# 15-441: Computer Networks
# Project 1: IRC and Routing

Lead TA: Daniel Spangenberger<dspangen@andrew.cmu.edu>

**Assigned: September 4, 2007**
**Checkpoint 1 due: September 11, 2007**
**Checkpoint 2 due: September 18, 2007**
**Final version due: September 25, 2007**

## 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]. Future extensions (Project 2) will allow you to build on this IRC server to allow global routing of messages between servers.

IRC is a global, distributed, real-time chat system that operates over the Internet. An IRC network consists of a set of interconnected servers. Once users are connected to an IRC server, they can converse with other users connected to any server in the IRC network. IRC provides for group communication, via named channels, as well as personal communication through "private" messages. For more information about IRC, including available client software and public IRC networks, please see The IRC Prelude[2].

If you have not used IRC before, you may want to try it out to get a feel for what it is. For a quick start, log in to an Andrew machine, and run *irssi -c irc.freenode.net -n nickname* where *nickname* is the nickname you want to use. Then type */join #networking* to join a networking discussion channel. Other channels you might be interested include #gentoo, #redhat, #perl, and #c++. After you have tried out the text mode IRC client, you may want to try out graphical clients such as xchat and chatzilla (part of mozilla).

## 2 Logistics

- The tar file for this project can be found here: *http://www.cs.cmu.edu/∼srini/15-441/F07/project1/project1.tar.gz*.
- This is a group project. You *must* find exactly one partner for this assignment. The only reason you should not have a partner is if there are an odd number of people in

the class and you are left out (in which case contact us). Talk to your neighbors and use the bboards.

- Once you have found a partner, email Albert Sheu<albert@cmu.edu>your names and andrew logins so we can assign a group number to you. Use "15441 GROUP" as the subject. 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. We recommend adhering to a schedule like:

| date | milestone |
|---|---|
| 9/4 | project assigned |
| 9/7 | read and understand the project handout |
| 9/11 | **checkpoint due** – version control basics |
| 9/18 | **checkpoint due** – handle multiple clients |
| 9/20 | simple standalone irc server complete |
| 9/23 | simple standalone irc server tested thoroughly |
| 9/25 | last minute rush to get things done and hand-in |

## 3 Overview

An IRC network is composed of a set of nodes interconnected by virtual links in an arbitrary topology. Each node runs a process that we will call a routing daemon. Each routing daemon maintains a list of IRC users available to the system. Figure 1 shows a sample IRC network composed of 5 nodes. The solid lines represent virtual links between the nodes. Each node publishes a set of users (i.e., the nicks of the IRC clients connected to it) to the system. The dotted lines connect the nodes to their user sets.

The usage model is the following: If Bob wants to contact Alice, the IRC server on the left first must find the route or path from it to the node on the right. Then, it must forward Bob's message to each node along the path (the dashed line in the figure) until it reaches the IRC server at Alice's node, which can then send the message to the client Alice.

In essence, each node in the system performs functions similar to the ones performed in the network layer, namely forwarding and routing. Forwarding is the action performed by each node to guide a packet toward its destination. Routing refers to the action of building the data structures necessary to reach particular destinations (in terms of the IRC server, a destination is a username/nick).

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.

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.

