15-441 Computer Networks
Project 2: IRC Routing

Lead TA: Xin Zhang <xzhang1@cs.cmu.edu>

Assigned: February 14, 2008
Checkpoint 1 Due: February 26, 2008
Checkpoint 2 Due: March 18, 2008
Final Due: March 25, 2008 (11:59 PM EST)

1 Introduction

The purpose of this project is to give you experience in developing concurrent network applications. You will use the BERKELEY SOCKETS API to write an Internet chat server using a subset of the Internet Relay Chat protocol (IRC)[1] and implement two different routing protocols so chat messages can be exchanged between a network of chat servers.

You will implement a shortest path link state routing protocol. In this protocol, each node in the network periodically exchanges information with its neighbors so that everyone in the network knows the best path to take to reach each destination. This is similar to the protocols used in Internet routers. At the end of this project, you will have your own network of chat servers, which could be used to talk with users across the world.

2 Logistics

- The tar file for this project can be found at:

`http://www.cs.cmu.edu/~dga/15-441/S08/project2/project2.tar.gz`

- This is a group project. You **must** find *exactly one* partner for this assignment. Bboard is an excellent channel for looking for partners. In case there are an odd number of people in the class and you are left out, please contact xzhang1@cs.cmu.edu.
- Once you have found a partner, email Xin (xzhang1@cs.cmu.edu) your names and andrew logins so we can assign a group number to you. Use "**15441 GROUP**" as the subject line. Please try to be sure you know who you will work with for the full duration of the project so we can avoid the hassle of people switching later.
- This is a large project, but not impossible. Here is a recommended set of suggested milestones:

√	Date	Milestone
	2/14	Project Assigned
	2/19	Understand the project handout and learn pertinent knowledge and skills
	2/26	**CHECKPOINT 1 Due – IRC server extensions/interfaces tested thoroughly**
	3/18	**CHECKPOINT 2 Due – Routing daemon tested thoroughly**
	3/25	Last minute rush to get things done and hand-in

[1] http://www.irchelp.org/irchelp/rfc/

3 General Overview

The routing daemon will be a separate program from your IRC server. Its purpose is to maintain the routing state of the network (e.g., build the routing tables or discover the routes to destinations). When the IRC server wants to send a message to a remote user, it will ask the routing daemon how to get there and then send the message itself. In other words, the routing daemon does the routing and the IRC server does the forwarding.[2]

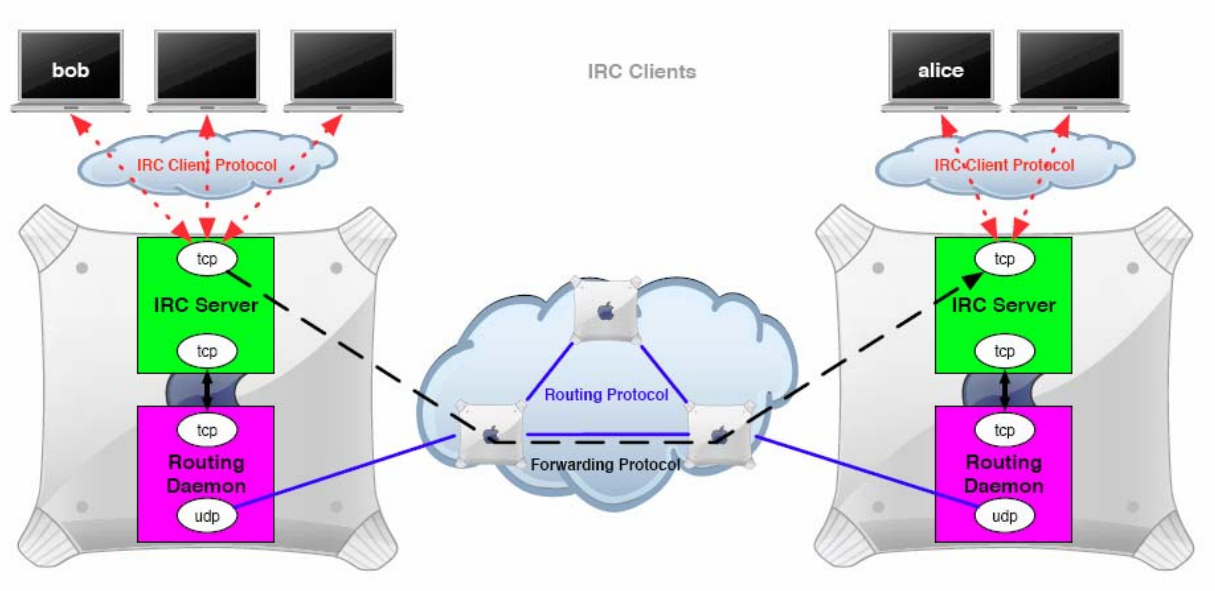

Figure 1 – Sample IRC Network

In your implementation, the routing daemon will communicate with other routing daemons (on other nodes) over a UDP socket to exchange routing state. It will talk to the IRC server that is on the same node as it via a local TCP socket. The IRC server will talk to other IRC servers via the TCP socket that it also uses to communicate with clients. It will simply use special server commands. This high level design is shown in the two large IRC server nodes in Figure 1.

In order to find out about the network topology, each routing daemon will receive a list of neighboring nodes when it starts. In this project, you can assume that no new nodes or links will ever be added to the topology after starting, but nodes and links can fail (i.e., crash or go down) during operation (and may recover after failing).

4 Definitions

Before jumping into the gory details, let us define some terminology.

- **node –** An IRC server and routing daemon pair running together that is part of the larger network. In the real world, a node would refer to a single computer, but we can run multiple "virtual" nodes on the same computer since they can each run on different ports. Each node is identified by

[2] In actual routers, and even overlay networks like peer-to-peer file sharing networks, the notion of the separate routing daemon is atypical. Normally, the forwarding program should keep the forwarding table, not query the route daemon for each route lookup.

its nodeID.

