LogosNet

Computer Science 367

Read all of the instructions. Late work will not be accepted.

Overview

In our final network programming assignment, you will implement the server and two distinct types of client for a simple online chat application. The first type of client is an *observer*. As was the case in Program 2, observer clients receive and display a stream of messages from the server in real-time; they do not send any data. Again like Program 2, the second type of client is a *participant*. Participant's primary job is to send messages. In a more sophisticated program, both of these clients would be integrated into a single display. To simplify the implementation of Program 3, we treat these separately. The server should be able to support up to 255 participants, and an equal number of observers. The following shows an example output of an observer client.

```
A new observer has joined
User Herbert has joined
User Agnus has joined
>    Agnus: hi
>  Herbert: hey
User Grover has joined
>   Grover: hello!
>    Agnus: hi
>  Herbert: bye
User Herbert has left
>   Grover: where'd Herbert go?
>    Agnus: no idea
User Herbert has joined
>  Herbert: I'm back!
>   Grover: welcome back
>  Herbert: thanks
User Grover has left
User Angus has left
>  Herbert: hello?
```

In this program, a range of events can happen at arbitrary times, and your program must be able to respond to them immediately. At any given time, a new participant or observer could join, a participant or observer could quit, or any of the participants could send a message. To accommodate this, you must use `select(2)` (or `poll(2)`) in your implementation of Program 3. `select(2)` is a system call that allows a program to monitor several socket (or file) descriptors, and return as soon as any of them are ready to read (or write, or report an error). Letting `select` babysit your socket descriptors permits you to react immediately to the aforementioned events. `poll(2)` is a similar system call with roughly equivalent functionality.

You and your partner are responsible for implementing (in the C programming language) both the client and the server for this game, using sockets. It will be developed in either your or your partner's git version control repository. You will work together using *pair programming*, described in the next section.

Pair Programming[1]

Pair programming is a software development technique where two programmers work together in front of one keyboard. One partner types code while the other is suggesting and/or reviewing every line of code as it is being typed. The person typing is called the driver. The person reviewing and/or suggesting code is called the navigator. Note that the navigator need only lead the development (e.g. "now we need an if-statement to check if x is greater than zero...") and **does not** need to "speak" code (e.g. "open curly brace enter tab i assigned 0 semicolon enter..."). The two programmers should switch roles frequently (e.g. every 20 minutes). For this to be a successful technique the team needs to start with a good program design so they are on the same page when it is finally time to start typing on the computer. **No designing or programming is to be done without both partners present!** Pair programming has been shown to increase productivity in industry and may well increase yours, but there are additional reasons it is being used in this class. First, it is a means to increase collaboration, which is something department graduates now working in industry report that they wish they had more experience with. Second, working in pairs is a good teaching tool. Inevitably, in each pairing the partners will have different styles and abilities (for instance, one person may be better at seeing the big picture while the other is better at finding detailed bugs or one person might like to code on paper first while the other likes to type it in and try it out). Because of that you will have to learn to adjust to another person's style and ideally you will meet each other half way when there are differences in approach. It's important that each person completely understands the program and so both parties need to be assertive. Be sure to explain your ideas carefully and ask questions when you are confused. Also it is crucial that you be patient! There is plenty of time allotted to complete this assignment as long as you proceed at a steady pace. Ask for help from me or the department tutors if you need it.

Program 3 Specifications

Your server and clients must be compliant with all of the following specifications in order to be considered correct. Non-compliance will result in penalties.

File and Directory Naming Requirements

Pick one of the two partners to host `LogosNet` in their git repository. **You will both need to submit its URL to Canvas**. This way I can keep track of your progress The following instructions apply to the partner hosting `LogosNet`. Spacing, spelling and capitalization matter.

- name your git repository as `logosnet`. Example of resulting URL: https://gitlab.cs.wwu.edu/tsikerm/logosnet

[1]These guidelines are based on a previous version developed by Perry Fizzano.

- All files should be on the root directory of the git repository.
- Your client source code should be contained in a file named `prog3_participant.c` and `prog3_observer.c`. If you create a corresponding header file you must name it `prog3_participant.h` and `prog3_observer.h`.
- Your server source code should be contained in a file named `prog3_server.c`. If you create a corresponding header file you must name it `prog3_server.h`.
- If you wish to have a shared header used by both the client and server, name it `prog3.h`.
- Your writeup must be a plain-text file named `writeup.txt`.
- Your plan must be a plain-text file named `plan.txt`.

