그램을 개별화할 수 없고, 송신자에게 알릴 수 있는 다른 프로토콜 메커니즘을 가지고 있지 않은 경우라고 한다. Code의 의미는 정확히 모르겠지만, Port unreachable에서 그 의미를 추상적으로나마 파악할 수 있다. 때문에, 클라이언트는 ICMP패킷을 통해 자신이 보낸 데이터가 정상적으로 전송되지 않았음을 알고, ICMP헤더로부터 그 원인과 이유를 확인할 수 있다는 것을 확인할 수 있었다.

## 2. 개요

Broadcast Programming에서도 세부적인 동작 결과 보고에 앞서, Broadcast Client, Server이 대략적으로 어떻게 동작하는지를 먼저 설명하고자 한다. Client에서는 코드 내부에 port넘버를 '9099'로 지정하여 해당 포트로 Local broadcast한다. (255.255.255.255로 메시지를 보낸다.) 그리고, 클라이언트에서는 학번을 로컬 변수로 가지고있다. 그리고 메시지를 보낼 때 마다, 메시지 내용과 함께 일련번호와 학번을 보낸다. 서버를 실행시키면 해당 포트로 broadcast 메시지를 대기하는 상태가 된다.

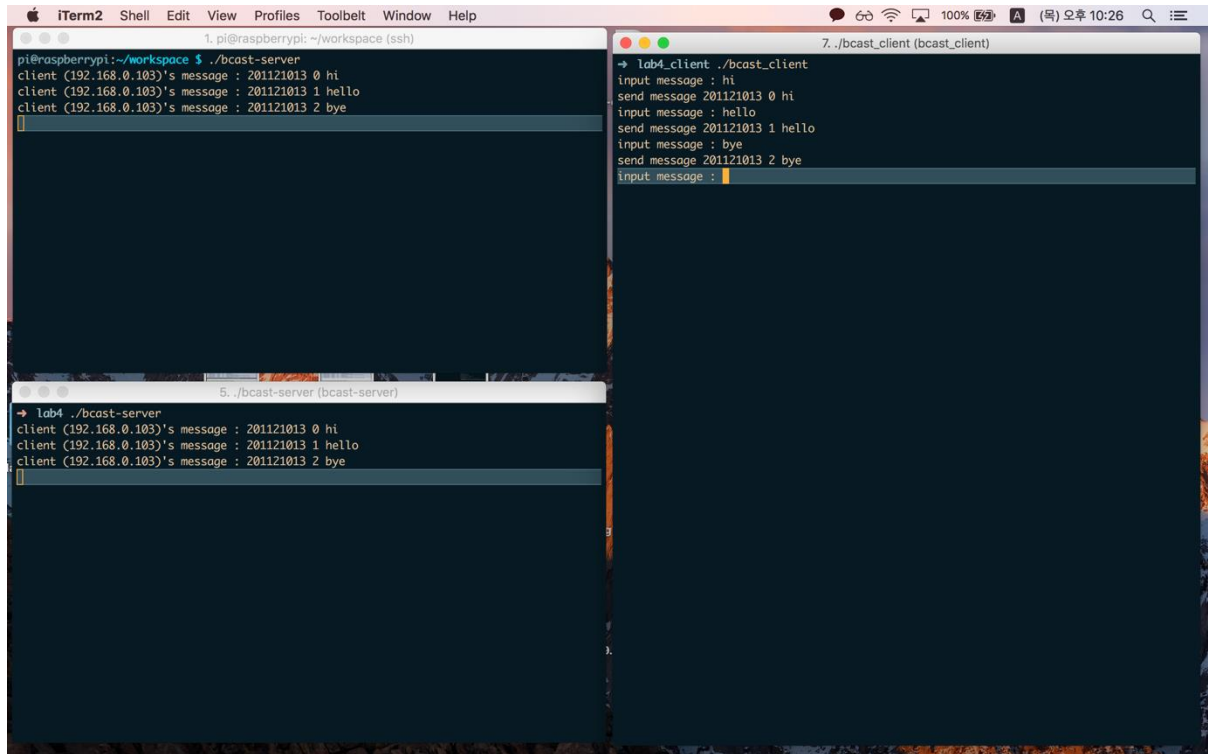
### 2-1 Broadcast Programming

먼저, 라즈베리파이와 맥북에서 각각 서버를 하나씩 실행시키고, 맥북에서 클라이언트를 실행시켰다.