- **nodeID** – A unique identifier that identifies a node. This is an unsigned 32-bit integer that is assigned to each node when its IRC server and routing daemon start up.
- **neighbor** – Node 1 is a neighbor of node 2 if there is a virtual link between 1 and 2. Each node obtains a list of its neighbors' nodeIDs and their routing and forwarding ports at startup.
- **destination** – An IRC username or nick as a null terminated character string. As per the IRC RFC, destinations will be at most 9 characters long and may not contain spaces.
- **IRC port** – The TCP port on the IRC server that talks to clients and other IRC servers.
- **forwarding port** – Same as IRC port.
- **routing port** – The UDP port on the routing daemon used to exchange routing information with other routing daemons.
- **local port** – The TCP port on the routing daemon that is used to exchange information between it and the local IRC server. For example, when the IRC server wants to find out the route to remote user, it queries the routing daemon on this port. The socket open for listening will be on the routing daemon. The IRC server will connect to it.
- **OSPF –** The shortest path link state algorithm that inspires the (much simpler) algorithm you will implement

  routing table – The data structure used to store the "next hops" that packet should take used in OSPF.

5 Link-State Routing Protocol

5.1 Basic Operation

You will implement a link-state routing protocol similar to OSPF, which is described in the textbook in chapter 4, and in more detail in the OSPF RFC[3]. Note, however, that your protocol is greatly simplified compared to the actual OSPF spec. As described in the references, OSPF works by having each router maintain an identical database describing the network's topology. From this database, a routing table is calculated by constructing a shortest-path tree. Each routing update contains the node's list of neighbors, users, and channel. Upon receiving a routing update, a node updates its routing table with the "best" routes to each destination. In addition, each routing daemon must remove entries from its routing table when they have not been updated for a long time. The routing daemon will have a loop that looks similar the following:

```
while (1)
{
  /* each iteration of this loop is "cycle" */
  wait_for_event(event);

  if (event == INCOMING_ADVERTISEMENT)
  {
    process_incoming_advertisements_from_neighbor();
  }
  else if (event == IT_IS_TIME_TO_ADVERTISE_ROUTES)
```

[3] http://www.rfc-editor.org/rfc/rfc2328.txt

```
  {
    advertise_all_routes_to_all_neighbors();
    check_for_down_neighbors();
    expire_old_routes();
    delete_very_old_routes();
  }
}
```

Let's walk through each step. First, our routing daemon A waits for an event. If the event is an incoming link-state advertisement (LSA), it receives the advertisement and updates its routing table if the LSA is new or has a higher sequence number than the previous entries. If the routing advertisement is from a new router B or has a higher sequence number than the previously observed advertisement from router B, our router A will flood the new announcement to all of its neighbors except the one from which the announcement was received, and will then update its own routing tables.

If the event indicates a predefined period of time has elapsed and it is time to advertise the routes, then the router advertises all of its users, channels, and links to its direct neighbors. If the routing daemon has not received any such advertisements from a particular neighbor for a number of advertisements, the routing daemon should consider that neighbor down. The daemon should mark the neighbor down and re-flood LSA announcements from that neighbor with a TTL of zero When your router receives an announcement with a TTL of zero, it should delete the corresponding LSAs from its table.

If the event indicates that a user has joined or left a channel or the server, the router should send a *triggered update* to its neighbors. This is simply a new link state advertisement with a higher sequence number that announces the router's new state. If a node has not sent any announcements for a very long time, we expire it by removing it from our table.

If B receives an LSA announcement from A with a lower sequence number than it has previously seen (which can happen, for example, if A reboots), B should echo the prior LSA back to A. When A receives its own announcement back with a higher sequence number, it will increment its transmitted serial number to exceed that of the older LSAs.

Each routing announcement should contain a full state announcement from the router – all of its neighbors, all of its users, and all of its channels. This is an inefficient way to manage the announcements, but it greatly simplifies the design and implementation of the routing protocol to make it more tractable for a 5 week assignment. Each time your router originates a new LSA, it should increment the serial number it uses. When a router receives an updated LSA, it recomputes its local routing table. The LSAs received from each of the peer nodes tell the router a link in the complete router graph. When a router has received all of the LSAs for the network, it knows the complete graph. Generating the user routing table is simply a matter of running a shortest-paths algorithm over this graph.

5.2 Reliable Flooding

OSPF is based upon reliable flooding of link-state advertisements to ensure that every node has an identical copy of the routing state database. After the flooding process, every node should know the exact network topology. When a new LSA arrives at a router, it checks to see if the sequence number on the LSA is higher than it has seen before. If so, the router reliably transmits the message to each of its peers except the one from which the message arrived. The flooding is made reliable by the use of acknowledgement packets from the neighbors. When router A floods an LSA to router B, router B responds with an "LSA Ack." If router A does not receive such an ack from its neighbor within a certain amount of time, router A will retransmit the LSA to B.

With the information contained in the LSAs, each server should be able to deliver messages from one user to another without much trouble. To send messages to a channel, however, requires a little more work; this is multicast routing instead of unicast routing. A channel can exist on multiple servers, so the distribution can take multiple branches at a time. How does the local node know which neighbors to forward the message to in this case?