Command-Line Specification

The server should take exactly two command-line arguments:

1. The port on which your server will listen for participants (a `uint16_t`).
2. The port on which your server will listen for observers (a `uint16_t`).

An example command to start the server is:

```
./prog3_server 36799 46799
```

Either client should take exactly two command-line arguments:

1. The name or address of the server (e.g. `cf416-01.cs.wwu.edu` or a 32 bit integer in dotted-decimal notation)
2. The port on which the server is running, a 16-bit unsigned integer

An example command to run the participant client is:

```
./prog3_participant 127.0.0.1 36799
```

An example command to run the observer client is:

```
./prog3_observer 127.0.0.1 46799
```

Compilation

Your code should be able to compile cleanly with the following commands on CF 416 lab machines:

```
gcc -o prog3_server prog3_server.c
gcc -o prog3_observer prog3_observer.c
gcc -o prog3_participant prog3_participant.c
```

Protocol Specification

The protocol is summarized by the following high-level rules:

- When a participant connects, it must negotiate a unique username.
- When a client (participant or observer) types `/quit` the client must terminate and close any outstanding connections
- Once a participant has confirmed a unique username, it will become an "active participant" and may send messages up to 1000 characters at any time.
- When an observer connects, it must affiliate with one of the participants.
- Once an observer has affiliated with one of the participants, it will receive all public messages as well as private messages sent by or sent to that participant.
- The server must be able to support 255 participants and 255 observers concurrently. If either type of client attempts to connect when there are already 255 of that type, they will receive a message letting them know the server is full and will be disconnected.
- The server should respond immediately to all messages being sent to it (e.g. messages from active participants, connect attempts, negotiating usernames, etc.). It should do so by using the `select` call to monitor all sockets. *No blocking on recv, send or accept!*
- Participant and observer clients may connect and disconnect at any point. When a participant disconnects, any corresponding observer client must also be disconnected by the server. When an observer client disconnects, you *do not* need to disconnect the participant: he or she can "fly blind" if desired.

Protocol details:

- Connecting and disconnecting clients to the server
  - Immediately upon connecting a participant to the server, the server should check if the maximum number of participants has already been reached; if it has the server sends the character `'N'` and closes the socket. If the limit has not been reached, the server sends the character `'Y'`.
  - Immediately upon connecting an observer to the server, the server should check if the maximum number of observers has already been reached; if it has the server sends the character `'N'` and closes the socket. If the limit has not been reached, the server sends the character `'Y'`.
  - When the server detects that an active participant[2] has disconnected, it should send the message `User $username has left` to all observers, where `$username` is the username affiliated with the participant. To send this, the server should first send a `uint16_t` in network standard byte order[3] specifying the length of this string, and then send the string as a character array (no newlines or null terminator should be sent).
  - When the server detects that an observer has disconnected, no message is generated. If that observer was affiliated with an active participant, the server should allow a future observer to affiliate with the same active participant.
- Negotiating a unique username.

[2] See below for a definition of "active participant."

[3] See the htons and ntohs functions.

