# Computer Networks I 367

LogosNet *

October 24, 2017

## 1 Overview

In our final network programming assignment, you will implement a server and client for a simple online chat application. The client should be able to send and receive messages in real-time. The server should be able to support up to 255 participants. The following shows an example output of an observer client.

```
$ python3 logosnet_client.py --ip 127.0.0.1 --port 8021
Enter username, max 10 chars: Michael
User Michael has joined
User Thanos has joined
> Thanos: Hello
> Michael: Hello, asl?
> Thanos: 99 M Moon
> Michael: Nope
User Thanos has left
> Michael: @Thanos r u there?
```

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 could join, a participant could quit, or any of the participants could send a message. To accommodate this, you must use `select` in your implementation of LogosNet.

You, working **individually**, are responsible for implementing (in the Python 3 programming language) the server and client for this program, using sockets. It must be developed in your git version control repository.

## 2 Specifications

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

*The assignment and guidelines are based on a previous version developed by Brian Hutchinson.

## 2.1 File and Directory Naming Requirements

Host **LogosNet** in your git repository. **You will need to submit its URL to Canvas**. This way I can keep track of your progress. Spacing, spelling and capitalization matter.

- name your git repository as `LogosNet`. Example of resulting URL: https://gitlab.cs.wwu.edu/tsikerm/logosnet
- All files should be on the root directory of the git repository.
- Client should be named `logosnet_client.py`
- Server should be named `logosnet_server.py`
- Do not forget to include any additional files such as `helper.py` or any other file you have created for the assignment
- Your writeup must be a plain-text file named `writeup.txt`.

## 2.2 Command-Line Specification

The following are the command line parameter specifications for your server program. You must use the module `argparse` for this.

```
python3 logosnet_server.py -h
usage: logosnet_server.py [-h] --port p --ip i

LogosNet Server, the server version for the primitive networked chat program

optional arguments:
  -h, --help  show this help message and exit
  --port p    port number
  --ip i      IP address for client
```

Server mode requires the following arguments:

```
python3 logosnet\_server.py --ip 127.0.0.1 --port 8021
```

Notice that IP is the loopback address (meaning this computer). An alternative is to use a host name, e.g., localhost or 0.0.0.0 (meaning all interfaces).

Client mode requires the following arguments:

```
python3 logosnet_client.py -h
usage: logosnet_client.py [-h] --port p --ip i

LogosNet Client, the server client for the primitive networked chat program

optional arguments:
```

```
  -h, --help  show this help message and exit
  --port p    port number
  --ip i      IP address for client
```

Example:

```
python3 logosnet_client.py --ip 127.0.0.1 --port 8021
```

## 2.3 Communication Protocol

TCP guarantees byte-order but no whole delivery of a packet when `recv` is called. That means that multiple e.g., `socket.recv(2)` (receive up to 2 bytes at a time) may have to be called in order to receive the complete object that was sent by a sender. Implement your code by using `socket.recv(2)`. This is an extreme case but this way you will know that your "buffering" code works as expected. We will establish a communication protocol so that we will never receive incomplete objects. All communication between sender and receiver will have the following format:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|datalen|                           data...                       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

where datalen is a 4-bytes (32-bit integer) describing the length of the data that will be submitted in bytes. This needs to be in big-endian (network byte order).

To implement this protocol, write two new functions `recv(connection)` and `send(connection,message)` that you will use and they will internally take care of communicating our protocol. Since we are using `select` to not block, it is unwise and you **should not** use `socket.MSG_WAITALL` in this case. Instead, you need to make `recv` buffer in a global dictionary all incoming bytes for each separate client (connection) and if the length is larger or equal to the expected bytes then you can read the buffered global variable for a connection. Auxiliary variables to facilitate expected length and or state of header or data may be necessary. This approach guarantees that `select` transfers us to a ready read socket, our `recv` picks whatever is available and bounces back to `select` unless we have received what we have expected in full. As with the previous assignment, you will need to control (`try..except`) for what happens when `recv` times out or spams `b''`. Additionally explore `struct.pack()`, `struct.unpack()` for translating python objects to exact byte representation of known types, `json` for packing and unpacking python objects into string and in turn byte streams. Hint: Note that single byte streams are not dependent on endianess (e.g., ASCII or UTF-8). Refer to encode and decode functions for strings.

## 2.4 Protocol Specification

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

- When a participant connects, it must negotiate a unique username.
- 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.
- 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 clients may connect and disconnect at any point.
- You will have to invent convient protocol codes for many commands send between server and client. These can be readable or can be in coded format (e.g., HTTP codes).