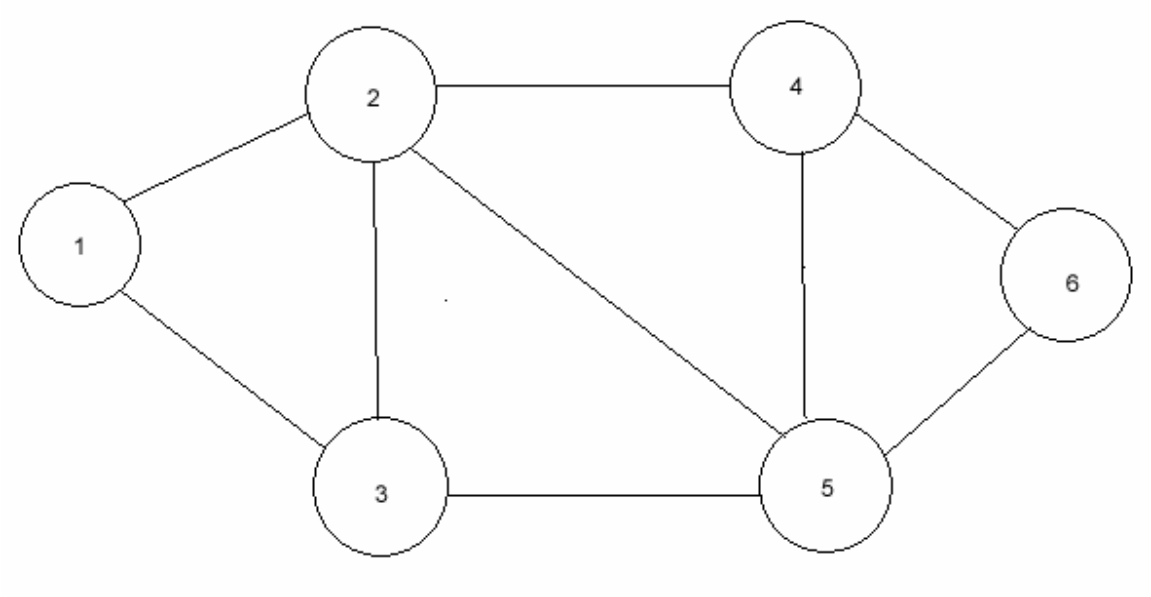

Figure 2 – Sample Network

Since a channel can exist on multiple servers and the server knows the network topology, each server is able to construct a source rooted shortest paths tree for that message, which tells the server what outgoing links it should use. Note that this tree is rooted at the *message* source, not the router making the computation. The algorithm for computing the shortest paths tree for the multicast case is the same as for the unicast to a user case, except that the source may not be the local node. With these trees, a server will know which servers it should propagate a channel message to, depending upon which server sent the message. Note that there is a different shortest paths tree for every channel/source pair.

Why does it need to know the source? Consider the network pictured in Figure 2. Now, suppose nodes 1, 2, 5 and 6 have users subscribed to channel `#perl`, and nodes 1, 3, 4, and 6 have users subscribed to channel `#c`. If a user on node 1 wants to send a message to `#perl` then it should propagate the message to node 2. Node 2 knows nodes 1 and 5 also have users in `#perl`, but since the message came from node 1, it should not propagate the message back to node 1. So, it only propagates the message to node 5. Examples are given below to help you understand the protocol. Make sure you can understand the examples.

The channel routing table for node 1 would look like:

Channel name	Source Node	Next-Hops
`#perl`	1	2
`#perl`	2	None
`#perl`	5	None
`#perl`	6	None
`#c`	1	2,3
`#c`	3	None
`#c`	4	None
`#c`	6	None

The channel routing table for node 2 would look like:

Channel name	Source Node	Next-Hops
#perl	1	4,5
#perl	2	1,4,5
#perl	5	1
#perl	6	1
#c	1	4
#c	3	4
#c	4	1,3
#c	6	1

The channel routing table for node 5 would look like:

Channel name	Source Node	Next-Hops
#perl	1	None
#perl	2	None
#perl	5	2,6
#perl	6	None
#c	1	None
#c	3	6
#c	4	None
#c	6	3

The channel routing table for node 6 would look like:

Channel name	Source Node	Next-Hops
#perl	1	None
#perl	2	None
#perl	5	None
#perl	6	4,5
#c	1	None
#c	3	None
#c	4	None
#c	6	4,5

Now consider a case where a client on node 2 has not subscribed to the channel #c and sends a PRIVMSG to the channel. (Note that such a client can only send messages but cannot receive messages being sent by other users on the channel). In this case, the node will compute the shortest path tree. The message will be forwarded to nodes 1,3 and 4. Node 4 will then forward the message to node 6. This should clear the fact that being a source node and being a subscriber are independent.

There are two ways that a router could potentially compute these routes. It could compute all possible (source, channel) trees in advance, and populate its routing table with the results. Or, the router could compute the routes dynamically on demand when a message arrives for a particular channel from a particular source, and cache the results. **In this assignment, we will implement the dynamic computation and caching version**. The router's multicast routing table (channel routing table) should act as a cache, and the router should compute the trees dynamically if it has no entry. Remember to have a

way to indicate "route calculated, but there were no neighbors" in your routing table so that you don't eternally recompute local channels. Also, these cached trees must however be discarded when there is a change in the topology or subscription to channels as notified by triggered updates or normal advertisements.

Your router should be robust to misbehaving neighbors. If the router receives a channel message forwarded to it by a peer who should not have forwarded the message (remember, the router can check this, since it knows the shortest paths tree), it should silently drop the message. Such an event could occur during a route change when the routing table became temporarily inconsistent, and it can lead to routing loops. Because multicast can generate a large number of packets, and the IRC network is a less trusted environment than an ISP's own routers, dropping the message is safer (for the project), but might cause packet delivery to be somewhat less reliable. The congestion caused by routing loops is also typically addressed through a TTL in the packets being forwarded.

5.3 Protocol Specifications

Figure 3 shows the routing update message format, with the size of each field in bytes in parenthesis.

