Project 1: Internet Relay Chat (IRC) Server

Assigned: February 15, 2016
Checkpoint 1 due: February 22, 2016
Checkpoint 2 due: February 29, 2016
Final version due: March 9, 2016

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].
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 a 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 mIRC, xchat, and chatzilla (part of mozilla).

2 Logistics

- The tar file for this project can be found on NYUclasses under resources (project1.tgz)
- This is a solo project. You must implement and submit your own code.

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 this project, you only need to implement a standalone IRC server. You can assume that there is only one IRC server and all clients are connected to the server.

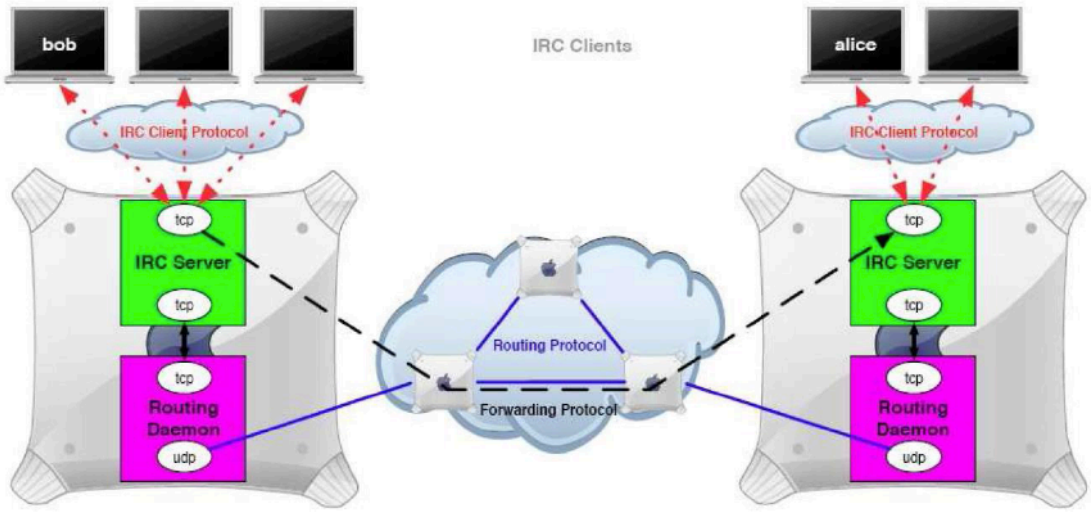

Figure 1: IRC network

4 Definitions

- **nodeID** – unique identifier that identifies an IRC server, or node. In the first project, the nodeID for the standalone IRC server should be 1.
- **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.

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 report an error message if a user attempts to use an already-taken nickname.

- **USER** – Specify the username, hostname, and real name of a user.

- **QUIT** – End the client session. The server should announce the client's 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 – both 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**. Threads are **NOT** permitted at all for the project. 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 must 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 included with the tarball. Testing and debugging of your IRC server can be done with our provided sircc client provided (discussed later in section 7), or a telnet client for issuing commands and receiving responses.

6 Implementation Details and Usage

Your server must be written in the C programming language. You are not allowed to use any custom socket classes or libraries, only the standard socket library and the provided library functions. You **may not** use the csapp wrapper library from 15-213, or libpthread for threading. We disallow csapp.c for two reasons: first, to ensure that you understand the raw standard BSD sockets API, and second, because csapp.c's wrapper functions are not suitable for robust servers. Temporary system call failures (e.g., EINTR) in functions such as Select could cause the server to abort, and utility functions like rio readlineb are not designed for nonblocking code.

That said, it is acceptable to use csapp.c, other libraries, or other languages in test code, which is separate from your server.

6.1 Compiling

You responsible for making sure your code compiles and runs correctly (we will provide soon an Ubuntu image that you can use for testing). 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 unix machines that are suggested are ElectricFence [7] (link with -lefence) and Valgrind [8]. These tools will help detect 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 [5] for details. When we run gmake we should end up with the simplified IRC Server, which you **must call 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, should be 1 for the standalone IRC server.
config file – The configuration file name.

