EE450 Socket Programming Project, Fall 2025
Due Date: Thursday, Dec 4th, 2025, 11:59 PM (Midnight)
(The deadline is the same for all on-campus and DEN off-campus students)
Hard Deadline (Strictly enforced)

OBJECTIVE:

The objective of this assignment is to familiarize you with UNIX socket programming. **It is an individual assignment, and no collaborations are allowed. Any cheating will result in an automatic F in the course (not just in the assignment).**
If you have any doubts/questions, post your questions on Piazza. You must discuss all project-related issues on Piazza. We will give extra points to those who actively help others out by answering questions on Piazza.

Important Notice: Before proceeding, you must read and understand the **Academic Integrity** and **AI Policy** below. Adherence to this policy is mandatory.

Academic Integrity

All students are expected to write all their code on their own!!!

Do not post your code on Github, especially in a public repository before the deadline!!!
Double check the setting and do some testing before posting in a private repository!!!

Copying code from friends or from any unauthorized resources (webpages, github, etc.) is called plagiarism not collaboration and will result in an F for the entire course. **Any libraries or pieces of code that you use or refer and you did not write must be listed in your README file.** Students are only allowed to use the code from Beej's socket programming tutorial. Copying the code from any other resources may be considered as plagiarism. Please be careful!!! All programs will be compared with automated tools to detect similarities; examples of code copying will get an F for the course. **IF YOU HAVE ANY QUESTIONS ABOUT WHAT IS OR ISN'T ALLOWED ABOUT PLAGIARISM, TALK TO THE TA. "I didn't know" is not an excuse.**

AI Policy

We allow the use of AI to assist you in understanding how to write code, utilize libraries, and similar tasks. However, you are not permitted to copy the AI-generated content directly. It can only be used as a reference. Additionally, you must include comments in the code specifying the prompt used and the AI model involved.

You should never attempt to present or include content created by others, including generative AI as your own. Attempting to take credit for content generated by AI or others without proper acknowledgement is a violation of USC's policies and standards for academic integrity and can result in disciplinary action. For any other details not explicitly mentioned, please refer to the USC AI Policy webpage: https://libguides.usc.edu/generative-AI/scholarship-research

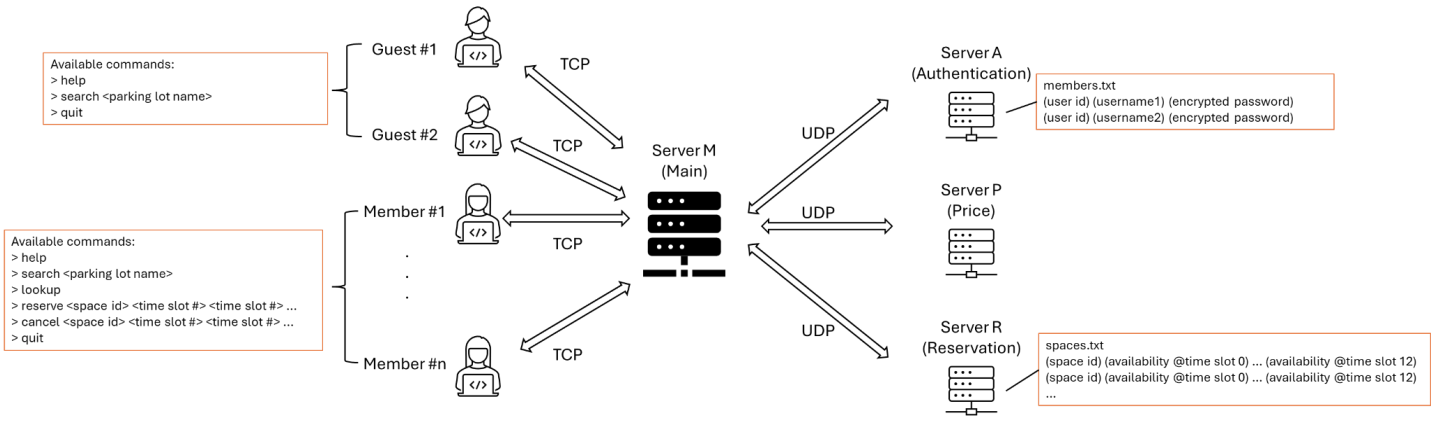
PROBLEM STATEMENT:

Parking reservation on campus presents significant challenges requiring efficient resource allocation, real-time availability tracking, and dynamic pricing strategies. This project implements a distributed parking reservation system that addresses these challenges through a multi-server architecture. The system manages parking spots across two USC campuses: UPC and HSC. This system will feature basic user authentication, real-time availability checking, reservation processing, and dynamic pricing based on multiple factors including zone location, time of day, and duration.

In this project, we need three backend servers: Server A, Server R, and Server P, one main server: Server M, and client interfaces. Users will be able to perform various operations through the client interface, and these operations will be forwarded to the corresponding backend servers via the main server. Here are the functional descriptions of each server and client.

- Client: This is the only interface for user interaction. There are two types of users: guests and members. Through the client terminal, every guest can perform actions listed below:
  - Check availability
  - Close the connection (Log out)

  Every member can perform actions listed below:
  - Log in
  - See their reservations
  - Check availability
  - Reserve a parking space
  - Cancel their reservations
  - Close the connection (Log out)
- Server M (Main): Handles all member actions by dispatching requests to the appropriate backend servers. Additionally, **all communications between the three backend servers must go through Server M**.
- Server A (Authentication): Verifies users' usernames and encrypted passwords and returns whether they are valid. It uses "members.txt" as its database.
- Server R (Reservation): Records whether parking spaces in each lot are available for reservation. It uses "spaces.txt" as its database.
- Server P (Price Calculator): Calculates the total cost of a parking reservation. The rate varies based on the time of day (peak vs. off-peak hours) and the length of the booking.