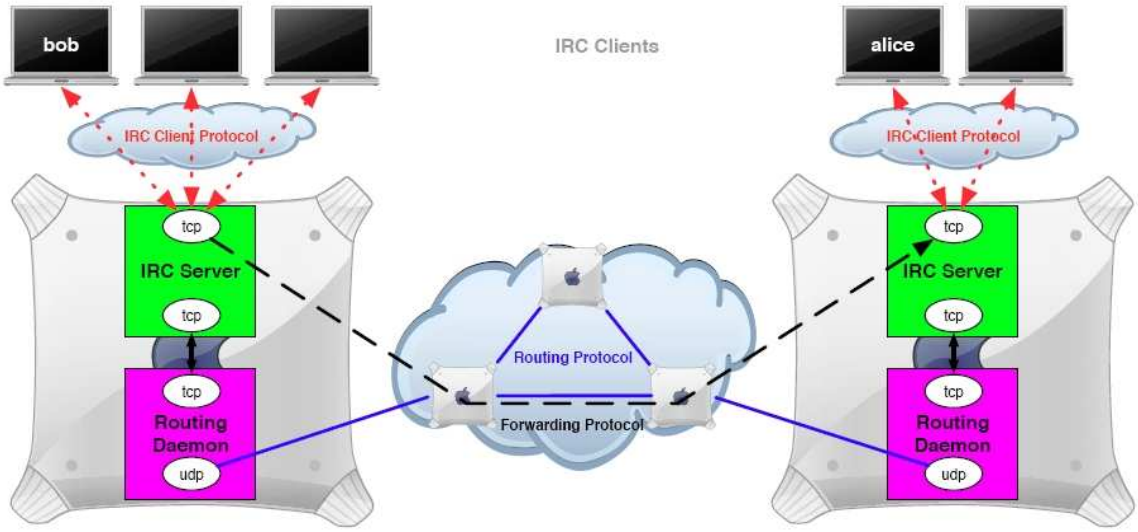


**Figure 1: IRC network**

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 the 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

- **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 its nodeID.
- **nodeID** – 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** – IRC nickname or channel 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 you will implemenit
- **routing table** – The data structure used to store the "next hops" that packet should take used in OSPF. See your textbook pp. 274–280 for description.

## 5 The IRC Server

Your server will implement a subset of the original IRC protocol. The original IRC protocol is defined in RFC 1459[3]. Because RFC 1459 omits some details that are required to implement an IRC server, we have provided an annotated version of the RFC[4]. For this project, you should **always refer to the annotated version of the RFC**, not the original version. We have chosen a subset of the protocol that will provide you with experience developing a concurrent network application without spending an inordinate amount of time implementing lots of features. Specifically, your server must implement the following commands:

**Basic Commands**

- **NICK** – Give the user a nickname or change the previous one. Your server should handle duplicate nicknames and report an error message.
- **USER** – Specify the username, hostname, and real name of a user.
- **QUIT** – End the client session. The server should announce the clients departure to all other users sharing the channel with the departing client.

**Channel Commands**

- **JOIN** – Start listening to a specific channel. Although the standard IRC protocol allows a client to join multiple channels simultaneously, your server should restrict a client to be a member of at most one channel. Joining a new channel should implicitly cause the client to leave the current channel.
- **PART** – Depart a specific channel. Though a user may only be in one channel at a time, PART should still handle multiple arguments. If no such channel exists or it exists but the user is not currently in that channel, send the appropriate error message.
- **LIST** – List all existing channels on the local server only. Your server should ignore parameters and list all channels and the number of users on the local server in each channel.

**Advanced Commands**

- **PRIVMSG** – Send messages to users. The target can be either a nickname or a channel. If the target is a channel, the message will be broadcast to every user on the specified channel, except the message originator. If the target is a nickname, the message will be sent only to that user.
- **WHO** – Query information about clients or channels. In this project, your server only needs to support querying channels on the local server. It should do an exact match on the channel name and return the users on that channel.

For all other commands, your server must return ERR_UNKNOWNCOMMAND. If you are unable to implement one of the above commands (perhaps you ran out of time), your server must return the error code ERR_UNKNOWNCOMMAND, rather than failing silently, or in some other manner.

Your server should be able to support multiple clients concurrently. The only limit to the number of concurrent clients should be the number of available file descriptors in the operating system (the min of ulimit -n and FD_SETSIZE – typically 1024). While the server is waiting for a client to send the next command, it should be able to handle inputs from other clients. Also, your server should not hang up if a client sends only a partial command. In general, concurrency can be achieved using either select or multiple threads. However, in this project, you **must implement your server using select to support concurrent connections**. See the resources section below for help on these topics.

As a public server, your implementation should be robust to client errors. For example, your server should be able to handle multiple commands in one packet. It should not overflow any buffers when the client sends a message that is too long (longer than 512 bytes). In general, your server should not be vulnerable to a malicious client. This is something we will test for.