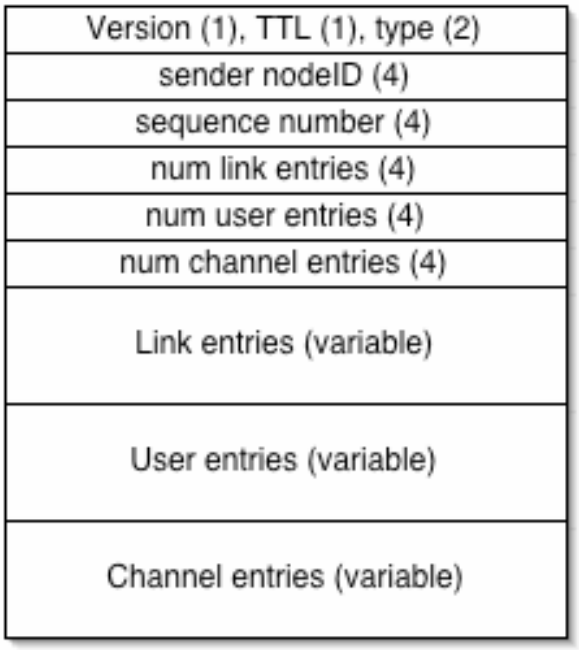

Figure 3 – OSPF Packet Format

- **Version** – the protocol version, always set to 1
- **TTL** – the time to live of the LSA. It is decremented each hop during flooding, and is initially set to 32.
- **Type** – Advertisement packets should be type 0 and Acknowledgement packets should be type 1.
- **Sender nodeID** – The nodeID of the sender of the message, not the immediate sender.
- **Sequence number** – The sequence number given to each message
- **Num link entries** – The number of link table entries in the message.
- **Num user entries** – The numbers of users announced in the message
- **Num channel entries** – The number of channels announced in the message
- **Link entries –** Each link entry contains the nodeID of a node that is directly connected to the sender. This field is 4 bytes.
- **User entries** – Each user entry contains the name of the destination user as a null terminated

string. Since the IRC RFC indicates that nicknames should be at most 9 characters and we have added the constraint that channels can be at most 9 characters (including & or #), it should definitely fit within 16 (the unused bytes will be ignored).

- **Channel entries** – Same as a user entry, above.

All multi-byte integer fields (nodeIDs, TTLs, link entries, etc) should be in **network byte order**.

An acknowledgement packet looks very similar to an announcement packet, but it does not contain any entries. It contains the sender nodeID and sequence number of the original announcement, so that the peer knows that the LSA has been reliably received.

5.4 Requirements

Your implementation of OSPF should have the following features:

- Given a particular network configuration, the routing tables at all nodes should converge so that forwarding will take place on the paths with shortest length.
- In the event of a tie for shortest path, the next hop in the routing table should always point to the nodeID with the lowest numerical value. Note that this implies there should be a unique solution to the routing tables in any given network.
- Remove the LSAs for a neighbor if it hasn't given any updates for some period of time.
- You should implement *Triggered Updates* (when a link goes down or when users join or leave a server or channel).
- If a node or link goes down (e.g., routing daemon crashes, or link between them no longer works and drops all messages), your routing tables in the network should re-converge to reflect the new network graph. You shouldn't have to do anything more to make sure this happens, as the above protocol already ensures it.

You do not have to implement the following:

- You do not have to provide authentication or security for your routing protocol messages.
- You only need to store the single best route to a given user.
- You do not have to "jitter" your timer with randomized times.

6 Local Server–Daemon Protocol

This section describes the mini-protocol that an IRC Server uses to talk to the local routing daemon on the same node. It is important that you follow these specifications carefully because we will test your routing daemon independently of your IRC server!

The routing daemon listens on the local port when it starts up to service route lookup requests. When the IRC server on the same node starts up, it connects to the local port of the routing daemon. Since the local port is only supposed to service local client programs (like the IRC server) on the same machine that it trusts, you can assume that we won't do anything intentionally malicious to try to break it. However, you may find it useful to make it robust to invalid input, since you may make typos when testing it. Specifically, you can assume:

- We will only use the protocol as defined below. We will not send invalid requests.
- Only a single IRC server will connect to the routing daemon.
- Your IRC server may block while waiting for a response from the routing daemon. (i.e., you can

treat it as a function call)

This is a line-based protocol like the IRC-protocol itself.
Each request and response pair looks like this:

command arguments . . .
results . . .

Where *command* is the name of the request, *arguments . . .* is a space-separated list of arguments to the command, and *results . . .* is a space-separated list of results returned. All requests and responses are terminated with a newline character (\n) and are case sensitive, but some responses have multiple lines.
You must implement the following request/response pairs in your routing daemon:

Request: `ADDUSER` *nick*
Response: `OK`
Description: This request is issued when a new user is registered with the IRC server. The user's nick is added to the routing daemon's list of local users so that other nodes can find the user. This should trigger an immediate update for that nick.
Examples:

req: `ADDUSER bob`
resp: `OK`
req: `ADDUSER alice`
resp: `OK`

Request: `ADDCHAN` *channel*
Response: `OK`
Description: This request is issued when a new channel is created in the IRC server. The channel name is added to the routing daemon's list of local channels so that other nodes can find the channel. This should trigger an immediate update for that channel.
Examples:

req: `ADDCHAN #perl`
resp: `OK`
req: `ADDCHAN #networks`
resp: `OK`

Request: `REMOVEUSER` *nick*
Response: `OK`
Description: This request is issued when a local user leaves the IRC server. The user's nick is removed from the routing daemon's list of local destinations so that other nodes will know that they can no longer reach the user there. This should trigger an immediate update for that nick.
Examples:

req: `REMOVEUSER bob`
resp: `OK`
req: `REMOVEUSER baduser`
resp: `OK`

Request: `REMOVECHAN` *channel*
Response: `OK`
Description: This request is issued when the last local user leaves a channel. The channel name is removed from the routing daemon's list of local channels so that other nodes will know that they should no longer send channel messages to that server. This should trigger an immediate update for that channel.
Examples:

req: `REMOVECHAN bob`
resp: `OK`
req: `REMOVECHAN baduser`
resp: `OK`

Request: `NEXTHOP` *nick*
Response: `OK` *nodeID distance*
Response: `NONE`
Description: This request is used to find nodeID of the next hop to use if we want to forward a message to the user *nick*. It should return OK if the routing table has a valid next hop for the *nick* along with the distance to that destination, and NONE otherwise (e.g., if the destination's distance is not known or user does not exist).
Examples:

req: `NEXTHOP bob`
resp: `OK 2 5`
req: `NEXTHOP alice`
resp: `OK 3 2`
req: `NEXTHOP baduser`
resp: `NONE`

