Program 2

Assigned: Nov. 3th
Due: Nov. 30 (11:59 pm)
Notes: Undergraduates may work in pairs.

You should check the Clarifications and F.A.Q. on Program 2 page for clarifications to the assignment.

Imaginarians Business Plan

In your last project, you implemented a chat server and a client. Since then, the people of Imaginaria have deployed your software nationwide. Your software has become so popular that they require multiple servers to handle the load. Their plan is for each user to connect to a server near to them to improve responsiveness. However, the Imaginarians need the ability to talk to users on other servers.

In this project, you will improve your server software to support server-to-server communication. The Imaginarians will provide a list describing their servers and how they want them connected. Your job will be to implement the Duckchat Server-to-Server protocol so that users on different servers can communicate.

The simplest design would be to flood each message sent by every user. However, this is inefficient and would not meet the performance requirements of the Imaginarians. Instead, only Join messages will be flooded. These will be used to form a tree of servers listening to that channel. That way, text from users can be distributed efficiently.

The Duckchat Server-to-Server protocol has a particularly nice aspect: no changes are required to the client program. This delights the Imaginarians, since they won't need to upgrade their users' software.

Duckchat: Server-to-Server

In Project #1, your server took two parameters as arguments. In this assignment, your server must take a variable number of arguments. The first two arguments will always be present and are the IP address and port number of the server. Any additional

arguments are the IP addresses and port numbers of additional servers. We will provide a script that starts up several servers and connects them together. It's available here.

Each server must maintain additional state. Each server already keeps track of which users are in each channel. Now, the server must additionally note if any adjacent servers are subscribed to a channel. The servers subscribed to a channel will form a tree. When a user types a message, the user's server transmits the message throughout the tree.

When a server receives a message from one of its users, the server broadcasts the message to any adjacent servers on the channel. If a server receives a message from another server, it forwards it to any other servers on that channel. Obviously, the server also sends the message to any interested users.

Keep in mind that this tree is an per-channel overlay built over the existing topology. The topology itself never changes; servers never form new connections once started. The tree is a subset of this topology, and a separate tree exists for each channel.

For simplicity, you will not be required to implement inter-server versions of List and Who.

Forming Trees

When a user joins a channel, the user's server checks to see if it is already subscribed to that channel. If so, the server need not take any additional steps. If not, the server must attempt to locate other servers subscribed to that channel.

The server begins by sending a Join message to all adjacent servers. When a server receives one of these messages, it checks to see if it is already subscribed. If so, the join message ends there. If not, the server subscribes itself to the channel, and forwards the message out all remaining interfaces. Intermediate servers must subscribe themselves in this way to ensure there is a path to distant servers on the same channel.

Removing unnecessary servers from a channel will be done in a lazy fashion. When a server receives a Say message, but has no where to forward it, it responds with a Leave message. In other words, if a server is a leaf in tree with no users, it removes itself from the tree. Note that this means that a server has to make sure that it does not have any clients and that it also only has at most one other server subscribed to the

channel before it removes itself from the tree. In other words server should not have any client in the particular channel and it should know about only one other server in the channel (the one that is sending messages to it) when it decides to remove itself from the tree.

To guard against loops, additional steps must be taken. Inter-server Say messages must include a unique identifier, and each server must maintain a list of recent identifiers. When the server receives a new message, it checks against this list. If a duplicate is detected, the server knows a loop has been found. It discards the Say message, and sends a Leave message to the sender. The Leave message is not forwarded further. This removes the extra link from the loop.

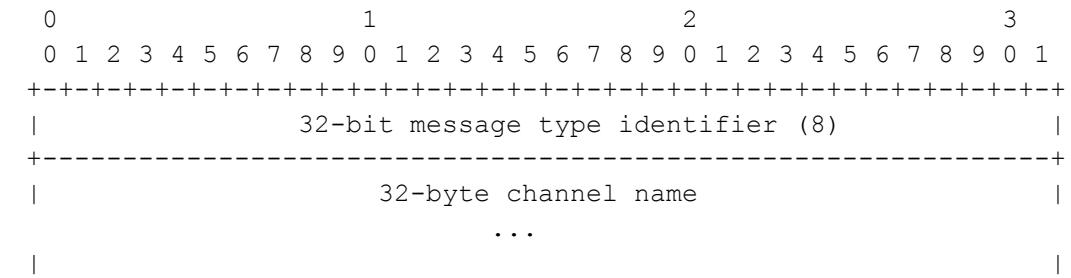
Finally, to guard against network failures, soft-state Joins will be used. Every server must renew its subscriptions by sending a new Join message once per minute. A server interprets a two minute interval with no Join as a Leave. This ensures that a crashed server will not split the tree.

There is no simple way for a server to generate globally unique identifiers. Therefore, the Imaginarians have agreed to settle for identifiers that are unique with high probability. To accomplish this, servers use their random number generator to create the unique identifiers. Servers must seed their random number generator by reading bytes from /dev/urandom.

