Guide for Programming Assignment 2

This guide is intended to help you with Programming Assignment 2, especially in understanding how trees are formed. You can use the test cases described here in testing your work.

We have provided you a script (start_servers.sh) which starts several servers and builds a topology. Note that the tree for each channel is an overlay built over this topology. The topology does not change once the servers are started.

Example 1:

First let's look at an example with the simple two-server topology. You can start servers to build this topology by uncommenting the relevant section of the file. The first half of the file should look like this after the change.

```
#!/usr/bin/bash

# Change the SERVER variable below to point your server executable.
SERVER=./server

SERVER_NAME=`echo $SERVER | sed 's#.*/\(.*\)#\1#g'`

# Generate a simple two-server topology
$SERVER localhost 5000 localhost 5001 &
$SERVER localhost 5001 localhost 5000 &

# Generate a capital-H shaped topology
#$SERVER localhost 4000 localhost 4001 &
```

Note that I have used port numbers 5000 and 5001 because 4000 and 4001 were not available at the time of testing.

After running this script, the topology of servers is as follows.

5000 ---------- 5001

Until a client joins a channel, there are no trees per channels because there are no channels.

Now let's start a client and join the server at port 5000 from the client.

```
./client localhost 5000 dayacw
```

This creates a sequence of messages between the servers. The actual output is given below.

```
127.0.0.1:5000 127.0.0.1:54427 recv Request login dayacw
127.0.0.1:5000 127.0.0.1:54427 recv Request join dayacw Common
127.0.0.1:5000 127.0.0.1:5001 send S2S Join Common
127.0.0.1:5001 127.0.0.1:5000 recv S2S Join Common
```

According to the standard client functionality, it joins channel “Common” immediately after login.

When the server at port 5000 receives this join message, it realizes that it has not subscribed to this channel (meaning that it has no records about this channel in its internal data structures) and sends an S2S join message for this channel to server at 5001. It should send S2S join messages in this manner for all other servers that it knows but in this case server at 5001 is the only other server that it knows.

Upon receiving the S2S join message, the server at 5001 internally records that it has subscribed to this channel. It has nowhere else to forward the message so no more S2S messages are sent.

At this point, the tree for channel Common is as follows.

5000 ---------- 5001

Now the client sends a say message to its server (server at port 5000).

```
>hello
```

This results in the following messages.

```
127.0.0.1:5000 127.0.0.1:54427 recv Request say dayacw Common "hello"
127.0.0.1:5000 127.0.0.1:5001 send S2S say dayacw Common "hello"
127.0.0.1:5001 127.0.0.1:5000 recv S2S say dayacw Common "hello"
127.0.0.1:5001 127.0.0.1:5000 send S2S Leave Common
127.0.0.1:5000 127.0.0.1:5001 recv S2S Leave Common
```

As you can see, the client runs at port 54427. After receiving the message server@5000 sends an S2S say message to server@5001.

After receiving this message, server@5001 realizes that it has no where to forward it (it does not have any clients subscribed to channel Common and it does not know any other server subscribed to the channel). As such it has to leave the channel. So it sends a S2S leave message to server@5000. Then it removes internal records of this channel.

Now the tree for channel Common consists of server@5000 only.

Let’s assume that nothing happens after this for 1 minute. After 1 minute, the servers have to send S2S join messages regarding each channel it has subscribed to *all* of its neighbors. This is to guard against network failures. In this case, after 1 minute server@5000 sends a S2S join message for channel “Common” to its neighbor,

server@5001. This doesn't seem to make any sense in this simple example but this mechanism is useful in more complex topologies.

This step results in the following messages.

```
127.0.0.1:5000 127.0.0.1:5001 send S2S Join Common
127.0.0.1:5001 127.0.0.1:5000 recv S2S Join Common
```

Now the tree consists of the two servers again as follows.

5000 ---------- 5001

Example 2:

Now let's look at an example on using the capital H shaped topology. Here I have uncommented the following section.

```
# Generate a capital-H shaped topology
$SERVER localhost 5000 localhost 5001 &
$SERVER localhost 5001 localhost 5000 localhost 5002 localhost 5003 &
$SERVER localhost 5002 localhost 5001 &
$SERVER localhost 5003 localhost 5001 localhost 5005 &
$SERVER localhost 5004 localhost 5005 &
$SERVER localhost 5005 localhost 5004 localhost 5003 localhost 5006 &
$SERVER localhost 5006 localhost 5005 &
```

This builds the following topology.

```
5000            5004
  |               |
5001 — 5003  — 5005
  |               |
5002            5006
```

Now let's connect to the server@5000 from a client.