Source Code Files

Your implementation should include the source code files described below, for each component of the system:

- **Client:** The name of this piece of code must be `**client.c**` or `**client.cpp**` (all in lowercase). The header file (if you have one; it is not mandatory) must be called `**client.h**` (all in lowercase).

- **serverM (Main Server)**: You must name your code file: `**serverM.c**` or `**serverM.cpp**` (all lowercase except for the `M`). If you have a header file, it must be named `**serverM.h**` (all lowercase except for the `M`).

- **Backend Servers A, P, R:** You are required to create three distinct files, choosing from the following naming conventions: server#.c or server#.cpp. The filename must utilize one of these formats, substituting "#" with the specific server identifier (either "A" or "P" or "R") to reflect the server it represents, resulting in filenames like serverA.c, serverA.cpp, serverP.c, or serverP.cpp (note that the name should be entirely in lowercase except for the letter replacing "#"). If available, you should also include a corresponding header file named server#.h, adhering to the same naming rule for the "#" replacement. This ensures a clear, organized naming structure for your code and its associated header file, if any.
Note: You are not allowed to use one executable for all four servers (i.e. a "fork" based implementation).

- **Optional files:** You may also include additional files that you write yourself with common functions, using .c, .cpp, or .h extensions.

Input Files:

- members.txt: Located in Server A (Authentication server) which maintains the user credentials. It is a headerless, space-delimited text file with each line containing a User ID, username, and **encrypted** password in that specific order. Each user's data is stored on a separate, new line. For instance, the line

12 james aaa111

represents a user with the ID "12," the username "james," and the encrypted password "aaa111." For the encryption method, please refer to "authentication" in the Detailed Explanation section.

- spaces.txt: Structured to list availability of each parking space. This is a headerless, space-delimited text file where each line contains a space ID and 12 values for the day's 12 time slots. Each value indicates either availability (0) or a user ID. For example, the line:

  U666 0 0 0 0 12 0 5 5 5 5 0 12

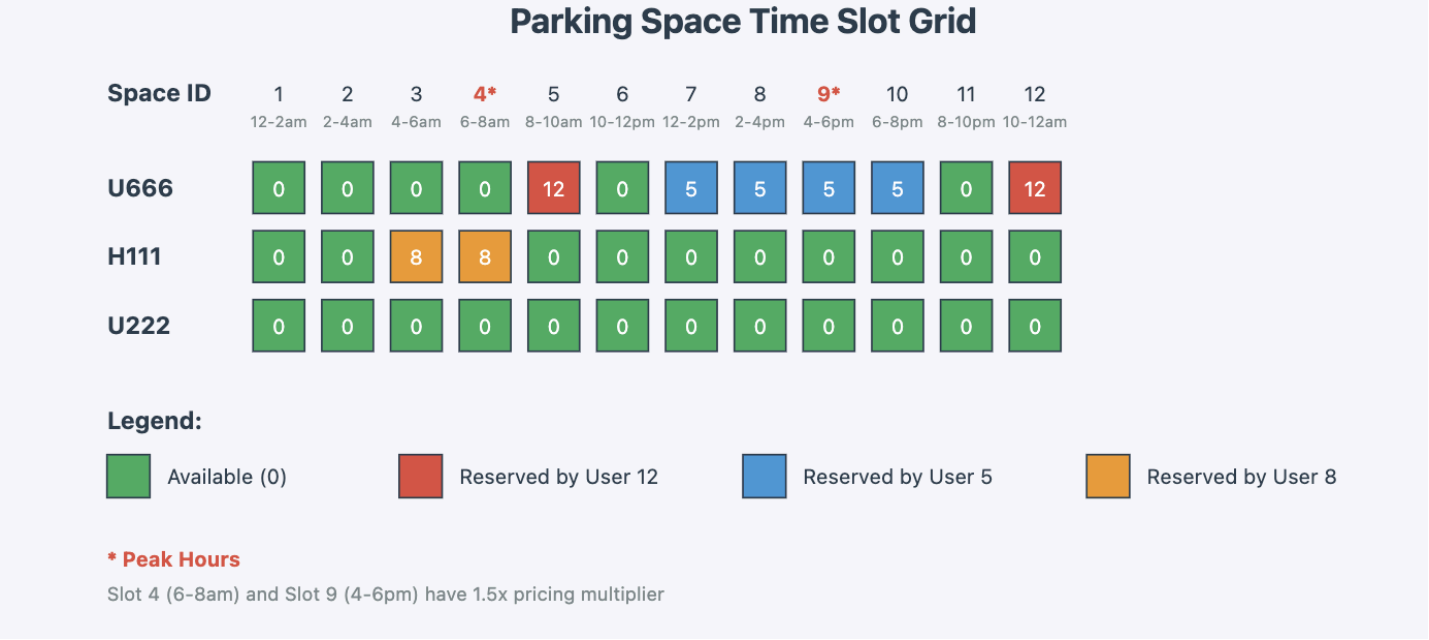
  indicates that for space U666, user ID 12 has reserved time slots 5 and 12, and user ID 5 has reserved time slots 7 through 10.