  - Upon receiving a `'Y'` indicating that the connection is valid, the participant client should prompt the user for a username of 1-10 characters.
  - The username must consist only of upper-case letters, lower-case letters, numbers and underscores. No other symbols are allowed, nor is whitespace.
  - The user has 10 seconds to enter a username; if no username is received by the server within 10 seconds, it should close the connection on the participant.
  - If the user enters a username that is not 1-10 characters long, the client should prompt the user to enter a new username. This process repeats until the client enters a username of length 1-10 or the 10 second timeout expires.
  - If the user does enter a username within the time limit, the username should be sent to the server as follows: first send a `uint8_t` containing the length of the username, followed by the username as a series of characters (do not include the null terminator).
  - If that username is not currently being used and meets the criteria for a valid username, the server will send a `'Y'` in response.
  - If the username is already taken, the server sends a `'T'`; upon receiving a `'T'`, the client's timer resets to 10 seconds (i.e. they have 10 seconds to try again).
  - If the username is invalid, the server sends a `'I'`; this does *not* reset the timer.
  - If a valid username is sent and accepted by the server, the string `User $username has joined` should be sent to all observers, where `$username` is replaced by the accepted username. To send, first send a `uint16_t` in network standard byte order indicating the length of the string, then send the string as a character array.
  - Once the username has been accepted, the participant is now an "active participant."
- Affiliating an observer with a participant
  - Upon receiving a `'Y'` indicating that the connection is valid, the observer client should prompt the user for a username of 1-10 characters. The username should match the one they selected for their participant and the participant must be active.
  - The user has 10 seconds to enter a username; if no username is received by the server within 10 seconds, it should close the connection on the observer.
  - If the user enters a username not in the length of 1-10 characters, the client should prompt the user to enter a new username. This process repeats until the client enters a username of length 1-10 or the 10 second timeout expires.
  - If the user does enter a username within the time limit, the username should be sent to the server as follows: first send a `uint8_t` containing the length of the username, followed by the username as a series of characters (do not include the null terminator).
  - If that username matches that of an active participant, and no observer has affiliated yet with that participant, the server will send a `'Y'` in response.
  - If an observer has already affiliated with that username, the server sends a `'T'`; upon receiving a `'T'`, the client's timer resets to 10 seconds (i.e. they have 10 seconds to try again).

  - If no activate participant has that username, the server sends a `'N'` and *disconnects the observer*.
  - If a valid username accepted by the server, the string `A new observer has joined` should be sent to all observers; first send a `uint16_t` in network standard byte order indicating the length of that string, followed by that string as a character array.
- Public and private messages rules for active participants
  - The active participant client should repeat an infinite loop of: a) prompt the user for a message ("`Enter message: `"), b) send the message to the server.
  - The format for sending messages is: first send a `uint16_t` in network standard byte order indicating the length of the message, followed by the character array containing the message itself.
  - No null terminators should be sent (even trailing ones).
  - The message may or may not end with a newline character.
  - The message length is capped at 1000. The participant should never sent a message longer than 1000 characters.
  - The message must contain at least one non-whitespace character.
  - If the participant intends to send the message to a particular user only, the message should begin with "`@username` ", where username is replaced by the actual username of the intended recipient. There should be no whitespace before the `@` and there must be at least one space character after the `username`.
- Public and private messages rules for the server
  - If the server ever receives a message whose stated length is greater than 1000, it should disconnect on that participant and any affiliated observer. Otherwise, it processes the message as described below.
  - Once a server has received the entire message from a participant, it should check to see if it begins with `@`. If so, it should treat it as a private message; otherwise, it should treat it as a public message.
  - For public messages, the server prepends 14 characters to the message received: the `>` symbol, followed by a variable number of spaces, followed by the username of the sender, followed by a colon and then another space. The variable number of spaces should be picked to be 11 minus the length of the username (making the entire string prepended length 14).
  - After prepending, the server should send this message to all observer clients. First it should send a `uint16_t` in network standard byte order conveying the length of this message (14 more than whatever it received), and then send the message as a character array.
  - For private messages, the server should check to see if the intended recipient is an active participant.
  - If the recipient is valid, the server prepends 14 characters to the message received: the `-` symbol, followed by a variable number of spaces, followed by the username of the sender, followed by a colon and then another space. The variable number

of spaces should be picked to be 11 minus the length of the username (making the entire string prepended length 14).

  - After prepending, the server should send this private message only to 1) observer affiliated with the recipient and 2) the observer affiliated with the sender. First it should send a `uint16_t` in network standard byte order conveying the length of this message (14 more than whatever it received), and then send the message as a character array.
  - If the recipient is invalid, the server should send the message `Warning: user $username doesn't exist...` to the observer affiliated with the sender, where `$username` is replaced by the intended recipient. This message should be sent as a `uint16_t` in network standard byte order containing the length of that warning, following by sending the warning as a character array.

- Public and private messages rules for observers
  - Upon receiving an entire message from the server, the observer should check whether the message ends with a newline character; if it doesn't, one should be appended to the end. This message should then be printed to standard out.

