CS 455/555: Programming Project 1

Chat Server with Sockets

150 points

Due date on Canvas

1 Setup

Before starting on the project, please make sure you have accepted the team assignment via GitHub. The details are in the Canvas assignment named "Team Repository Setup."

2 Introduction

In this project, we will build a simplified chat server that can support multiple clients over the Internet. There are a gazillion chat programs out there using various protocols. One such popular protocol is IRC (Internet Relay Chat). We will implement a simplified version of this protocol in this project.

3 Protocol Design

3.1 IRC Protocol

Here is a very simplified version of the IRC protocol, which is a text-based protocol.

command	description
/connect <server-name>	Connect to named server
/nick <nickname>	Pick a nickname (should be unique among active users)
/list	List channels and number of users
/join <channel>	Join a channel, all text typed is sent to all users on the channel
/leave [<channel>]	Leave the current (or named) channel
/quit	Leave chat and disconnect from server
/help	Print out help message
/stats	Ask server for some stats

The full IRC protocol is fully described in RFC 1459. RFCs (Request For Comments) are the official documents of Internet specifications, communications protocols, procedures, and events. If you are curious, check out the RFC 1459 in the link below:

http://tools.ietf.org/html/rfc1459.html

3.2 Object-based Protocol

A client can accept the IRC commands but all communication with the server is using objects. Design an object-based protocol that gives the same functionality as the subset of IRC given earlier. Note that you will not only have objects that deliver commands but you will also need objects that deliver events.

4 Chat Server

The name of the main chat server class must be *ChatServer*, It should accept the following command line arguments:

```
java ChatServer -p <port#> -d <debug-level>
```

where debug level of 0 is normal (only error messages are reported) and at level 1 all events are reported. When testing in the lab, choose a port number in the range assigned to your team.

It is recommended to develop the server in the following stages. Note that you won't necessarily write the code for each version but thinking in stages will make it easier to design the multi-channel, multithreaded server that we want to develop.

- **Single Channel, Single-threaded Server**: First, focus on implementing a server that is single-threaded and only supports a single channel.
- **Multiple Channel, Single-threaded Server**: Once you have a single-threaded single-channel server working, then add multiple channels.
- **Multiple Channel, Multi-threaded Server** Add multiple threads (for example, one thread per channel or one thread per client).
- Limit the number of threads to four or use a thread pool to limit denial-of-service attacks.

Notes:

- The chat server should shut itself down if it has been idle for more than three minutes.
- Add a *shutdown hook*, so if the user uses Ctrl-c on the server, it shuts down gracefully. For example, it can send a message to all users on all channels before shutting down. It can also prints its stats before shutting down.

5 Chat Client

Develop a simple text-based chat client to demonstrate the functionality of your server. The client should support the subset of IRC commands shown earlier. The name of the main client class must be *ChatClient*. It takes no command line arguments (as we can specify server name and port number via the connect command).

6 Extra Credit

- Develop a Web-based Client. You would want to use JSON objects in this case instead of Java objects.

7 Documentation

- Create a MPEG video (ten minutes or less) that demonstrates various features and scenarios for the chat server and clients. The video must have participation from all team members and must have an audio track (no silent movies, please!). For recording the video, you can use Zoom or Panopto (the university has site licenses for both via your Boise State account).

- The README.md file must have at least the following elements:
  - Project number and title, team members, course number and title, semester and year.
  - A file/folder manifest to guide reading through the code.
  - A section on building and running the server/clients.
  - A section on how you tested it.
  - A web link to the demo video.
  - A section on observations/reflection on your development process and the roles of each team members.

8 Required Files

- Manage your project using your team git repository for the course. The project folder (named `p1`) has already been created for you in the repository.
- There must be a file named `README.md` in the project folder with contents as described earlier in the Documentation section.
- Make sure to have professional javadoc comments. No need to generate the javadocs though as I can do that myself.
- The project must have a `Makefile`. You can use any build system underneath (like gradle, maven, ant etc) but the Makefile should trigger it to generate the project.
- All the source code, build files et al.

9 Submitting the Project

You should be using Git throughtout the project with commits along the way. When you are eay to submit the project, open up a terminal or console and navigate to the team projects repository for the class. Navigate to the subdirectory that contains the files for the project.

- Clean up your directory and remove any auto-generated files such as .class files
  ```
  make clean
  ```
- Add and commit your changes to Git (this would be specific to your project)
  ```
  git add README.md (other files and folders that needed to be added)
  ```
- Commit your changes to Git
  ```
  git commit -a -m "Finished project p1"
  ```
- Create a new branch for you code
  ```
  git branch p1_branch
  ```
- Switch to working with this new branch
  ```
  git checkout p1_branch
  ```
  (You can do both steps in one command with `git checkout -b p1_branch`)

- Push your files to the GitHub server
  `git push origin p1_branch`
- Switch back to the master branch
  `git checkout master`
- Here is an example workflow for submitting your project. Replace p1 with the appropriate project name (if needed) in the example below.

```
[amit@localhost p1(master) ]$ git checkout -b p1_branch
Switched to a new branch 'p1_branch'

[amit@localhost p1(p1_branch) ]$ git push origin p1_branch
Total 0 (delta 0), reused 0 (delta 0)
To git@nullptr.boisestate.edu:amit
 * [new branch]      p1_branch -> p1_branch

[amit@localhost p1(p1_branch) ]$ git checkout master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
```

- Help! I want to submit my code again! No problem just checkout the branch and hack away!

```
[amit@localhost p1(master) ]$ git branch -r
  origin/HEAD -> origin/master
  origin/master
  origin/p1_branch

[amit@localhost p1(master) ]$ git checkout p1_branch
Branch p1_branch set up to track remote branch p1_branch from origin.
Switched to a new branch 'p1_branch'

[amit@localhost p1(p1_branch) ]$ touch newfile.txt
[amit@localhost p1(p1_branch) ]$ git add newfile.txt

[amit@localhost p1(p1_branch) ]$ git commit -am "Adding change to branch"
[p1_branch 1e32709] Adding change to branch
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 newfile.txt

[amit@localhost p1(p1_branch)↑ ]$ git push origin p1_branch
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 286 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@nullptr.boisestate.edu:amit
   36139d1..1e32709  p1_branch -> p1_branch

[amit@localhost amit(p1_branch) ]$ git checkout master
```

```
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.
[amit@localhost amit(master) ]$
```

- We highly recommend reading the section on branches from the Git book here: Git Branches in a Nutshell