Note your server behaves differently from a standard IRC server for some of the required commands (e.g., JOIN). Therefore, you should not use a standard IRC server as your reference for your implementation. Instead, refer to the annotated version of the RFC on the

course web page. Testing and debugging of your IRC server can be done with our provided sircc client provided (discussed later in section 10), or a telnet client for issuing commands and receiving responses.

# 6 Implementation Details and 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, and the provided library functions. While the CS-APP header files from 213 may look enticing you may not use them–their error handling functionality is not robust enough for this project. You may use the pthread library, but you are responsible for learning how to use it correctly yourself if you choose to (link with libpthread -lpthread). You may use a standard linked list and hash table library; we suggest SimCList[13] and libghthash[14]. We are not, however, supporting these libraries in any way–if you have trouble using them you must figure it out yourself, switch to a new library, or write your own. Additionally, if you wish to use other libraries, please contact us.

## 6.1 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. Other tools available on the Andrew unix machines that are suggested are ElectricFence[8] (link with -lefence) and Valgrind[9]. Valgrind is superior to ElectricFence and generally easier to use. These tools will detecting overflows and memory leaks respectively. For this project, you will also be responsible for turning in a GNUMake (gmake) compatible Makefile. See the GNU make manual[6] for details. When we run gmake we should end up with the simplified IRC Server which is called sircd.

## 6.2 Command Line Arguments

**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.

## 6.3 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. Note that it is safe to ignore both the routing-port and local-port at this point. These will become important for project 2. 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.

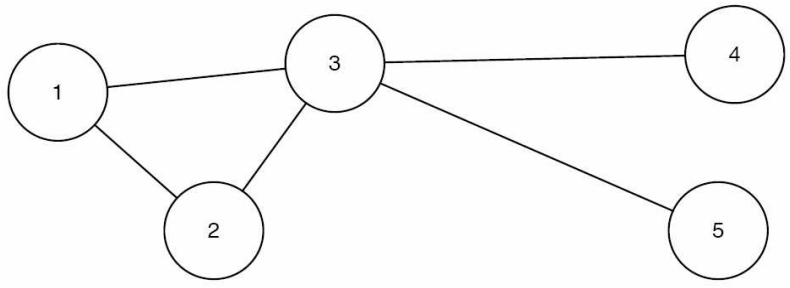


**Figure 4 – Sample Node Network**

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

**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.

### 6.4 Running

This is how we will start your IRC server:

*./sircd 0 node0.conf &*

Your IRC Server will be passed its nodeID and the configuration file to find out what ports it should use/talk to.

### 6.5 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.

**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.

## 7 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.

Additionally you must also submit automated test cases that you have developed to test your code. These can be written using the test harness that we will release by checkpoint

2, or with your own. See below for some examples of testing infrastructures that you may use. You will be judged on the thoroughness of your test cases. Full credit will be reserved for those groups that thoroughly test their server along edge cases as well as providing automated performance and stability tests.

To help you get started on testing, we have provided a simple IRC client sircc and several example test scripts. These will give you an idea of what tests we will use to evaluate your work, and ensure that you are on the right track with your server.

**sircc:**

The *sircc* program takes input from stdin as client commands to send to the server, and echoes server reply on the screen. This can be used to check the exact formats of responses from your server and test how your server behaves when given input is not compliant with the IRC specification.

unix>./sircc -h
usage: sircc <ip_address> <port>

When using *sircc*, <*ip_address*> and <*port*> are the address and port number of your IRC server. By default, the address is set to your local machine and the port number is 6667.

**Ports:**

To facilitate the development, you can run all your routing engines in the same machine using different ports. In order to avoid port collisions among routing engines from different groups, use port numbers in the range:

[20,000 + group number * 100 , 20,000 + group number * 100 + 99]

For example, if your group number is 23, then you should choose the port numbers from the following range: [22300, 22399]. If you are testing a configuration with 3 nodes, then the node addresses could be the following:

| Node ID | IP Address | Routing Port | Local Port | IRC Port |
|---|---|---|---|---|
| 1 | 127.0.0.1 | 22300 | 22301 | 22302 |
| 2 | 127.0.0.1 | 22303 | 22304 | 22305 |
| 3 | 127.0.0.1 | 22306 | 22307 | 22308 |

**IRC Test scripts:**

The test scripts test your IRC server against different types of commands.