  The first letter of a space's ID indicates its parking lot: 'U' stands for the UPC lot and 'H' stands for the HSC lot (e.g., space U111 is in the UPC lot).

Parking Lot Name	Starts from
UPC	U
HSC	H

Constraints:

- Space ID is a letter followed by three digits, e.g. U111, H222.
- 1 <= number of spaces in each parking lot <= 100
- User IDs are integers. 1 <= User ID <= 9999



DETAILED EXPLANATION

Phase 1: Bootup

Please refer to the following order to start your programs: server M, server A, server R, server P, and then multiple Clients. Your programs must start in this order. Each of the servers and the clients have boot-up messages that must be printed on the screen. Please refer to the on-screen messages section for further information.

When three backend servers (server A, server R, and server P) are up and running,each backend server should read the corresponding input file (Server A reads members.txt, Server R reads spaces.txt, and Server P doesn't read any file) and store the information in a certain data structure. You can choose any data structure that accommodates your needs. The communication between the backend servers and the main server (server M) should be via UDP over the port mentioned in the PORT NUMBER ALLOCATION section. In the following phases, you have to make sure that the correct backend server is being contacted by the main server for corresponding requests. You should print correct on-screen messages onto the screen for the main server and the backend servers, indicating the success of these operations as described in the “ON-SCREEN MESSAGES" section.

After the servers are booted and the required information is stored on the backend servers, at least two clients will connect concurrently via TCP. Once the clients boot up and the initial boot-up messages are printed, the clients wait for the system to check the authentication and then enter the parking reservation system.

Please check on-screen messages for the on-screen messages of different events on each server and client side.

Phase 2A: Login and confirmation

Authentication:

command format: **./client <username> <unencrypted password>**

In this phase, the client will be asked to enter the username and their unencrypted password on the terminal. In order to be authenticated, the user will run the command as described at the beginning of this section.

There are two types of users for this system:

- Guest: This type of user will need to write down “guest” for username and “123456” for password. After providing this information, they will have access to check availability operations, as explained in the problem description.
- Member: After providing credentials to the main server, it will request information to server A (Authentication server). Server A has the members.txt and will check if there is an instance in this file that has the <username> and <password> provided at the beginning of this phase. If a line which

includes the given user and password is found then we consider that the authentication has been completed and the user can proceed with future operation.

If the credentials provided do not fall under any of the previous cases, server A, server M and client must provide an error message (detailed in ON SCREEN messages section) and the process must end. If a client wants to start over they need to run the command again.

For members, this system will provide an encryption scheme for the authentication of their passwords. The scheme would be as follows:

- Offset each character and/or digit by 3.
- character: cyclically alphabetic (A-Z, a-z) update for overflow
- digit: cyclically 0-9 update for overflow
- The scheme is case-sensitive.
- Special characters will not be encrypted or changed.

A few examples of encryption are given below:

Example	Original Text	Encrypted Text
#1	WelcometoEE450!	ZhofrphwrHH783!
#2	199xyz@$	422abc@$
#3	0.27#&	3.50#&

Constraints:

- The password will be case-sensitive (1~50 chars).

The password information indicated on the members.txt corresponds to the encrypted version of the passwords. For members, when asked for credentials they are required to provide their username and original password (non-encrypted). A short list of usernames and original passwords will be provided as an additional file for reference ("members_unencrypted.txt").

Phase 2B: Help

Help function:

command format: help

This command displays the list of available commands and their formats. The output differs depending on whether the client is logged in as a guest or a member. The corresponding on-screen messages are defined in the On-Screen Messages section.

Here, we will encounter two scenarios:

- For Guest clients, the help function will only display the commands available to guests, which are checking availability and logging out.
- For Member clients, the help function will display the complete set of commands available to members.

Phase 3:

Check availability:

Command format: **search <parking lot name>**

This command retrieves a list of available parking spaces and their corresponding free time intervals for the specified parking lot. The request is routed from the client through the Main Server to the Reservation Server.

If the user provides a parking lot name, the output will include the available spaces and time slots for that specific lot. If the user only types “search” without specifying a lot name, the system will return availability for all parking spaces across both parking lots. Guest clients can use “search <parking lot name>” to view availability but cannot reserve any spaces. Member clients, on the other hand, can both check availability and proceed with a reservation if desired.
Note: Please refer to the 'Input Files' section for a list of valid parking lot names.

Make Reservation:

Command format: **reserve <parking space code> <timeslot> <timeslot>...**

The command must be followed by a parking space code and at least one timeslot; otherwise, an error message will be displayed. This operation allows a member client to reserve one or more specific time slots for a given parking space. The request is routed from the client through the Main Server to the Reservation Server, where availability is verified and the reservation is recorded.
Here, we will encounter two scenarios:

- Scenario 1: One or more requested slots are unavailable.
  If the client attempts to reserve a slot that is already taken, the system will return an on-screen message specifying which slot(s) are unavailable.
    - If there are still remaining available slots in the request, the client will be prompted to enter Y or N (case-insensitive) to decide whether to reserve the remaining slots.
        - If Y is entered, the Reservation Server will confirm the reservation for the remaining slots and return a success message.
        - If N is entered, the reservation will be cancelled and a failure message will be displayed.

    - If there are no remaining available slots, the system will immediately return a failure message indicating that the reservation failed, without prompting for Y/N.
- Scenario 2: All requested slots are available.
  If all the requested slots are free, the Reservation Server will confirm the reservation and return an on-screen message indicating that the reservation was successful.

Note: Please also review the “Pricing” section when writing this part.