왼쪽 위는 라즈베리파이에서 실행시킨 서버이며, 왼쪽 아래는 맥북에서 실행시킨 서버이다. 그리고, 오른쪽은 클라이언트이다. 다음으로, 클라이언트에서 각각 'hi', 'hello', 'bye'의 순으로 메시지를 보내었다.



```
1. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./bcast-server
client (192.168.0.103)'s message : 201121013 0 hi
client (192.168.0.103)'s message : 201121013 1 hello
client (192.168.0.103)'s message : 201121013 2 bye

5. ./bcast-server (bcast-server)
→ lab4 ./bcast-server
client (192.168.0.103)'s message : 201121013 0 hi
client (192.168.0.103)'s message : 201121013 1 hello
client (192.168.0.103)'s message : 201121013 2 bye

7. ./bcast_client (bcast_client)
→ lab4_client ./bcast_client
input message : hi
send message 201121013 0 hi
input message : hello
send message 201121013 1 hello
input message : bye
send message 201121013 2 bye
input message :
```

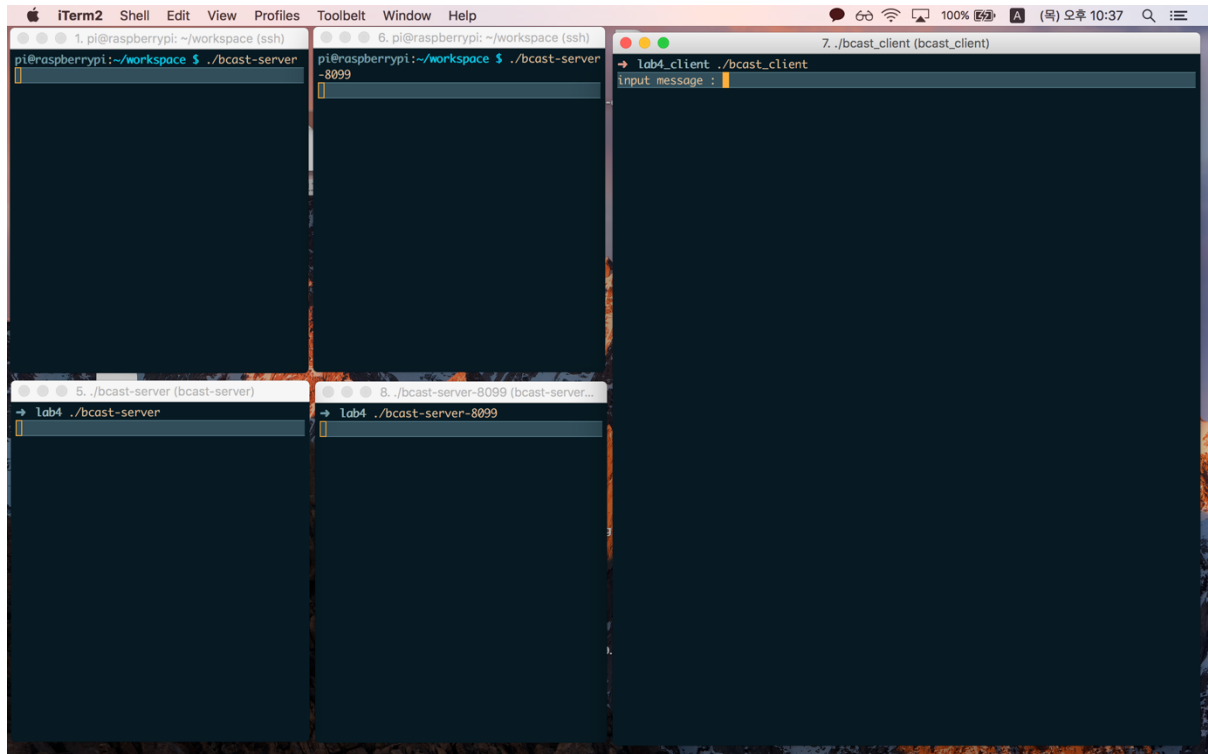
각 서버에서 정상적으로 해당 메시지를 전달받았음을 확인할 수 있다.