For example, login.exp checks the replies of the command NICK and USER.

unix>./login.exp
usage: login.exp <host> <port>

Here <*host*> and <*port*> are the address and port number of your IRC server.

You may use the provided test scripts as a base to build your own test case. You may also find the following tools to be useful in your test code development:

**expect**

Quoting from the expect man page,

Expect is a program that "talks" to other interactive programs according to a script. Following the script, Expect knows what can be expected from a program and what the correct response should be. An interpreted language provides branching and high-level control structures to direct the dialogue.

**Net:::IRC**

A Perl module that simplifies writing an IRC client. Net:::IRC is not installed on the Andrew Linux machines, but you can download Net:::IRC from the Comprehensive Perl Archive Network (CPAN).

Note that Net:::IRC and a command line IRC client both implement the client-side IRC protocol for you. Presumably, they interact with the server in a standards-compliant manner.

# 8 Handin

Handing in code for checkpoints and the final submission deadline will be done through your subversion repositories. You can check out your subversion repository with the following command where you must change your Team# to "Team1" for instance, and your P# to the correct number such as "P1":

*svn co https://moo.cmcl.cs.cmu.edu/441/svn/Project1Team# –username Project1Team#P#*

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

Your repository should contain 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 an executable named 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.
- **extra.txt** – (optional) Documentation on any extra credit items you have worked on.

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

# 9 Grading

- **Server core networking:** 20 points

  The grade in this section is intended to reflect your ability to write the "core" networking code. This is the stuff that deals with setting up connections, reading/writing from them (see the resources section below). Even if your server does not implement any IRC commands, your project submission can get up to 20 points here. Thus it is better to have partial functionality working solidly than lots of code that doesn't actually do anything correctly.

- **Server IRC protocol:** 20 points

  The grade in this section reflects how well you read, interpreted, and implemented the IRC protocol. We will test that all the commands specified in the project handout work. All commands sent to your server for this part of the testing will be valid. So a server that completely and correctly implements the specified commands, even if it does not check for invalid messages, will receive 20 points here. To receive full credit here, your server must accept a connection from irssi (*irc* client on andrew machines) and be able to complete all required IRC commands as sent by this client.

- **Robustness:** 25 points
  - Server robustness: 13 points
  - Test cases: 12 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. The only exception is that your server may block while doing DNS lookups

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 25 points.

We will also look at your own documented test cases to evaluate how you tested your work as well as your written executable tests. We will ensure that your tests actually complete successfully. We will also check your programs for memory leaks using Valgrind. If your program leaks excessively (>2% of all allocated memory) you will lose points.

- **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. More granular logging strategies are possible and suggested.

- **Checkpoints: 20 points**

  Tests and extra credit sections need not be submitted. 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.

## 10 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!** 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. It is recommended that both work together on the IRC server since it is an important part of the project and must be understood for the routing protocol. 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.
- Read the revised RFC selectively. RFCs are written in a style that you may find unfamiliar. However, it is wise for you to become familiar with it, as it is similar to the styles of many standards organizations. We dont expect you to read every page of the RFC, especially since you are only implementing a small subset of the full protocol, but you may well need to re-read critical sections a few times for the meaning to sink in.
- Begin by reading Sections 1-3. Do not focus on the details; just try to get a sense of how IRC works at a high level. Understand the role of the clients and the server. Understand what nicknames are, and how they are used. You may want to print the RFC, and mark it up to indicate which parts are important for this project, and which parts are not needed. You may need to reread these sections several times.
- Next, read Section 4 and 6 of the RFC. You will want to read them together. In general, Section 4 describes the purpose of the commands in the IRC protocol. But the details on the possible responses are given in Section 6. Again, do not focus on the details; just try to understand the commands at a high level. As before, you may want to mark up a printed copy to indicate which parts of the RFC are important for the project, and which parts are not needed.
- Now, go back and read Section 1-3 with an eye toward implementation. Mark the parts which contain details that you will need to write your server. Read project related parts

in sections 4 and 6. Start thinking about the data structures your server will need to maintain. What information needs to be stored about each client?