Lookup Reservations:

Command format: **lookup**

This command allows members to look up their own reservations. The request is sent from the client to the Main Server, which forwards it to the Reservation Server. The Reservation Server retrieves all records associated with the authenticated client and returns them for display. This operation is available only to members. Guest clients do not have permission to look up reservations, as they cannot make reservations. The system will display the corresponding on-screen message, as defined in the On-Screen Messages section, indicating either the client’s reservations or that no reservations were found.

Note: The output format can be found in the onscreen message section.

Cancel Reservation:

Command format: **cancel <parking space code> <timeslot> <timeslot>...**

This command allows a member client to cancel one or more reserved time slots in a specific parking space. The request is sent from the client to the Main Server and then forwarded to the Reservation Server, where the reservation records and slot availability are updated accordingly.
Here, we will encounter two scenarios:

- If any of the specified time slots are not reserved by this member, a failure on-screen message will be shown.
- If all specified time slots are currently reserved by this member, the system will cancel the reservation, update the slot availability, and display the corresponding success on-screen message.

Note: Please also review the “Pricing” section when writing this part.

Pricing

Automatic Pricing (applies to reserve and cancel):

Parking prices are determined by a combination of base rates, reduced rates after the first two hours, and peak-hour multipliers. Peak hours are defined as **8:00–10:00 AM** and **4:00–6:00 PM**, during which a **1.5×** multiplier is applied to the base rate.

- UPC Parking Lot: The base rate is $10 per hour for the first two hours, and $7 per hour for each additional hour beyond two hours. Peak-hour charges are subject to the 1.5× multiplier.
- HSC Parking Lot: The base rate is $15 per hour for the first two hours, and $10 per hour for each additional hour beyond two hours. Peak-hour charges are subject to the 1.5× multiplier.

Cancelation:

- UPC Parking Lot: Refund $7 per hour.
- HSC Parking Lot: Refund $10 per hour

The Pricing Server applies these rules automatically whenever a reservation or cancellation is processed. All computed costs and refunds are returned with two-decimal precision. Parking duration is calculated on a cumulative daily basis and does not need to be consecutive. All calculations begin at midnight and are processed according to the sequence of time slots. In other words, the price is the same whether you book time slot #1 before a peak hour slot, or book a peak hour slot before time slot #1. **Parking time is accumulated separately for the different parking lots.**

Whenever a client issues a reserve or cancel command, the Main Server first coordinates with the Reservation Server to verify the request. For reservations, once the requested slots are confirmed, the Main Server forwards the reservation details for that specific member to the Pricing Server. The Pricing Server applies the lot's pricing policy and peak-hour multipliers to compute the final cost. This result is returned to the Main Server, which then sends a confirmation message to the client along with the calculated price. Please note that the calculated price represents the member's total current parking cost, not just the cost of this single reservation operation.

For cancellations, the process is similar. After verifying the cancellation request, the Main Server forwards the details to the Pricing Server, which computes the refund amount based on the lot's pricing rules. The Main Server then returns a success message to the client showing the canceled slots and the corresponding refund value.

Here are some examples of the calculated price.

All of the following commands are executed sequentially on a single member, and all reservation and cancellation operations complete successfully.

Command	Output price	Calculation
reserve U666 1	Total cost: $20.00	$2 \times 10 = 20$
reserve U777 1 2	Total cost: $48.00	$20 + 4 \times 7 = 48$
reserve H666 7	Total cost: $78.00	$48 + 2 \times 15 = 78$
reserve H777 9	Total cost: $108.00	$78 + (2 \times 10) \times 1.5 = 108$
cancel H666 7	refund: $20.00	$2 \times 10 = 20$

Note: reserve H666 7 is calculated as $48 + 2 \times 15$ instead of $48 + 2 \times 10$ because parking time is accumulated separately for different parking lots.

Close Connection (Log out):

Command format: **quit**

This command allows the client to close the connection with the Main Server and end the session. Upon receiving this command, the Main Server will terminate the TCP connection with the client and display the corresponding on-screen message.