6.3 Configuration File Format

This file describes the neighborhood of a node. In this project, since there are no neighbors other than the standalone server itself, the file is used to specify the RFC port used by the server. The format of the configuration file is very simple, and we will supply you with code to parse it. Note that most of the fields will become important only in the next project. Each line has the following format:

nodeID hostname routing-port local-port IRC-port

nodeID - An identifier of the server (it should match the nodeID argument when starting the IRC server)
Hostname - The name or IP address of the machine where the neighbor node is running, it should belocalhost for the standalone server.
local-port - The TCP port on which the routing daemon should listen for the local IRC server, and you can ignore this.
routing-port - The port where the neighbor node listens for routing messages, and you can ignore this.
IRC-port - The TCP port on which the IRC server listens for clients and other IRC servers, you should specify the port to use.

How does a node find out which ports it should use as IRC 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 number as RFC port.

6.4 Running

This is how we will start your IRC network.

./sircd 1 node1.conf

The IRC Server will be passed its nodeID and the configuration file to find out about 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, parsing the configuration file, and parsing IRC commands. You do not have to use any of this code if you do not want to. This code is documented in rtlib.h & irc proto.h and implemented in rtlib.c & irc proto.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.

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.

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.

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
From the 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 Linux Lab 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.

Handing in code for checkpoints and the final submission deadline will be done through your subversion repositories. You can setup a private repository on bitbucket and share it with yasir.zaki@nyu.edu and nabil.rahiman@nyu.edu.
You can check out your subversion repository with the following command, where you must change Project1Name# to correct numbers as "Project1Yasir":
svn co https://moo.cmcl.cs.cmu.edu/441-s10/svn/Project1Name# –username netid

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

- Checkpoint1 – YOUR REPOSITORY/tags/checkpoint1
- Checkpoint2 – 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 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 IRC server.

- **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.

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: 25 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.

- **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.

- **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 1 means basic logging of users signing on and off, -v 2 means logging every message event.

- **Checkpoint: 15 points**
  Tests section need not be submitted. Late policy DOES apply to the checkpoint. However, considering the fact that you only have 5 late days for the entire semester, we strongly encourage you to plan ahead and not to use late days for checkpoints. In the first checkpoint, you need to checkout and

checkin files from your repository. Core networking, i.e. server handling multiple clients, will be tested for the second checkpoint.

10 Getting Started

This section gives suggestions for how to approach the project. Naturally, other approaches are possible, and you are free to use them.

- 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. 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.

- 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 don't 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 of RFC. 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 a standalone IRC server. But 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 next project will build upon the standalone IRC server, so thorough testing will save time in debugging in the next project.

- Make sure to check the return code of all system calls and handle errors appropriately. Temporary failures (e.g., EINTR) should not cause your server to abort.

- 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!

- You may want to consider turning warnings into errors to avoid bad programming style. Do this by passing -Werror to gcc during compilation.

11 Resources

- For information on network programming, the following may be helpful:
- Class Textbook – Sockets, etc
- Computer Systems: A Programmer's Perspective (CSO text book) [9]
- BSD Sockets: A Quick and Dirty Primer [10]
- An Introductory 4.4 BSD Interprocess Communication Tutorial [11]
- Unix Socket FAQ [12]
- Sockets section of the GNU C Library manual
  - Installed locally: info libc
  - Available online: GNU C Library manual [13]
- man pages
  - Installed locally (e.g. man socket)
  - Available online: the Single Unix Specification [14]
- Google groups - Answers to almost anything [15]

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: in the tarball
[5] GNU Make Manual: http://www.gnu.org/manual/software/make/html mono/make.html
[6] RFC 1122: http://www.ietf.org/rfc/rfc1122.txt, page 11
[7] ElectricFence: http://perens.com/FreeSoftware/ElectricFence/
[8] Valgrind: http://valgrind.org/
[9] CSAPP: http://csapp.cs.cmu.edu
[10] http://www.frostbytes.com/ jimf/papers/sockets/sockets.html
[11] http://docs.freebsd.org/44doc/psd/20.ipctut/paper.pdf
[12]http://www.developerweb.net/forum/forumdisplay.php?s=f47b63594e6b831 233c4b8ebaf10a614&f=70
[13] http://www.gnu.org/software/libc/manual/
[14] http://www.opengroup.org/onlinepubs/007908799/
[15] http://groups.google.com