Request: `NEXTHOPS` *sourceID channel*
Response: `OK` *nodeID nodeID nodeID ...*
Response: `NONE`
Description: This request is used to find which links a server should send messages to if it wants to forward a message to a channel. It should return OK if the routing table has a valid entry for the channel from the given source node and then list the nodes to which it should propagate the message. Otherwise, it should return NONE (e.g., if the channel does not exist). See graph in Link-State section.
Examples:

req: `NEXTHOPS 1 #perl`
resp: `OK 2 5 9`
req: `NEXTHOPS 5 #perl`
resp: `OK 1`
req: `NEXTHOPS #badchan`
resp: `NONE`

Request: `USERTABLE`
Response: `OK` *size*
Description: If this request is issued, the routing daemon should respond with OK, the *size* or number of entries in the routing table, and a multi-line response with its entire user table in the following format:

```
nickname next-hop distance
nickname next-hop distance
nickname next-hop distance
...
```

Where *nick* is the nickname, *next-hop* is the nodeID of the next hop, and *distance* is the current distance value for that destination. You should not include local nicknames in this list. The order of entries does not matter. Your IRC Server will probably not need to use this command. We will use this to test your routing daemon. This would be similar to calling NEXTHOP on every user on the server.

Examples:

req: `USERTABLE`

resp:
```
OK 3
bob 2 2
alice 3 1
jim 3 2
```

Request: `CHANTABLE`
Response: `OK` *size*
Description: If this request is issued, the routing daemon should respond with OK, the *size* or number of entries in the channel table, and a multi-line response with its entire channel table in the following format:

```
channel sourceID next-hop next-hop next-hop ...
channel sourceID next-hop next-hop next-hop ...
channel sourceID next-hop next-hop next-hop ...
...
```

Where *channel* is the channel name, *sourceID* is the nodeID on which the message would come, and *next-hop* is a list of nodeIDs to which the server should propagate a message for that channel. You should not include channels that exist only locally in this list. The order of entries does not matter. Your IRC Server will probably not need to use this command. We will use this to test your routing daemon. This would be similar to calling NEXTHOPS on every channel on the server.

Examples:

req: `CHANTABLE`

resp:
```
OK 4
#perl 1 2 5 9
#perl 2 5 9
#perl 5 1 9
#perl 9 5 2
```

---

7 IRC Server (revisited)

Now that we have covered the IRC server, the routing protocols, and the server-daemon protocol, the only major issue remaining is how to extend your IRC Server to use the routing daemon so it can send messages to users on remote IRC Servers.

Remember that the `PRIVMSG` command has two targets: nicknames and channels. If the target is a nickname, the IRC server must first determine if there is a local user with that nickname. If not, then it should try to locate the user on a remote IRC Server (using the routing daemon) and, if found, forward the message to that IRC Server which will then send it to the target. If the target is not found, then you should

send the user an `ERR_NOSUCHNICK` error. If the target is a channel, then you must echo that message to every user on that channel.

7.1 Requirements

Your extensions to the IRC server should have the following features:

- Connect to the routing daemon's local port when it starts up. You can assume the routing daemon will be started first.
- When a new user is registered with the IRC server, it should add the user's nick to the routing daemon's list of users using the ADDUSER request.
- When a user leaves the IRC server, it should remove the user's nick from the routing daemon's list of users using the REMOVEUSER request.
- When a channel is created on the IRC server, it should send an ADDCHAN message to the routing daemon.
- When the last user leaves a channel on the IRC server, it should send a REMOVECHAN message to the routing daemon.
- If a user changes his or her nick, remove the old nick and add the new one to the routing daemon.
- When a PRIVMSG is sent to a nick that we don't know locally, the IRC Server should ask the routing daemon to find it, if possible, and forward the message to that user. The remote IRC server receiving the message should send it to the target user the same way it would send any other PRIVMSG to him or her.
- If the target is not found, then you should send the user an ERR_NOSUCHNICK error as defined in section 4.4.1 of the IRC RFC.
- The PRIVMSG command should support multiple targets; i.e., the PRIVMSG command may have a comma-separated list of target users or channels that should all be sent the message.
- If the routing daemon dies or you cannot communicate with it, your IRC server may exit.

You do not have to implement the following:

- Forwarding messages to target servers, host masks, or anything mentioned in the IRC RFC that is not mentioned in this document.

7.2 Message Forwarding

Once the IRC Server has found the next hop or route to a remote nickname, it must forward the message to the remote IRC Server. You are responsible for designing a protocol to be used between your IRC Servers for forwarding these messages so that they will reach the destination. Here are a couple things to keep in mind when designing your protocol:

- When using OSPF, you can only obtain the next hop from the routing daemon. Hence, each IRC server along the path will have to query its routing daemon to figure out where to send the packet next.
- When using OSPF, while forwarding, a node or virtual link may go down (or the target user may leave). In this circumstance, you can just drop the message. You do not have to inform the user that sent the message that it was dropped.
- You may have to send the message to multiple peers when forwarding to a remote channel.
- If the same nick is logged on to more than one IRC Server in the network, OSPF should find the route to the "closest" one. Your forwarding protocol only needs to forward the message to one of them.

- IRC Servers and virtual links may go down and come back up. If you detect that your neighbor is down (i.e., the socket is closed), you should check to see if they have come back up at least once every 3 seconds. In fact, when the network first starts up, since only one server will come up at a time, all its neighbors will appear to be down at first.
- You should not have IRC Servers communicate if they are not neighbors.
- Your forwarding protocol should not be "flood every message to every IRC server on the network." That is not efficient and doesn't require the routing layer at all.
- You should not rely on any special extensions to the local port mini-protocol. We may test your IRC Server on our own routing daemon.