PORT NUMBER ALLOCATION

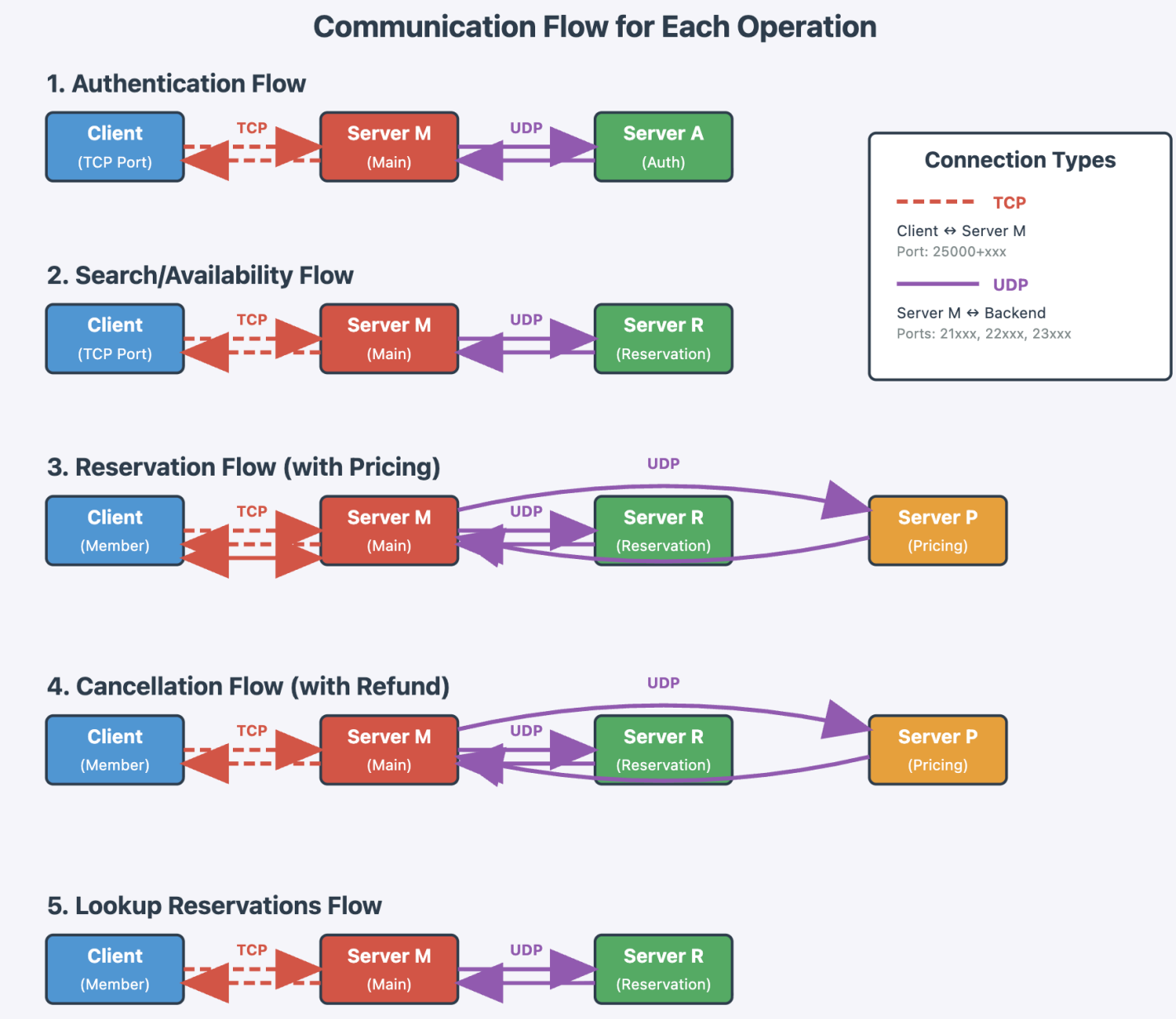
Port Number Allocation

Server	UDP Port	TCP Port
Server M	24000 + xxx	25000 + xxx
Server A	21000 + xxx	—
Server R	22000 + xxx	—
Server P	23000 + xxx	—
Client	—	Dynamic

Note: xxx = last 3 digits of your USC ID

Example: If USC ID ends in 456, Server M's UDP port = 24456

Communication Flow:



ON-SCREEN MESSAGES

*Note: The order of the on-screen messages shown below in the tables does not carry any specific meaning.

Table 1. Server A (Authentication Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting)	[Server A] Booting up using UDP on port <port number>.
Upon Receiving the auth request	[Server A] Received username <USERNAME> and password ******.
If the auth request is member user	[Server A] Member <USERNAME> has been authenticated.
If either username or password is incorrect	[Server A] The username <USERNAME> or password ****** is incorrect.
If the auth request is guest user	[Server A] Guest has been authenticated.

Table 2. Server R (Reservation Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting)	Server R is up and running using UDP on port <port number>.
Availability request	
Upon receiving availability request from the main server	Server R received an availability request from the main server.
After returning the availability information	Server R finished sending the response to the main server.
Reservation Request	
Upon receiving a reservation request from the main server	Server R received a reservation request from the main server.
After checking availability and all slots are available	All requested time slots are available.
After checking availability and some slots are unavailable	Time slot(s) <slot numbers> not available for <space code>. Requesting to reserve rest available slots (Y/N).
Upon receiving a Y response to the partial reservation confirmation	User confirmed partial reservation.

Upon receiving a N response to the partial reservation confirmation	Reservation cancelled.
After successfully making reservation	Successfully reserved <space code> at time slot(s) <slot numbers> for <username>.
Lookup Request	
Upon receiving a lookup request from the main server	Server R received a lookup request from the main server.
After returning the user's reservations	Server R finished sending the reservation information to the main server.
Cancellation Request	
Upon receiving a cancellation request from the main server	Server R received a cancellation request from the main server.
After successfully cancelling reservation	Successfully cancelled reservation for <space code> at time slot(s) <slot numbers> for <username>.
If cancellation fails	No reservation found for <username> at <space code> time slot(s) <slot numbers>.

Table 3. Server P (Pricing Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting)	Server P is up and running using UDP on port <port number>.
Pricing Request	
Upon receiving a pricing request from the main server	Server P received a pricing request from the main server.
Upon calculating reservation cost	Calculated total price of $<amount> for <username>.
After sending the price to the main server	Server P finished sending the price to the main server.
Refund Request	
Upon receiving a refund calculation request from the main server	Server P received a refund request from the main server.

Upon calculating refund amount	Calculated refund of $<amount> for <username>.
After sending the refund amount to the main server	Server P finished sending the refund amount to the main server.