- Get started with a simple server that accepts connections from multiple clients. It should take any message sent by any client, and "reflect" that message to all clients (including the sender of the message). This server will not be compatible with IRC clients, but the code you write for it will be useful for your final IRC server. Writing this simpler server will let you focus on the socket programming aspects of a server, without worrying about the details of the IRC protocol. Test this simple server with the simple IRC client sircc. A correct implementation of the simple server gives you approximately 20 points for the core networking part.

- At this point, you are ready to write an IRC server. This simple IRC server should run alone and implement all the features of the IRC protocol outlined above. In particular, it should allow users to log in, join channels, list channels, and send messages to either users or channels. This will go a very good start towards your final IRC server and will give you an idea of how your final IRC server should work. Make sure this server is solid. You will be building upon it in project 2 and you do not want to waste time trying to figure out if a particular error is in the IRC server or the routing daemon. Remember only to use ports in the range of your group number to avoid weird conflicts!

- Do not try to write the whole server at once. Decompose the problem so that each piece is manageable and testable. Read related parts of RFC again carefully and think about how the commands work together. For each command, identify the different cases that your server needs to handle. Find common tasks among different commands and group them into procedures to void writing the same code twice. You might start by implementing the routines that read and parse commands. Then implement commands one by one, testing each with the simple client sircc or telnet.

- Thoroughly test the IRC server. Use the provided scripts to test basic functionality. For further testing, use the provided sircc client or telnet. It may be useful to learn the basics of a scripting language to make some repeatable "regression tests." As said, the routing daemon will build upon the IRC server, so thorough testing of the latter will save time in debugging the former. Also, think up interesting corner test cases and implement them as automated tests.

- Be liberal in what you accept, and conservative in what you send[7]. 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!

## 11 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)[10]
- BSD Sockets: A Quick And Dirty Primer[11]
- An Introductory 4.4 BSD Interprocess Communication Tutorial[12]
- Unix Socket FAQ[15]
- Sockets section of the GNU C Library manual
  - Installed locally: info libc
  - Available online: GNU C Library manual[16]
- man pages
  - Installed locally (e.g. man socket)
  - Available online: the Single Unix Specification[17]
- Google groups - Answers to almost anything[18]

## References


[1] IRC RFC: *http://www.irchelp.org/irchelp/rfc/*

[2] The IRC Prelude: *http://www.irchelp.org/irchelp/new2irc.html*

[3] RFC 1459: *http://www.ietf.org/rfc/rfc1459.txt*

[4] Annotated RFC: http://www.cs.cmu.edu/∼srini/15-441/F07/project1/rfc.html

[5] OSPF RFC: *http://www.rfc-editor.org/rfc/rfc2328.txt*

[6] GNU Make Manual: *http://www.gnu.org/manual/software/make/html_mono/make.html*

[7] RFC 1122: *http://www.ietf.org/rfc/rfc1122.txt, page 11*

[8] ElectricFence: *http://perens.com/FreeSoftware/ElectricFence/*

[9] Valgrind: *http://valgrind.org/*

[10] CSAPP: *http://csapp.cs.cmu.edu*

[11] *http://www.frostbytes.com/ jimf/papers/sockets/sockets.html*

[12] *http://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf*

[13] C Linked List Library: *http://mij.oltrelinux.com/devel/simclist/*

[14] C Hash Table Library: *http://www.ipd.bth.se/ska/sim_home/libghthash.html*

[15] *http://www.developerweb.net/forum/forumdisplay.php?s=f47b63594e6b831233c4b8ebaf10a614&f=70*

[16] *http://www.gnu.org/software/libc/manual/*

[17] *http://www.opengroup.org/onlinepubs/007908799/*

[18] *http://groups.google.com*