Protocol details:

- Connecting and disconnecting clients to the server
  1. 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 a "server is full" (e.g., `['full']`) code and closes the socket. If the limit has not been reached, the server sends an "accept" code.
  2. When the server detects that an active participant has disconnected, it should send the message User username has left to all clients, where username is the username affiliated with the participant (if any otherwise `Anonymous`).
- Negotiating a unique username.
  1. Upon receiving an "accept" indicating that the connection is valid, the participant client should prompt the user for a username of 10 or fewer characters. Prompt should be: `Enter username, max 10 chars:`
  2. The username must not have any whitespace.
  3. The user has 60 seconds to enter a username; if no username is received by the server within 60 seconds, it should close the connection on the participant.

4. If the user enters a username longer than 10 characters, the client should prompt the user to enter a new username and reset the 60 second timer. This process repeats until the client enters a username of length ¡= 10 or the 60 second timeout expires.
5. If that username is not currently being used and meets the criteria for a valid username, the server will send a "username-accepted" code in response.
6. If the username is already taken, the server sends a "username-alreadyinuse"; upon receiving a "username-alreadyinuse", the timer resets to 60 seconds (i.e. they have one minute to try again).
7. If the username is invalid, the server sends a new request for username and timer is reset.
8. If a valid username is sent and accepted by the server, the string`User <username> has joined` to all clients, where ¡username¿ is replaced by the accepted username.
9. Once the username has been accepted, the participant is now an "active participant."

- Public and private messages rules for active participants

1. The active participant client should repeat an infinite loop of: a) prompt the user for a message ("`> <username>:` "), b) send the message to the server.
2. When the server detects that an active participant has disconnected, it should send the message User username has left to all clients, where username is the username affiliated with the participant (if any otherwise `Anonymous`).
3. The message length is capped at 1000. If a participant sends a message longer than 1000 it should never been broadcasted by the server.
4. 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

1. 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.
2. For public messages, the server prepends 14 characters to the message received: the > symbol, followed by a space, the username of the sender, followed by a colon and then another space. See the introduction section.

3. For private messages, the server should check to see if the intended recipient is an active participant.

- Whenever a user types `exit()` after the username has already been established the client needs to close the socket and exit.

# 3 The Repository

- All code for this program will be developed under a git repository in WWU CS's GitLab (gitlab.cs.wwu.edu). Checkout requirements on section 2.1.
- As noted above, your work will be done in the root directory of your working copy. 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 repository files (/logosnet_server.py, /logosnet_client.py) found 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)

# 4 Plan

Prior to the checkpoint, you will need to construct a plan for the program, documented in a plaintext file named plan.txt. Using proper spelling [1] and grammar, the plan should include the following numbered sections:

- A one paragraph summary of the program in your own words. What is being asked of you? What will you implement in this assignment?
- Your thoughts (a few sentences) on what you anticipate being the most challenging aspect of the assignment.
- A list of at least three resources you plan to draw from if you get stuck on something.

[1] Hint: aspell -c plan.txt

# 5 Checkpoint

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

- Create your repository
- Create, add and commit the following files:
    1. logosnet_server.py (can be empty for now)
    2. logosnet_client.py (can be empty for now)
    3. writeup.txt (can be empty for now)
    4. plan.txt (the finished plan in the previous section)
    5. any additional files required to execute your server and client

# 6 Grading

## 6.1 Submitting your work

When the clock strikes 10 PM on the due date you need to have your work *committed* to your remote repository (gitlab or github). The repository should have in it:

- logosnet_client.py, logosnet_server.py
- plan.txt (from the checkpoint)
- writeup.txt
- any other files necessary for executing server and client

## 6.2 Points

Issues with your code or submission will cause you to lose points. These include (but are not limited to) the following:

- Lose fewer points:
    1. Issues with the checkpoint (e.g. no plan, misnamed files or directories). Issues with naming will need to fixed or assignment will not be graded.
    2. Not including a writeup or providing a overly brief writeup
    3. Poor code style (inconsistent formatting, incomprehensible naming schemes, etc.)
    4. Files that are misnamed

5. Insufficient versioning in GIT (missing commits)

- Lose a moderate to large amount of points:
  1. Not including logosnet_client.py or logosnet_server.py
  2. Not using `select`
  3. Using `MSG_WAITALL`
  4. Server cannot handle 255 participant clients and 255 observer clients concurrently
  5. A client or server that does not take the correct arguments
  6. A client or server that does not follow the specified protocol
  7. Code that generates runtime errors

# 7 Write-Up

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

- Your name
- 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.
- In a few sentences, describe how you tested that your code was working.
- What was the most challenging aspect of this assignment, and why?

# 8 Academic Honesty

To remind you: you must not share code with anyone other than and your professor: you must not look at any one elses 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 any other groups, 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.