Table 4. Server M (Main Server) on-screen messages

Event	On Screen Message
Booting Up (Only while starting)	[Server M] Booting up using UDP on port <port number>.
Authentication	
Upon receiving the authentication request from the client	Server M received username <USERNAME> and password ******.
Upon sending the authentication request to server A	Server M sent the authentication request to Server A.
Upon receiving the authentication response from server A	Server M received the response from Server A using UDP over port <port number>.
Upon sending the authentication response to the client	Server M sent the response to the client using TCP over port <port number>.
Availability Request	
Upon receiving an availability request from a guest	Server M received an availability request from Guest for <parking lot> using TCP over port <port number>.
Upon receiving an availability request from a member	Server M received an availability request from <username> for <parking lot> using TCP over port <port number>.
After forwarding the availability request to server R	Server M sent the availability request to Server R.
After receiving the response from server R	Server M received the response from Server R using UDP over port <port number>.
After forwarding the response to the client	Server M sent the availability information to the client.
Reservation Request	

Upon receiving a reservation request from the Member User	Server M received a reservation request from <username> using TCP over port <port number>.
After forwarding the reservation request to server R	Server M sent the reservation request to Server R.
After receiving partial availability from server R	Server M received the response from Server R using UDP over port <port number>.
After forwarding the partial reservation confirmation to the client	Server M sent the partial reservation confirmation request to the client.
After forwarding the confirmation response to Server R	Server M sent the confirmation response to Server R.
After forwarding reservation details to server P	Server M sent the pricing request to Server P.
After receiving the price from server P	Server M received the pricing information from Server P using UDP over port <port number>.
After forwarding the result to the client	Server M sent the reservation result to the client.
Lookup Request	
Upon receiving a lookup request from member	Server M received a lookup request from <username> using TCP over port <port number>.
After forwarding the lookup request to server R	Server M sent the lookup request to Server R.
After receiving the response from server R	Server M received the response from Server R using UDP over port <port number>.
After forwarding the response to the client	Server M sent the lookup result to the client.
Cancellation Request	
Upon receiving a cancellation request from member	Server M received a cancellation request from <username> using TCP over port <port number>.
After forwarding the cancellation request to server R	Server M sent the cancellation request to Server R.

After receiving the response from server R	Server M received the response from Server R using UDP over port <port number>.
After forwarding to server P for refund calculation	Server M sent the refund request to Server P.
After receiving the refund amount from server P	Server M received the refund information from Server P using UDP over port <port number>.

Table 5. Client on-screen messages

Event	On Screen Message
Booting Up	Client is up and running.
Authentication Phase	
Upon sending auth request	<username> sent an authentication request to the main server.
Enter the correct credentials (member)	<username> received the authentication result. Authentication successful.
Enter the correct credentials (guest)	You have been granted guest access.
Enter wrong credentials	Authentication failed: username or password is incorrect.
Help	
Asking for help commands (member)	Please enter the command: <search <parking lot>>, <reserve <space> <timeslots>>, <lookup>, <cancel <space> <timeslots>>, <quit>
Asking for help commands (guest)	Please enter the command: <search <parking lot>>, <quit>
Search Request	
Upon sending a search request (member)	<username> sent an availability request to the main server.
Upon sending a search request (guest)	Guest sent an availability request to the main server.
After receiving the response (spaces available)	The client received the response from the main server using TCP over port <port number>.

	<space1>: Time slot(s) <index of available slots separated by a space> <space2>: Time slot(s) <index of available slots separated by a space> ... ---Start a new request---
After receiving the response (no spaces available)	The client received the response from the main server using TCP over port <port number>. No available spaces in <parking lot name>. ---Start a new request---
After receiving the response (invalid lot)	The client received the response from the main server using TCP over port <port number>. Invalid parking lot name. ---Start a new request---
Reservation Request	
No space code or timeslots specified	Error: Space code and timeslot(s) are required. Please specify a space code and at least one timeslot.
Upon sending a reservation request	<username> sent a reservation request to the main server.
Partial availability - asking for confirmation	Time slot(s) <unavailable slots> not available. Do you want to reserve the remaining slots? (Y/N):
After successful reservation	The client received the response from the main server using TCP over port <port number>. Reservation successful for <space> at time slot(s) <slots>. Total cost: $<amount> ---Start a new request---
After failed reservation	The client received the response from the main server using TCP over port <port number>. Reservation failed. No slots were reserved. ---Start a new request---
Lookup Request	
Upon sending a lookup request	<username> sent a lookup request to the main server.
After receiving reservations	The client received the response from the main server using TCP over port <port number>. Your reservations: <space1>: Time slot(s) <index of reserved slots> <space2>: Time slot(s) <index of reserved slots> ... ---Start a new request---

After receiving no reservations	The client received the response from the main server using TCP over port <port number>. You have no current reservations. ---Start a new request---
Cancellation Request	
No space code or timeslots specified	Error: Space code and timeslot(s) are required. Please specify what to cancel.
Upon sending a cancellation request	<username> sent a cancellation request to the main server.
After successful cancellation	The client received the response from the main server using TCP over port <port number>. Cancellation successful for <space> at time slot(s) <index of slots>. Refund amount: $<amount> ---Start a new request---
After failed cancellation	The client received the response from the main server using TCP over port <port number>. Cancellation failed: You do not have reservations for the specified slots. ---Start a new request---
Invalid Commands	
Guest attempting member-only command	Guests can only check availability. Please log in as a member for full access. ---Start a new request---

Assumptions

1. You have to start the processes in this order: serverM, serverA, serverR, serverP, and at least two clients. If you need to have more code files than the ones that are mentioned here, please use meaningful names and all small letters and mention them all in your README file.
2. You are allowed to use blocks of code from Beej's socket programming tutorial (Beej's guide to network programming) in your project. However, you need to cite the copied part in your code. Any signs of academic dishonesty will be taken very seriously.