## 2-2 Broadcast Programming

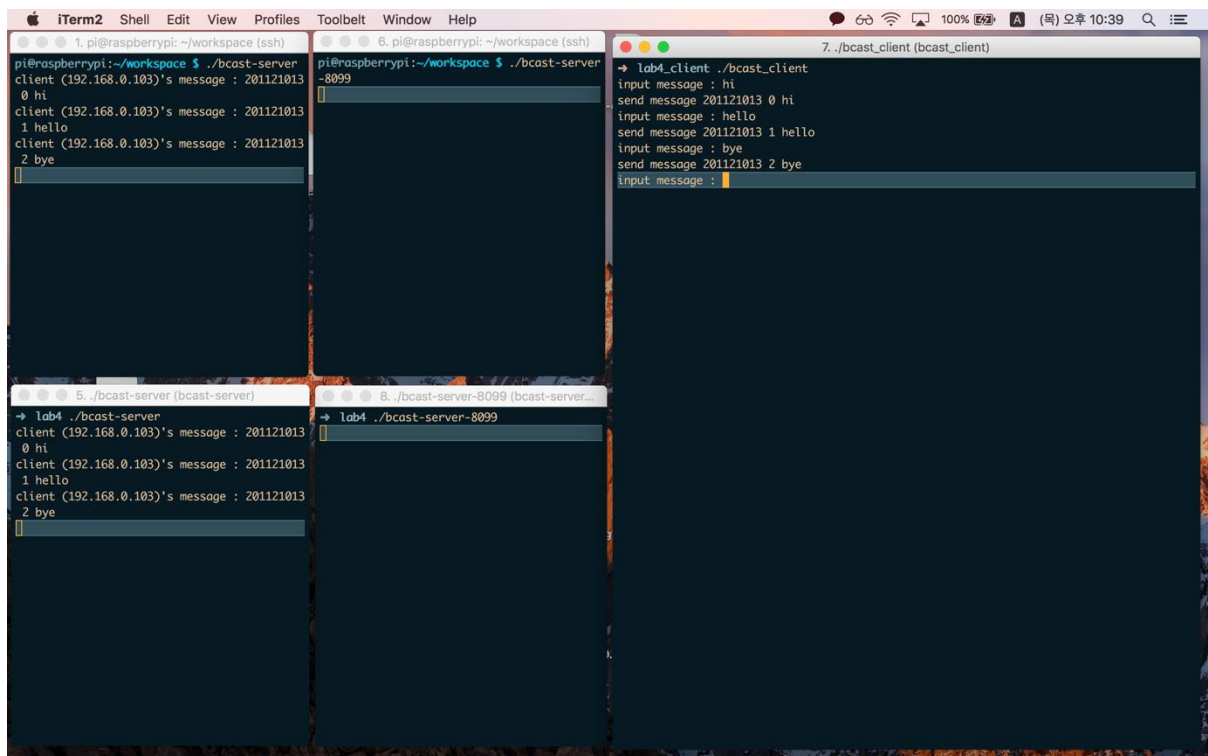
해당 실습의 경우, 실습실에서 진행하여 결과를 확인하였으나 여러 머신을 확보할 수 없어 보고서 작성시에는 제대로 테스트할 수 없었다. 예상 결과는 '같은 로컬 네트워크 내에 있으므로 정상적으로 모든 Server에서 메시지를 수신할 수 있을 것이다.'였다. 실제로 실습 시 서버에서는 Client의 ip주소와 해당 클라이언트가 보낸 학번, 일련번호, 문자열이 조합된 형태로 메시지를 받을 수 있었다.