8 Implementation Details & Usage

Your programs must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, only the standard libsocket, the provided library functions, and the `csapp` wrapper library developed for 15-213. You may use the `pthread` library, but you are responsible for learning how to use it correctly yourself if you choose to. To use the `csapp` wrapper library, you must link with `libpthread` (-lpthread). If you wish to use other libraries, please contact us.

8.1 Compiling

You responsible for making sure your code compiles and runs correctly on the Andrew x86 machines running Linux (i.e., linux.andrew.cmu.edu). We recommend using `gcc` to compile your program and `gdb` to debug it. You should use the `-Wall` flag when compiling to generate full warnings and to help debug. For this project, you will also be responsible for turning in a GNUMake (`gmake`) compatible Makefile. See the GNU make manual[4] for details. When we run `gmake` we should end up with the routing daemon which you should call `srouted` and the simplified IRC Server which is called `sircd`.

8.2 Command Line Arguments

Your routing daemon must take the following command line arguments in any order. We will provide you some framework code that will read in these arguments.

usage: `./srouted -i` *`nodeID`* `-c` *`config_file`* *`[options]`*

`-i` ***integer***
NodeID. Sets the nodeID for this process.

`-c` ***filename***
Config file. Specifies the name of the configuration file that contains the information about the neighbor nodes. The format of this file is described below.

It should also recognize the following optional switches:

`-a` ***seconds***
Advertisement cycle time. The length of time between each advertisement cycle. Defaults to 30.

[4] http://www.gnu.org/manual/software/make/html_mono/make.html

`-n` ***seconds***
Neighbor timeout. The elapsed time after which we declare a neighbor to be down if we have not received updates from it. You may assume that this value is a multiple of **advertisement cycle time**. Defaults to 120.

`-r` ***seconds***
Retransmission timeout. The elapsed time after which a peer will attempt to retransmit an LSA to a neighbor if it has not yet received an acknowledgement for that LSA. This value is an integral number of seconds. Defaults to 3.

`-t` ***seconds***
LSA timeout. The elapsed time after which we expire an LSA if we have not received updates for it. You may assume that this value is a multiple of **advertisement cycle time**. Defaults to 120.

Your IRC server will always have two arguments:

usage: `./sircd` *`nodeID config_file`*

nodeID
The nodeID of the node.

config_file
The configuration file name.

8.3 Configuration File Format

This file describes the *neighborhood* of a node. The neighborhood of a node 1 is composed by node 1 itself and all the nodes `n` that are directly connected to 1. For example, in Figure 4, the neighborhood of node 1 is {1, 2, 3}. The format of the configuration file very simple, and we will supply you with code to parse it. The file contains a series of entries, one entry per line. Each line has the following format:

```
nodeID hostname routing-port local-port IRC-port
```

`nodeID`
Assigns an identifier to each node.

`hostname`
The name or IP address of the machine where the neighbor node is running.

`local-port`
The TCP port on which the routing daemon should listen for the local IRC server.

`routing-port`
The port where the neighbor node listens for routing messages.

`IRC-port`
The TCP port on which the IRC server listens for clients and other IRC servers.

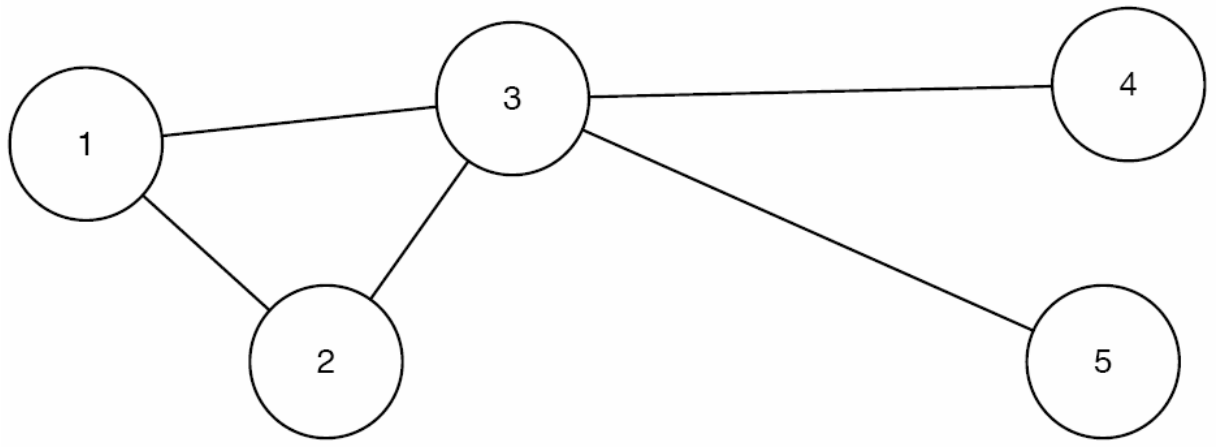

Figure 4 – Sample Node Network

Node 2	Node 5

`2 localhost 20203 20204 20205`
`1 unix1.andrew.cmu.edu 20200 20201 20202`
`3 unix3.andrew.cmu.edu 20206 20207 20208`
	`3 unix3.andrew.cmu.edu 20206 20207 20208`
	`5 localhost 20209 20210 20211`

Figure 5 – Sample Configuration file for nodes 2 and 5

How does a node find out which ports it should use as routing, IRC, and local ports? When reading the configuration file if an entry's `nodeID` matches the node's nodeID of the node (passed in on the command line), then the node uses the specified port numbers to route and forward packets. Figure 5 contains a sample configuration files corresponding to node 2 and node 5 for the network in Figure 4. Notice that the file for node 2 contains information about node 2 itself. Node 2 uses this information to configure itself.

We have provided you with a simple script called `genconfig.pl` that will auto-generate all the configuration files for a specified network graph, which you can find in the `./util` subdirectory of the handout. Read the text at the top of the script for documentation.