3. When you run your code, if you get the message port already in use or "address already in use," please first check to see if you have a zombie process. If you do not have such zombie processes or if you still get this message after terminating all zombie processes, try changing the static UDP or TCP port number corresponding to this error message (all port numbers below 1024 are reserved and must not be used). If you have to change the port number, please do mention it in your README file and provide reasons for it.
4. You do not need to consider the racing conditions for parking reservations. Assume that when two or more clients are connected, any given operation completes **fully** before another client begins. In other words, we do not consider cases where a parking space availability might change mid-reservation due to another client's action.

Requirements

1. Do not hardcode the TCP or UDP port numbers that are to be obtained dynamically. Refer to Table Port Number Allocation to see which ports are statically defined and which ones are dynamically assigned. Use `getsockname()` function to retrieve the locally-bound port number wherever ports are assigned dynamically as shown below:

```
1. /* Retrieve the locally-bound name of the specified socket and store it in the sockaddr structure */
2. getsock_check = getsockname(TCP_Connect_Sock, (struct sockaddr*)&my_addr, (socklen_t
   *)&addrlen);
3. // Error checking
4. if (getsock_check == -1) {
5.     perror("getsockname");
6.     exit(1);
}
```

2. The host name must be hard coded as "localhost" or "127.0.0.1" in all codes.
3. Your client, backend servers, and main server should remain running and continue waiting for additional requests until the TAs terminate the servers and clients using Ctrl+C or terminate the clients with the "quit" command. If any of them terminate prematurely, you will lose points as a result.
4. All the naming conventions and the on-screen messages must conform to the previously mentioned rules.
5. You are not allowed to pass any parameters or values or strings or characters as a command-line argument except what is already described in the project document.
6. All the on-screen messages must conform exactly to the project description. You should not add any more on-screen messages. If you need to do so for debugging purposes, you must comment out all of the extra messages before you submit your project.

7. Please do remember to close the socket and tear down the connection once you are done using that socket.

Programming Platform and Environment

1. All your submitted code MUST work well on the provided virtual machine Ubuntu. No additional package installation is allowed.

2. All submissions will only be graded on the provided Ubuntu. TAs/Graders won't make any updates or changes to the virtual machine. It's your responsibility to make sure your code works well on the provided Ubuntu. "It works well on my machine" is not an excuse.

3. Your submission MUST have a Makefile. Please follow the requirements in the following "Submission Rules" section.
4. environment setup**:** **https://github.com/csci104/docker/**

Programming Languages and Compilers

You must use only C/C++ on UNIX as well as UNIX Socket programming commands and functions. Here are the pointers for Beej's Guide to C Programming and Network Programming (socket programming):
- http://www.beej.us/guide/bgnet/
  (If you are new to socket programming, please study this tutorial carefully as soon as possible and before starting the project)
- http://www.beej.us/guide/bgc/

You can use a UNIX text editor like `emacs` to type your code and then use compilers such as `g++` (for C++) and `gcc` (for C) that are already installed on Ubuntu to compile your code.
You must use the following commands and switches to compile your `.c` or `.cpp` files. It will make an executable by the name of "yourfileoutput":

gcc -o yourfileoutput yourfile.c
g++ -o yourfileoutput yourfile.cpp

Do NOT forget the mandatory naming conventions mentioned before!

Also, inside your code, you need to include these header files in addition to any other header files you think you may need:
#include <stdio.h>
#include <stdlib.h>

```
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/wait.h>
```

Submission File and Folder Structure:

Your submission must have the following folder structure and the files (the examples are of .cpp, but it can be .c files as well):

- ee450_lastname_firstname.tar.gz
    - ee450_lastname_firstname
        - client.cpp
        - serverM.cpp
        - serverA.cpp
        - serverR.cpp
        - serverP.cpp
        - Makefile
        - readme.txt (or) readme.md
        - <Any additional header or implementation files YOU WROTE YOURSELF>

Please make sure that your name matches the one in the class list. The TAs will extract the tar.gz file, and will place all the input data files in the same directory as your source files. The executable files should also be generated in the same directory as your source files.
Do NOT include any input data files or executable files, as this will result in losing points.

Submission Rules

Along with your code files, include a README file and a Makefile. The Makefile requirements are mentioned in the section below.
The README file can be in any format (Markdown or txt). The only requirement is that the TAs should be able to open the file and read it in the studentVM (the VM your project will be graded in) without installing any additional software.

In the README file, please include:

a. Your Full Name as given in the class list
b. Your Student ID
c. What you have done in the assignment
d. What your code files are and what each one of them does. (Please do not repeat the project description; just name your code files and briefly mention what they do).
e. The format of all the messages exchanged, e.g., username and password are concatenated and delimited by a comma, etc.
f. Any idiosyncrasy of your project. It should specify under what conditions the project fails, if any.
g. Reused Code: Did you use code from anywhere for your project? If not, say so. If so, state what functions and where they're from. (Also identify this with a comment in the source code).
h. Which version of Ubuntu (only the Ubuntu versions that we provided to you) are you using?

Reusing functions that are directly obtained from a source that does not belong to you (e.g., from the internet or generated by AI) with little to no modifications, is considered plagiarism—except for code from Beej's Guide. Whenever you are referring to an online resource, make sure to only look at the source, understand it, close it, and then write the code by yourself. The TAs will perform plagiarism checks on your code, so make sure to follow this step rigorously for every piece of code you submit.