Protocol

The protocol between clients and servers is unchanged. However, the new DuckChat Server-to-Server protocol must be supported. Like the DuckChat client-server protocol, each message begins with a 32-bit type identifier. There are just three messages in the DuckChat Server-to-Server protocol:

- S2S Join (8): The message contains a 32 byte channel name. Here's the layout of the packet:



- +-+
- S2S Leave (9): The message contains a 32 byte channel name. Here's the layout of the packet:
- 0 1 2 3
- 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
- +-+
- | 32-bit message type identifier (9) |
- +---+
- | 32-byte channel name |
- ...
- | |
- +-+
- S2S Say (10): The message contains a 64 bit unique identifier, a 32 byte username, a 32 byte channel name, and a 64 byte text field. Here's the layout of the packet:
- 0 1 2 3
- 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
- +-+
- | 32-bit message type identifier (10) |
- +---+
- | 64-bit unique identifier |
- | |
- +---+
- | 32-byte user name |
- ...
- | |
- +---+
- | 32-byte channel name |
- ...
- | |
- +---+
- | 64-byte text field |
- ...
- | |
- +-+
-

Output

To simplify debugging and grading, servers must output a message each time they transmit or receive a message. Because all the servers will typically be writing to the same terminal, it is necessary to label each message with the server's IP address and port number. These messages must follow these examples:

```
127.0.0.1:12792 127.0.0.1:4000 send S2S Join Common
127.0.0.1:4000 127.0.0.1:12792 recv S2S Join Common
```

```
127.0.0.1:12792 127.0.0.1:8295 recv Request Say Common "hello"
127.0.0.1:12792 127.0.0.1:4000 send S2S Say Agthorr Common "hello"
127.0.0.1:4000 127.0.0.1:12792 recv S2S Say Agthorr Common "hello"
```

Note that the server that receives or sends the message is always on the left side. Who the server receives from or sends to is on the right side. Also note that the 3rd line is a client (at 127.0.0.1:8295) sending a say message to the server at 127.0.0.1:12792.

Grading

- 100 points total, Server Program
    - 10 points. Server broadcasts Joins correctly when a user requests a Join.
    - 10 points. Server broadcasts Joins correctly when receiving a Join from another server.
    - 10 points. Servers generate unique identifiers correctly.
    - 14 points. Soft-state works correctly.
        - 7 points. Servers renew Joins once per minute.
        - 7 points. Servers remove Joins older than two minutes.
    - 12 points. Servers reply with a Leave message when they cannot forward a Say message.
    - 12 points. Servers reply with a Leave message when they receive a duplicate Say message.
    - 12 points. Servers do not send messages inappropriate Join, Leave, or Say messages.
    - 10 points. All functionality from Program #1 still works for a server in isolation.
    - 10 points. All client functionality and the client is robust.

"Robust" means that the program does not crash, even if we send it bogus packets.

Extra Credit

Extra credit will be given for the following inventing and coding Server-to-Server versions of Who or List. This will be worth an extra 10 points on the assignment. Note that you do not get more points for doing both. Both of these messages will need to be broadcast and include a unique identifier. You will also need to define a special Server-to-Server reply message. When a Server-to-Server List request is received, it should be forwarded to all servers except the one that issued the request. When a reply from all these servers is received, the replies should be bundled together into a single reply and sent to the requester. This bundling process must remove duplicates. Who works in a similar way, but it only needs to be broadcast to other servers on that channel.

To test your server, we will use the sample client from Program #1.

Tips

Don't leave this assignment until the last minute! You won't be able to finish it, and there'll be less time to ask questions if you have any.

Develop your program gradually. Get a minimum set of functionality coded, tested, and debugged before moving on to additional functionality. We recommend working on building the tree first, then work on pruning it. Here is one order to write the code in:

- Add the required debugging text for all messages received from clients.
- Add support to handle the additional command line arguments and setup the topology.
- Add support for broadcasting Joins when a user joins a channel.
- Add support for forwarding Joins from another server.
- Add support for Server-to-Server Say messages, including loop detection.
- Add support for sending Leave when a Say cannot be forwarded.
- Add support for the soft state features.
- Try several topologies. Verify that trees are formed and pruned correctly.
- Copy your server code and modify it to send invalid packets to see if you can make your server crash. Fix any bugs you find.

Note: while you may work in groups of at most 2 (for Undergrads) for coding this assignment, you are encouraged to test interoperabability with as many other students as possible. This may help you find unusual bugs in your code.

You might find the following guide helpful as well. It gives some examples on how trees are formed and you can use these examples to test your code.

Sample code

We are providing sample code for server and client from project 1 to allow all students to work on project 2 from a correctly working code. You can download the source code for client and server from project 1 in C from here.

While we provide these correct source codes, we remind you that working with these codes might be more difficult than using your own code from project 1. So we still encourage you to work with your own code from Project 1 if it is sufficiently stable or use provided codes to debug that parts of your code which are not working properly.

Submission

Step 1:

Fill the correct submission form and compress all the source files, makefile and the submission form, into a file (.zip, .tar , ...)

Link to the submission form is given below:

- Submission form

Again, put all the source files, makefile and the submission form into a folder by your last name (or the last name of one of the group members if you are working in a group of two). Make a compressed file from the folder. Please include any special instructions for compiling or running your program on ix.

Step 2:
Please use the TurnIN Homework submission page (Click Here) to upload the compressed file of your project .
First you need to login if you are not logged in, using your CIS login name and password - @cs.uoregon.edu -.
Then, Choose the "CIS 432" (532 students also should select this option) and press "GO",
Choose "prog2" and press "GO"
choose your compressed file and press the "GO" for the last time.

That's it!

Make sure that you submit your program before the deadline. (11:59pm on Nov. 30th.)