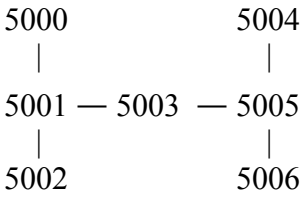
```
./client localhost 5000 user1
```

This creates the following sequence of messages.

```
127.0.0.1:5000 127.0.0.1:54878 recv Request login user1
127.0.0.1:5000 127.0.0.1:54878 recv Request join user1 Common
127.0.0.1:5000 127.0.0.1:5001 send S2S Join Common
127.0.0.1:5001 127.0.0.1:5000 recv S2S Join Common
127.0.0.1:5001 127.0.0.1:5002 send S2S Join Common
127.0.0.1:5001 127.0.0.1:5003 send S2S Join Common
127.0.0.1:5002 127.0.0.1:5001 recv S2S Join Common
127.0.0.1:5003 127.0.0.1:5001 recv S2S Join Common
127.0.0.1:5003 127.0.0.1:5005 send S2S Join Common
127.0.0.1:5005 127.0.0.1:5003 recv S2S Join Common
127.0.0.1:5005 127.0.0.1:5004 send S2S Join Common   (contd.)
```

```
127.0.0.1:5005 127.0.0.1:5006 send S2S Join Common
127.0.0.1:5004 127.0.0.1:5005 recv S2S Join Common
127.0.0.1:5006 127.0.0.1:5005 recv S2S Join Common
```

As in the previous example, the client sends a join message for channel “Common” immediately after login. As a result server@5000 sends a S2S join message to its neighbor server@5001. Then server@5001 sends S2S join messages to its neighbors at ports 5002 and 5003. This process continues until the following tree is created for channel Common. Note that this covers the entire topology.



Now let’s assume that another client joins the server@5005.

```
./client localhost 5005 user1
```

When the server@5005 receives the join message for channel Common from the new client, it sees that it has already subscribed to this channel. Therefore no additional messages are necessary.

As such, this step results in the following messages only.

```
127.0.0.1:5005 127.0.0.1:54879 recv Request login user2
127.0.0.1:5005 127.0.0.1:54879 recv Request join user2 Common
```

Now the client “user1” (connected to server@5000) sends a say message.

```
>hello
```

This results in the following messages. In the end, user2 connected to server@5005 gets the message typed by user1.

```
127.0.0.1:5000 127.0.0.1:54878 recv Request say user1 Common "hello"
127.0.0.1:5000 127.0.0.1:5001 send S2S say user1 Common "hello"
127.0.0.1:5001 127.0.0.1:5000 recv S2S say user1 Common "hello"
127.0.0.1:5001 127.0.0.1:5002 send S2S say user1 Common "hello"
127.0.0.1:5001 127.0.0.1:5003 send S2S say user1 Common "hello"
127.0.0.1:5003 127.0.0.1:5001 recv S2S say user1 Common "hello"
127.0.0.1:5003 127.0.0.1:5005 send S2S say user1 Common "hello"
127.0.0.1:5002 127.0.0.1:5001 recv S2S say user1 Common "hello"
127.0.0.1:5002 127.0.0.1:5001 send S2S Leave Common
127.0.0.1:5005 127.0.0.1:5003 recv S2S say user1 Common "hello"
127.0.0.1:5001 127.0.0.1:5002 recv S2S Leave Common
127.0.0.1:5005 127.0.0.1:5004 send S2S say user1 Common "hello"
127.0.0.1:5005 127.0.0.1:5006 send S2S say user1 Common "hello"
127.0.0.1:5006 127.0.0.1:5005 recv S2S say user1 Common "hello"
127.0.0.1:5006 127.0.0.1:5005 send S2S Leave Common
```

```
127.0.0.1:5005 127.0.0.1:5006 recv S2S Leave Common
127.0.0.1:5004 127.0.0.1:5005 recv S2S say user1 Common "hello"
127.0.0.1:5004 127.0.0.1:5005 send S2S Leave Common
127.0.0.1:5005 127.0.0.1:5004 recv S2S Leave Common
```

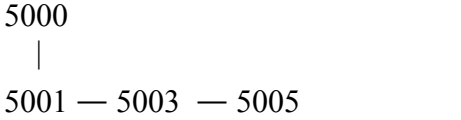
You can see how messages are sent and received by reading this output. The S2S Say messages originate from the server@5000 and forwarded through the tree.

As you can see, these S2S Say messages result in three S2S Leave messages. They are as follows.

- server@5002 sends an S2S Leave message to server@5001
- server@5004 sends an S2S Leave message to server@5005
- server@5006 sends an S2S Leave message to server@5005

Note that in each case, the server sending the S2S Leave message has no other client or server to forward the message. Also note that the servers receiving these S2S Leave messages also print lines in the output upon receiving the messages.

After these S2S Leave messages, the tree for channel Common is as follows.



This output does not have a line on server@5005 forwarding the message to its client (user2), but it's better to show this line as well.

You can come up with few more examples like this to test your code. After you get the output from the servers, read them carefully and try to identify any errors.