Submissions WITHOUT README and Makefile WILL NOT BE GRADED.

Makefile

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html

About the Makefile: makefile should support following functions:

make all	Compiles all your files and creates executables
make clean	Removes all the executable files

After "make all', TAs will run the following commands in sequence:

./serverM	start Main server
./serverA	start Backend server A
./serverR	start Backend server R
./serverP	start Backend server P
./client	start the client

TAs will first compile all codes using **make all**. They will then open at least 6 different terminal windows. On 4 terminals they will start servers M, A, R and P. On the remaining terminals, they will start the clients using "./client". Remember that all programs should always be on once started. TAs will check the outputs for multiple values of input. The terminals should display the messages shown in On-screen Messages tables in this project writeup.

Here are the instructions:
On your VM, go to the directory which has all your project files. Remove all executable and other unnecessary files. **ONLY** include the required source code files, Makefile and the README file. Now run the following two commands:
>> tar cvf ee450_YourLastName_YourFirstName.tar *
>> gzip ee450_YourLastName_YourFirstName.tar
Now, you will find a file named
"ee450_YourLastName_YourFirstName.tar.gz" in the same directory.
Please notice there is a space and a star(*) at the end of the first command.

An example submission would be:
First Name: John
Last Name: Doe
So the submission would be: ee450_Doe_John.tar.gz
Any compressed format other than .tar.gz will NOT be graded!

Upload “ee450_YourLastName_YourFirstName.tar.gz” to the Digital Dropbox on the BrightSpace website (BrightSpace -> EE450 -> Activities -> Assignments -> Project). After the file is uploaded to the dropbox, you must click on the “Submit” button to actually submit it. If you do not click on “Submit”, the file will not be submitted.
BrightSpace will keep only the last submission of your submissions. Submission after the deadline is considered as invalid. BrightSpace will send you a “Dropbox submission receipt” to confirm your submission. So please do check your emails to make sure your submission is successfully received. If you have not received a confirmation mail, contact your TA if it always fails. After receiving the confirmation email, please confirm your submission by downloading and compiling it on your machine. This is exactly what your designated TA would do, So please grade your own project from the perspective of the TA. If the outcome is not what you expected, try to resubmit and confirm again. We will only grade what you submitted even though it’s corrupted.
Please take into account all kinds of possible technical issues and do expect a huge traffic on the BrightSpace website very close to the deadline which may render your submission or even access to BrightSpace unsuccessful.
Please DO NOT wait till the last 5 minutes to upload and submit because some technical issues might happen and you will miss the deadline. And a kind suggestion, if you still get some bugs one hour before the deadline, please make a submission first to make sure you will get some points for your hard work!

There is absolutely zero tolerance for late submissions! Do NOT assume that there will be a late submission penalty or a grace period. If you submit your project late (no matter for what reason or excuse or even technical issues), you simply receive a ZERO for the project.

Grading Criteria

Notice: We will only grade what is already done by the program instead of what will be done. The grading criteria are subject to change.

Your project grade will depend on the following:

1. Correct functionality, i.e. how well your programs fulfill the requirements of the assignment, specially the communications through UDP and TCP sockets.
2. Inline comments in your code. This is important as this will help in understanding what you have done.
3. Whether your programs work as you say they would in the README file.
4. Whether your programs print out the appropriate error messages and results.
5. Your code will only be tested on a fresh copy of the provided Virtual Machine (https://github.com/csci104/docker/). If your programs are not compiled or executed on these VM, you will receive only minimum points as described below. Be careful if you are going to use other environments!!! Do not update or upgrade the provided VM as well!!!
6. If your submitted codes do not even compile, you will receive 5 out of 100 for the project.
7. If your submitted codes compile using make but when executed, produce runtime errors without performing any tasks of the project, you will receive 10 out of 100.
8. The minimum points for compiled and executable codes is 15 out of 100.
9. If your code does not correctly assign the TCP or UDP port numbers (in any phase), you will lose points each.
10. We will use similar test cases to test all the programs. These test cases cover all situations including edge cases, referring to the on-screen messages section.
11. There are no points for the effort or the time you spend working on the project or reading the tutorial. If you spend about 2 weeks on this project and it doesn't even compile, you will receive only 5 out of 100.
12. You must discuss all project related issues on the Piazza Discussion Forum. We will give extra points to those who actively help others out by answering questions on Piazza. (If you want to earn the extra credits, do remember to leave your names visible to instructors when answering questions on Piazza.)
13. Your code will not be altered in any way for grading purposes and however it will be tested with different inputs. Your TA/Grader runs your project as is, according to the project description and your README file and then checks whether it works correctly or not. If your README is not consistent with the description, we will follow the description.

Cautionary Words

1. Start on this project early!!!

2. In view of what is a recurring complaint near the end of a project, we want to make it clear that the target platform on which the project is supposed to run is Ubuntu 20.04. It is strongly recommended that students develop their code on this virtual machine. In case students wish to develop their programs on their personal machines, possibly running other operating systems, they are expected to deal with technical and incompatibility issues (on their own) to ensure that the final project compiles and runs on the requested virtual machine. If you do development on your own machine, please leave at least three days to make it work on Ubuntu. It might take much longer than you expect because of some incompatibility issues.

3. You may create zombie processes while testing your codes, please make sure you kill them every time you want to run your code. To see a list of all zombie processes, try this command:
   >>ps –aux | grep ee450
   Identify the zombie processes and their process number and kill them by typing at the command-line:
   >>kill -9 <process number>