15-441 Project 1: Simplified IRC Server

Assigned: Tue, 20 Jan 2004
Due: Monday, 9 February 2004, 11:59pm

Lead TA: John Heffner <heffner@cmu.edu>

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, which implements a subset of the [Internet Relay Chat protocol (IRC)](http://www.irchelp.org/irchelp/rfc/).

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 (irchelp.org)](http://www.irchelp.org/irchelp/new2irc.html).

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 `irc <nickname> irc.debian.org` where `<nickname>` is the nickname you want to use. Then type `/join #c<cr>` to join the C programming channel. Other channels you might be interested include #debian, #redhat, #perl, #Linux, and #c++. After you have tried out the text mode IRC client, you may want to try out more elaborate clients, such as xchat and chatzilla (part of mozilla).

2 Logistics

The tar file for this project can be retrieved from

http://www.cs.cmu.edu/~srini/15-441/S04/assignments/project1/project1.tar.gz

---

1 http://www.irchelp.org/irchelp/rfc/
2 http://www.irchelp.org/irchelp/new2irc.html
3 Your Assignment

3.1 The Server

Your server will implement a subset of the original IRC protocol. The original IRC protocol is defined in RFC 1459. Because RFC 1459 omits some details that are required to implement an IRC server, we have provided an annotated version of the RFC. 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. While the standard IRC architecture includes multiple servers, the server you implement for this project will be stand-alone. Thus, the parts of the RFC which deal with server to server communication do not apply to this assignment. However, your server will need to support basic group communication with channels. Specifically, your server must implement the following commands:

Basic commands:

- **NICK**: give user a nickname or change the previous one. You should handle duplicate nicknames and report an error message.
- **USER**: specify username, hostname, and real name of a user.
- **QUIT**: end the client session. The server should inform the client’s departure to all other users sharing the channel with the departing client.

Channel commands:

- **JOIN**: start listening a specific channel. Although the standard IRC protocol allows a client to join multiple channels simultaneously, your server should restrict a client to be the 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. Since a user may be in only a single channel, this need take only a single argument.
- **LIST**: list existing channels. In this project, your server should ignore parameters and list all existing channels.

Advanced commands:

- **PRIVMSG**: send messages to users. In this project, we only require the case where the target is a channel. A message will be broadcast to every user on the specified channel, except the message originator.

3http://www.ietf.org/rfc/rfc1459.txt
4http://www.cs.cmu.edu/~srini/15-441/S04/assignments/project1/rfc-html/index.html
WHO: query information about clients or channels. In this project, your server only needs to support querying channels.

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 number of concurrent clients should only be limited by 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. You should also make sure your server doesn’t hang up if a client sends only a partial command. In general, you could use either select or multiple threads. However, in this project, you must implement your server using select to support concurrent connections from multiple clients. 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.

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.

3.2 Test Cases

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 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(project1/simple_client) and several example test scripts (project1/test_case). 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 echos 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.

Test scripts:

The test scripts test your 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:

- **irc, xchat, chatzilla** Various IRC clients. irc is a text mode client, while xchat and chatzilla have graphical interfaces. Since you are writing a genuine IRC server, you should be able to test it with actual IRC clients.

- **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 duologue.

- **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](http://search.cpan.org/CPAN/authors/id/J/JE/JEEK/Net-IRC-0.73.tar.gz) 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.

## 4 Evaluation

- Core networking: 15 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 15 points here. Thus it is better to have partial functionality working solidly than lots of code that doesn’t actually do anything correctly.

---

5 [http://www.fnal.gov/docs/products/expect/expect.1.html](http://www.fnal.gov/docs/products/expect/expect.1.html)
6 [http://www.perldoc.com/perl5.8.0/pod/perlfaq1.html](http://www.perldoc.com/perl5.8.0/pod/perlfaq1.html)
7 [http://search.cpan.org/CPAN/authors/id/J/JE/JEEK/Net-IRC-0.73.tar.gz](http://search.cpan.org/CPAN/authors/id/J/JE/JEEK/Net-IRC-0.73.tar.gz)
• 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 ircII and be able to complete all required IRC commands as sent by this client.

• Robustness: 15 points
  
  – Server robustness: 10 points
  – Test cases: 5 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.

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: 10 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 (\(-v\ \text{level}\)) 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.