## 2-3 Broadcast Programming

역시 2-2와 마찬가지로, 여러 머신을 확보할 수 없어 로컬에서 port넘버를 바꾸어 진행하였다. 먼저, 라즈베리파이와 맥북에서 각각 9099, 8099 port넘버로 총 4개의 서버를 실행시킨 후, 클라이언트에서는 9099 port넘버로 local broadcast를 하여 테스트하였다. 예상 결과는 클라이언트에서 9099 port넘버로 broadcast하므로 9099 port넘버로 실행시킨 서버만 메시지를 받을 것이라고 예상하였다.



왼쪽 상단은 라즈베리파이에서 9099 포트번호로 실행시킨 서버, 중앙 상단은 라즈베리파이에서 8099포트번호로 실행시킨 서버이며, 왼쪽 하단은 맥북에서 실행시킨 9099서버, 중앙 하단은 맥북에서 실행시킨 8099서버이다.

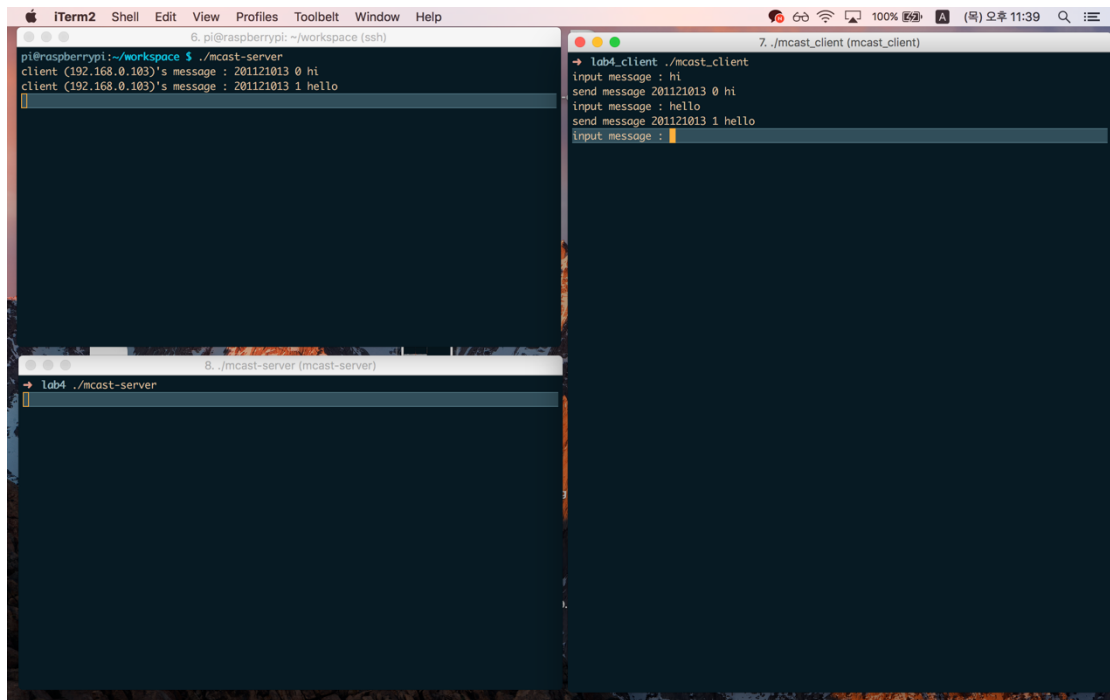


결과는 예상 결과와 같이 9099 포트번호에 해당하는 서버들만 메시지를 수신하였다.



### 3-1. Multicast Programming

먼저, 라즈베리파이와 맥북에서 각각 서버를 하나씩 실행시키고, 맥북에서 클라이언트를 실행시켰다. 그리고 클라이언트에서 'hi', 'hello'의 순으로 메시지를 보내었다. 처음에는, 맥북 서버만 IGMP관련 옵션을 주석처리하여 실행시켰다.