The Repository

- All code for this program will be developed under a git repository in WWU CS's GitLab (gitlab.cs.wwu.edu).
- As noted above, your work will be done in the root directory of your working copy. Therefore, exactly once at the beginning of working on Lexiguess, you will need to create the repository and files. If you do not add and commit your files, I cannot see them, and thus cannot give you any points for them.
- Each time you commit, ATHINA (my automated testing software) will run tests against your code in the root directory and send an updated grade to Canvas.
- You must actively use GIT during the development of your program. If you do not have at least 3-5 commits, points will be deducted.
- If you haven't used GIT before, checkout this interactive tutorial: https://try.github.io/levels/1/challenges/1 Look also for the GIT guide on Canvas. There are also many GUIs that you could also use (e.g., SourceTree, GitKraken).

Plan

Prior to the checkpoint, you and your partner will need to work together to construct a plan for the program, documented in a plaintext file named `plan.txt`. Using proper spelling[4] and grammar, the plan should include the following numbered sections:

1. A one paragraph summary of the program in your own words. What is being asked of you? What will you implement in this assignment?

[4]Hint: aspell -c plan.txt

2. Your thoughts (a few sentences) on what you anticipate being the most challenging aspect(s) of the assignment.
3. A list of at least three resources you plan to draw from if you get stuck on something.

Checkpoint

Shortly after the assignment is posted (deadline listed at the beginning of this document), you will have a checkpoint to make sure you are on track. To satisfy the checkpoint, you need to do the following:

- Identify the partner who will host the repository
- Create your repository.
- Create, add, commit and push the following files:
  - `prog3_observer.c`, `prog3_participant.c` and `prog3_server.c`
  - The corresponding `.h` header files, if you plan to use them
  - `writeup.txt` (can be empty for now)
  - `plan.txt` (the plan in the previous section)

Grading

Submitting your work

When the clock strikes 11:59 PM on the due date, a script will automatically pull the latest committed version of your assignment. **(Do not forget to add, commit and push the work you want submitted before the due date!)** The repository should have in it, at the least:

- `prog3_observer.c`, `prog3_participant.c` and `prog3_server.c`
- Your plan: `plan.txt` (from the checkpoint)
- Your write-up: `writeup.txt`
- Optionally, your header files (if used)

Your repository need not and **should not contain your compiled binaries / object files.** Upon checking out your files, I will compile all three of your programs, run them in a series of test cases, analyze your code, and read your documentation.

Points

By default, you will begin with 50 points. Issues with your code or submission will cause you to lose points, including (but not limited to) the following issues:

- Lose fewer points:
  - Issues with the checkpoint (e.g. no plan, misnamed files or directories)
  - Not including a writeup or providing a overly brief writeup
  - Minor code style issues (inconsistent formatting, etc.)
  - Files and/or directories that are misnamed
  - Insufficient versioning in git
- Lose a moderate to large amount of points:

- Not including one of the required source code (`.c`) files
- A server that blocks when there is an event to handle (e.g. a new message)
- Not using `select` or `poll`
- Server cannot handle 255 participant clients and 255 observer clients concurrently
- Code that does not compile
- Code that generates runtime errors
- A client or server that does not take the correct arguments
- A client or server that does not follow the specified protocol
- Major code style issues (incomprehensible naming schemes, poor organization, etc.)

Write-Up

You need to create, add and commit a *plaintext* document named `writeup.txt`. In it, you should include the following (numbered) sections.

1. Your name and your partner's name
2. Declare/discuss any aspects of your client or server code that are not working. What are your intuitions about why things are not working? What issues you already tried and ruled out? Given more time, what would you try next? Detailed answers here are critical to getting partial credit for malfunctioning programs.
3. In a few sentences, describe how you tested that your code was working.
4. What was the most challenging aspect of this assignment, and why?

Academic Honesty

To remind you: you must not share code with anyone other than the official class grader or your professor: you must not look at any one else's code or show anyone else your code. You cannot take, in part or in whole, any code from any outside source, including the Internet, nor can you post your code to it. If you need help from anyone, all involved parties *must* step away from the computer and *discuss* strategies and approaches - never code specifics. I am available for help during office hours. I am also available via email (do not wait until the last minute to email). If you participate in academic dishonesty, you will fail the course.