• Extra credit (see Section\[\text{7}\]): up to 6 points

\[\text{8}\] Just run ‘irc’ from the Andrew Linux command line.
\[\text{9}\] As an exception to this rule, your server may block while doing DNS lookups.
5 Handin

5.1 Invocation

The executable for your server must be named `sircd` (for Simplified IRC Daemon). It should accept the port number as the only argument. For example, if you choose 9000 as the port number, you should be able to run your server using the following command:

```
./sircd 9000
```

Without argument, your server should by default work with port number 6667.

5.2 Code Requirements and Restrictions

We will test your program on x86 computers running Linux on the Andrew systems. Thus, you must make sure your code runs correctly on the Andrew Linux machines.

You must write your server in C, using the standard libc library. If you choose, you may also make use of the csapp wrapper library developed for 15-213. (A copy of this library is included with the simple IRC client in the project distribution.) To use this library, you will also need to link with libpthread(-lpthread). Please contact us if you want to use other libraries. Your code should compile with `gcc` using the `–Wall` flag cleanly on an Andrew Linux machine without any warning messages.

5.3 Building

Your project must include a makefile called `Makefile`. We will build a binary from your source code using the makefile and GNU make. The makefile for this project should be simple. If you need help creating the makefile, everything you need to know (and much much more) about GNU make can be found in the GNU make manual.¹¹

We will build your program by executing `rm -f sircd *.o` and then `make sircd`.

5.4 Hand-in Procedure

You should submit the following files:

- Makefile, *.c, *.h
- Test code and the corresponding documentation named as `tests.txt`
- (optional) Documentation (extra.txt) on any extra credit items you have worked on (see Section 7).

The detailed hand-in procedure will be posted on the course web page later. Late submissions will be handled according to the policy given in the course syllabus.

¹⁰Specifically, the GNU C Library.
6 Resources and Hints

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.

- 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. 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 15 points for the core networking part.

- At this point, you are ready to write the IRC server. You should be able to use the code for the simple server as a base. 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.

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

Make your code modular and extensible where possible. This project project will be the basis for future projects. In the next project, you will be implementing a routing protocol on an overlay network of IRC servers. Try to keep this in mind.

“Be liberal in what you accept, and conservative in what you send.” Following this guiding principle of Internet design will help ensure your server works with many different and unexpected client behaviors.

Code quality is important. 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.

Start early! Although this is generally a good idea, it 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.

---

12http://csapp.cs.cmu.edu
15http://www.developerweb.net/forum/viewforum.php?f=53
17http://www.opengroup.org/onlinepubs/007908799/
7 Extra Credit

Our intent in suggesting extra credit items is to give interested students the opportunity to explore additional topics that we do not have time to cover in class. The primary reward for working on the suggested items is the additional experience and knowledge that they give you, not extra credit points. Extra credit will be granted at the discretion of the teaching staff.

For each suggestion, we list a rough estimate of the number of points you can receive. If you have more specific expectations about the extra credit you will receive, you should consult your TAs beforehand to avoid any disappointment.

If you work on the suggested topics below, please include in your project submission a file called extra.txt, describing what you have done. To receive credit for the programming items listed below, the code should be incorporated in your server. If it doesn’t work, don’t submit it!

Test case, 3 points In general, your test code will be evaluated in the robustness part (see evaluation section). But you can get 3 points if your test code captures an interesting error case and is adopted for project grading.

Denial of Service, 6 points (3 points for implementation, 3 points for evaluation.) Section 8.10 of the RFC provides some suggestions for dealing with malicious clients that attempt to make the IRC service useless by flooding the network. Implement the suggestion, and experimentally evaluate its effectiveness. Report your experimental findings. Propose any other solutions to the problem that you think of.

Async name lookups, 3 points Your IRC server must perform DNS lookups on clients’ addresses. As described in Section 8.11 of the RFC, the standard library calls may block for an extended time while the lookup times out. This is obviously an undesirable behavior for a server. Implement asynchronous name lookups using a separate thread to perform name lookups without blocking the primary server thread.

Scalability, 3 points Section 9 of the RFC notes that one of the current problems with the IRC protocol is scalability. Suggest how the protocol might be changed to improve scalability. You may get some ideas from looking at the design of IP multicast. (Or maybe not!)

Your Own Idea We welcome your suggestions for other interesting extensions to the project.