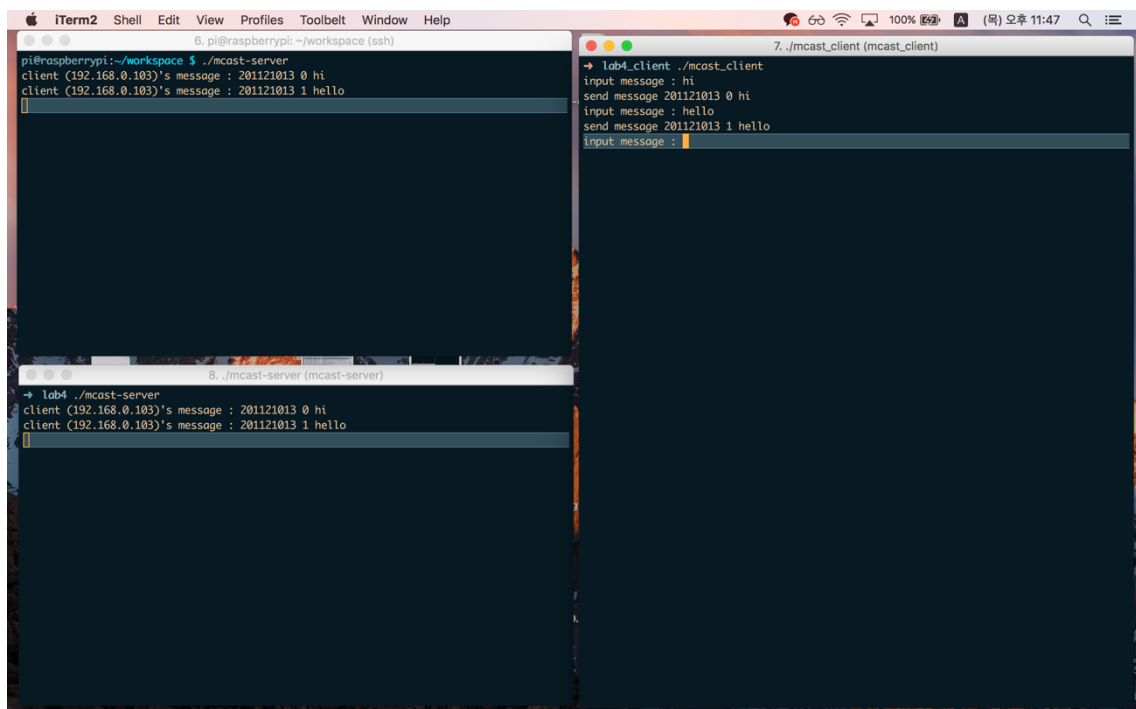
The screenshot shows three terminal windows. The top-left window (6) is a Raspberry Pi terminal running the mcast-server. It shows two messages received from client (192.168.0.103): 'hi' and 'hello'. The top-right window (7) is a Mac terminal running the mcast\_client. It shows the client sending 'hi' and 'hello' messages. The bottom window (5) is a Mac terminal running the mcast-server, which is currently empty.

```
6. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./mcast-server
client (192.168.0.103)'s message : 201121013 0 hi
client (192.168.0.103)'s message : 201121013 1 hello

7. ./mcast_client (mcast_client)
→ lab4_client ./mcast_client
input message : hi
send message 201121013 0 hi
input message : hello
send message 201121013 1 hello
input message :

5. ./mcast-server (mcast-server)
→ lab4 ./mcast-server
```

맥북에서 실행시킨 서버는 메시지를 받지 못하였고, 라즈베리파이에서 실행시킨 서버는 정상적으로 메시지를 받을 수 있었다. 그리고, 코드를 수정하여 모든 서버가 member join하여 테스트하였다. 결과는 다음과 같았다.