8.4 Running

This is how we will start your IRC network.

First, we start each routing daemon with the commands:

```
./srouted -i 0 -c node0.conf ... &
./srouted -i 1 -c node1.conf ... &
./srouted -i 2 -c node2.conf ... &
...
```

Each routing daemon will be started with its own configuration file to find out about its neighbors (described above) and its nodeID. In addition, we will pass it certain arguments to set the timer values.

Next, we will start each IRC server at each node:

```
./sircd 0 node0.conf &
./sircd 1 node1.conf &
./sircd 2 node2.conf &
...
```

Each IRC Server will be passed its nodeID and the configuration file to find out about its neighbors and what ports it should use/talk to. Now we will wait enough time such that the routing state should have converged and test your system. (We may also bring down nodes and restart them to test how resilient your system is to faults)

8.5 Framework Code

We have provided you with some framework code to simplify some tasks for you, like reading in the command line arguments and parsing the configuration file. You do not have to use any of this code if you do not want to. This code is documented in `rtlib.h` and implemented in `rtlib.c`. Feel free to modify this code also. However, you **must** use the following three routines, which are declared in `rtgrading.h` and implemented in `rtgrading.c`, and must **not** modify them:

- `rt_init(...)`: You must call this function when your routing daemon starts with the `argc` and `argv` passed to your program.

- `rt_sendto(...)`: Wrapper function for the `sendto()` system call. The parameters and semantics are the same as in the system call. You should use this function to send UDP packets in your routing daemon.

- `rt_recvfrom(...)`: Wrapper function for the `recvfrom()` system call. The parameters and semantics are the same as in the system call. You should use this function to receive UDP packets in your routing daemon.

We will replace `rtgrading.c` with implementations that we will use for grading so you should not modify it.

DISCLAIMER: We reserve the right to change the support code as the project progresses to fix bugs and to introduce new features that will help you debug your code. You are responsible for reading the b-boards to stay up-to-date on these changes. We will assume that all students in the class will read and be aware of any information posted to b-boards.

9 Testing

Code quality is of particular importance to server robustness in the presence of client errors and malicious attacks. Thus, a large part of this assignment (and programming in general) is knowing how to test and debug your work. There are many ways to do this; be creative. We would like to know how you tested your server and how you convinced yourself it actually works. To this end, you should submit your test code along with brief documentation describing what you did to test that your server works. The test cases should include both generic ones that check the server functionality and those that test particular corner cases. If your server fails on some tests and you do not have time to fix it, this should also be documented (we would rather appreciate that you know and acknowledge the pitfalls of your server, than miss them). Several paragraphs (or even a bulleted list of things done and why) should suffice for the test case documentation.

Daemon Debugging:

The daemon will have no user interface, but you can still telnet to the local port on your routing daemons to inject destinations, remove destinations, check routing tables, etc.

To test if your system can handle node faults, kill some of your routing daemons and IRC servers. To test if your system can handle link faults, try blocking off a pair of UDP ports between two routing daemons (You can do this artificially in your code by dropping packets that go between them).

10 Handin

Handing in code for checkpoints and the final submission will be done through your subversion repositories. You can check out your subversion repository with the following command where you must change `P2Group#` to correct numbers such as "`P2Group2`":

```
svn co https://moo.cmcl.cs.cmu.edu/441/svn/P2Group# - username P2Group#
```

The grader will check directories in your repository for grading, which can be created with an "`svn copy`":

```
Checkpoint 1 – YOUR REPOSITORY/tags/checkpoint1
Checkpoint 2 – YOUR REPOSITORY/tags/checkpoint2
Final Handin – YOUR REPOSITORY/tags/final
```

You should submit the following files:

- `Makefile` – Make sure all the variables and paths are set correctly such that your program compiles in the handin directory. The Makefile should build two executable named `srouted` and `sircd`.
- All of your source code (files ending with .c, .h, etc. only, no .o files and no executables)
- `readme.txt`: File containing a brief description of your design of your routing daemon and a complete description of the protocols you used for forwarding IRC messages.
- `tests.txt:` File containing documentation of your test cases and any known issues you have.

Late submissions will be handled according to the policy given in the course syllabus

11 Grading

- **OSPF User routing**: 15 points

  The OSPF routing protocol should find a route if it exists. If there is more than one, it should only accept one and ignore the others. If there is no route, it should timeout after a specified time and ignore any path it might receive after timeout. If there are two users, you should use only one path and ignore the others.

- **OSPF Channel routing**: 10 points

  The OSPF routing protocol should provide a list of nodes to which a channel message should be propagated. It must use shortest path finding and build the minimum spanning tree for each source node.

- **User Forwarding**: 15 points

  Using the PRIVMSG command with a nickname target, the server should communicate with the daemon to get a next-hop from the local server to the server where the destination resides. Then, you must send a packet using a protocol of your devising. When an IRC server gets a forwarding packet, it should deliver the message locally or query the local daemon for the next hop and propagate the message. The message should travel along the path returned by the daemons and should ultimately be received. If path fails, you can drop the message and do not have to return an error.

- **Channel forwarding**: 10 points

  Similarly, using the PRIVMSG command with a channel target, the server should communicate with the daemon to get a list of next-hops from the local server to the servers using that channel. Then, you must send a packet using a protocol of your devising. When an IRC server gets a forwarding packet, it should deliver the message locally and/or query the local daemon for the next list of hops given the source ID of the node from which it received the packet. If the path fails, you can drop the message and do not have to return an error.