The screenshot shows three terminal windows. The top-left window (6) is a Raspberry Pi terminal running the mcast-server. It shows two messages received from client (192.168.0.103): 'hi' and 'hello'. The top-right window (7) is a Mac terminal running the mcast\_client. It shows the client sending 'hi' and 'hello' messages. The bottom window (8) is a Mac terminal running the mcast-server. It shows two messages received from client (192.168.0.103): 'hi' and 'hello'.

```
6. pi@raspberrypi: ~/workspace (ssh)
pi@raspberrypi:~/workspace $ ./mcast-server
client (192.168.0.103)'s message : 201121013 0 hi
client (192.168.0.103)'s message : 201121013 1 hello

7. ./mcast_client (mcast_client)
→ lab4_client ./mcast_client
input message : hi
send message 201121013 0 hi
input message : hello
send message 201121013 1 hello
input message :

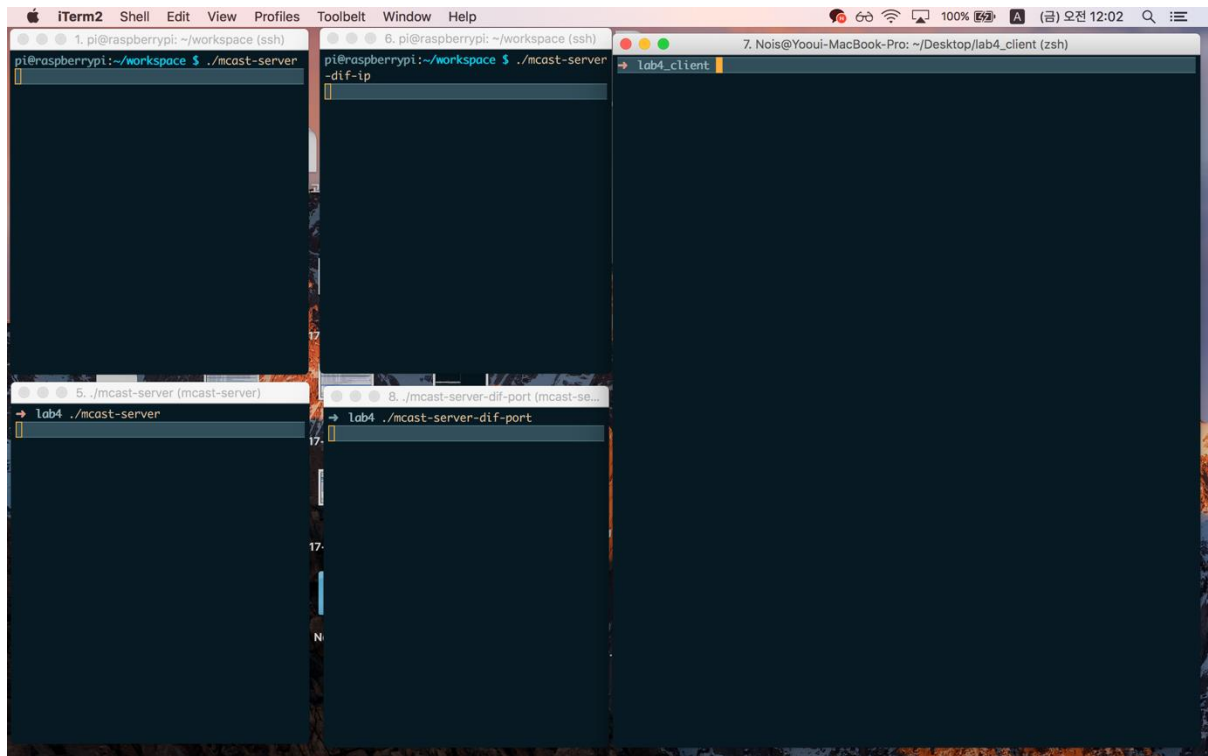
8. ./mcast-server (mcast-server)
→ lab4 ./mcast-server
client (192.168.0.103)'s message : 201121013 0 hi
client (192.168.0.103)'s message : 201121013 1 hello
```