- **Robustness**: 15 points
  - Server robustness: 8 points
  - Test cases: 7 points

  Since code quality is of a high priority in server programming, we will test your program in a variety of ways using a series of test cases. For example, we will send your server a message longer than 512 bytes to test if there is a buffer overflow. We will make sure that your server does something reasonable when given an unknown command, or a command with invalid arguments. We will verify that your server correctly handles clients that leave abruptly (without sending a QUIT message). We will test that your server correctly handles concurrent requests from multiple clients, without blocking inappropriately.[5]

  However, there are many corner cases that the RFC does not specify. You will find that this is very common in "real world" programming since it is difficult to foresee all the problems that might arise. Therefore, we will not require your server pass all of the test cases in order to get a full 15 points.

  We will also look at your own documented test cases to evaluate how you tested your work.

- **Style**: 15 points
  Poor design, documentation, or code structure will probably reduce your grade by making it hard for you to produce a working program and hard for the grader to understand it; egregious failures in these areas will cause your grade to be lowered even if your implementation performs adequately.

  To help your development and testing, we suggest your server optionally take a verbosity level switch (-v level) as the command line argument to control how much information it will print. For example, -v 0 means nothing printed, -v 0 means basic logging of users signing on and off, -v 2 means logging every message event.

- **Checkpoint**: 15 points

[5] As an exception to this rule, your server may block while doing DNS lookups.

Tests and extra credit sections need not be submitted. Just complete the scripts and mail the hash back. Late policy does not apply to the checkpoint. You may either submit on time or else you may not get the points applicable to the checkpoint. Core networking and IRC protocol on a standalone server will be tested for this checkpoint.

12 Getting Started

Depending on your previous experience, this project may be substantially larger than your previous programming projects. Expect the server implementation to require more than 1000 lines of code. With that in mind, this section gives suggestions for how to approach the project. Naturally, other approaches are possible, and you are free to use them.

- First, take a deep breath and do not panic.
- **Start early (again)!** The hardest part of getting started tends to be getting started. Remember the 90-90 rule: the first 90% of the job takes 90% of the time; the remaining 10% takes the other 90% of the time. Starting early gives your time to ask questions. For clarifications on this assignment, post to the main class bulletin board (academic.cs.15-441) and read project updates on the course web page. Talk to your classmates. While you need to write your own original program, we expect conversation with other people facing the same challenges to be very useful. Come to office hours. The course staff is here to help you.
- Decide how you will split up the work between you and your partner. **You can start with either partner's IRC server as the starting point, whichever one works best.** Some parts of this project can be done in parallel, but you should coordinate since they both have to work with the same program. Both of you should understand everything implemented for this project.
- Once the simple IRC server is complete (it should be by now!), you can begin extending it with interfaces between the IRC server and your routing daemon. Since the routing daemon and IRC server are two programs, it is probably best to have the two thoroughly tested independently to trap errors more efficiently. For this purpose, we will release a routing daemon binary whereby you can first only implement the IRC server interfaces to the routing daemon, and test the interfaces between a standalone pair of IRC server and the given routing daemon binary.
- Once the IRC server extensions are done, you should start worrying about the routing daemon. First, get familiar with UDP socket programming, which is almost identical to TCP socket. Next, write up a design of each part of the routing daemon and decide what data structures you will need. First tackle general flooding and table construction. Work on getting the link entry and user entry tables functional. Once the protocol works for messages between users, then start working on multicasting, minimum spanning trees, shortest path finding, and multicasting.
- Again, thoroughly test the routing daemon. Telnet is a very useful tool for this. Make you're your daemon can add paths, remove paths, find paths, withstand failures, and does not segfault.
- Before you start implementing message forwarding in your IRC Server, carefully design a protocol. You might need to differentiate between users and channels since channels need to be multicasted. Outline what parts of the original IRC server need to be modified in order to connect and talk to the routing daemon.
- Almost there! Hopefully, after implementing the message forwarding protocol and server extensions everything will work perfectly. More likely, though, something will break. Things that work perfectly separately do not always work perfectly together. This is a big software engineering problem. So, yet again, thoroughly test the final product. Run the same tests you used

on the individual pieces to make sure nothing broke in the merge.

- You may use some of the system call wrappers provided by CS 15-213 `csapp` library (included with the simple IRC client package). However, for server robustness, you should not use certain wrappers such as `Select` since temporary system call failures (e.g., `EINTR`) would cause the server to abort. Instead, you server should handle such errors gracefully. For the same reason, you should NOT use the `RIO` read/write functions provided by the `csapp` library as they may cause your server to block while reading/writing, or give inappropriate return codes.
- "Be liberal in what you accept, and conservative in what you send."[6] Following this guiding principle of Internet design will help ensure your server works with many different and unexpected client behaviors.
- Code quality is important. Make your code modular and extensible where possible. You should probably invest an equal amount of time in testing and debugging as you do writing. Also, debug incrementally. Write in small pieces and make sure they work before going on to the next piece. Your code should be readable and commented. Not only should your code be modular, extensible, readable, etc, most importantly, **it should be your own**!

13 Resources

For information on network programming, the following may be helpful:

- Class Textbook – Sockets, OSPF, etc
- Class B-board – Announcements, clarifications, etc
- Class Website – Announcements, errata, etc
- Computer Systems: A Programmer's Perspective (CS 15-213 text book)[7]
- BSD Sockets: A Quick And Dirty Primer[8]
- An Introductory 4.4 BSD Interprocess Communication Tutorial[9]
- Unix Socket FAQ[10]
- Sockets section of the GNU C Library manual
  - Installed locally: info libc
  - Available online: GNU C Library manual[11]
- man pages
  - Installed locally (e.g. man socket)
  - Available online: the Single Unix Specification[12]
- Google groups[13] - Answers to almost anything

6 http://www.ietf.org/rfc/rfc1122.txt, page 11
7 http://csapp.cs.cmu.edu
8 http://www.frostbytes.com/~jimf/papers/sockets/sockets.html
9 http://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf
10 http://www.developerweb.net/forum/forumdisplay.php?s=f47b63594e6b831233c4b8ebaf10a614&f=70
11 http://www.gnu.org/software/libc/manual/
12 http://www.opengroup.org/onlinepubs/007908799/
13 http://groups.google.com