## 3-2. Multicast Programming

2-2와 마찬가지로 해당 실습의 경우, 실습실에서 진행하여 결과를 확인하였으나 여러 머신을 확보할 수 없어 보고서 작성시에는 제대로 테스트할 수 없었다. 예상 결과는 모든 멤버가 join하면 모든 Server에서 메시지를 수신할 수 있을 것이라 예상하였고, 일부 멤버만 join한 경우 join한 멤버만 수신할 것이라고 예상하였다. 이유는 IGMP의 동작 원리 때문인데, 라우터가 join한 host들 에게만 해당 프레임을 전송하기 때문이다. 그리고 3-1의 결과가 이를 말해준다고 생각한다.

## 3-3. Multicast Programming

3-3역시 여러 머신을 확보할 수 없어 라즈베리파이와 맥북에서 각각 2개의 서버를 실행시켰다. 하나는 원래 서버이고 다른 하나는 환경을 변경하여 테스트하였다. 라즈베리파이에서는 mcast 주소를 변경한 서버를, 맥북에서는 port번호를 변경하였다. 아래 그림을 보자.



좌측 상단은 라즈베리파이 - 기존 멀티캐스트 서버, 중앙 상단은 라즈베리파이 - mcast주소 변경(224.10.0.4 -> 224.10.0.15), 좌측 하단은 맥북 - 기존 멀티캐스트서버, 중앙 하단은 맥북 - port 변경.(9010 -> 8010) 의 변화를 주어 진행하였다. 예상 결과는 기존 멀티캐스트 서버만 받을 것이라고 예상하였다. 왜냐하면 1) 라즈베리파이에서 mcast를 변경시킨 경우 ip주소 뒤의 23비트를 맥주소 뒤의 23비트로 매핑시키기 때문인데, 224.0.10.4 -> 224.0.10.15는 IP주소 뒤의 23비트에 변경이 있기 때문이다. 2) 맥북에서 port를 변경시킨 경우 해당 소켓이 다른 port번호로 bind되기 때문이다. 결과는 아래와 같았다.

The image shows four terminal windows from an iTerm2 session on a Raspberry Pi. The windows are arranged in a 2x2 grid. The top-left window (tab 1) shows the command `pi@raspberrypi: ~/workspace (ssh) $ ./mcast-server` and is currently empty. The top-right window (tab 6) shows the command `pi@raspberrypi: ~/workspace (ssh) $ ./mcast-server -d f-ip` and displays two messages from client (192.168.0.103): `client (192.168.0.103)'s message : 20112101 3 0 a2` and `client (192.168.0.103)'s message : 20112101 3 1 hi`. The bottom-left window (tab 5) shows the command `lab4 ./mcast-server` and displays two messages from client (192.168.0.103): `client (192.168.0.103)'s message : 201121013 0 a` and `client (192.168.0.103)'s message : 201121013 1 hi`. The bottom-right window (tab 7) shows the command `lab4_client ./mcast_client` and displays the input message `a`, the send message `201121013 0 a`, the input message `hi`, and the send message `201121013 1 hi`.

예상과 달리, mcast 주소를 변경한 경우, 먼저 실행한 쪽만 메시지를 전송받았다. 위의 그림에서는 mcast 주소를 변경한 서버를 먼저 실행시켰는데, 반대로 기존의 서버를 먼저 실행하는 경우 기존 서버만 메시지를 받았다. 이는 필자가 앞에서 말했던 매핑한 맥주소의 개념이 일치하지 않았기 때문이다. 앞에서 설명했던 매핑된 주소는 라우터가 가지고 있는 멤버의 맥주소가 된다. 또한 먼저 실행시킨 프로세스가 9010포트를 사용하고 있었으므로, 나중에 실행시킨 서버가 9010포트를 사용할 수 없기 때문에, 메시지를 받지 못했다. 이번 3-3 실습에서는, 필자가 예상했던 내용이 다소 다르게 나와서 의외였다. 때문에 해당 내용을 공부하여 정확히 이해할 수 있는 계기가 되었다.