Project 4 - Socket Programming

CSci 4061: Introduction to Operating Systems
Due Date: Friday, 6th December, 2019

1. Instructions

- You will extend the word counting application of programming assignment 2 by using server-client socket programming.
- You could complete this project in a group of up to two students.
- Each group should turn in **one** copy with the names of all group members on it.
- The code must be originally written by your group. No code from outside the course texts and slides may be used—your code cannot be copied or derived from the Web, from past offerings, other students, programmer friends, etc.
- All submissions must compile and run on any CSE Labs machine in KH 4-250.
- A zip file should be submitted through Canvas by 11:59pm on Friday, Dec 6th.
- Note: Do not publicize this assignment or your answer to the Internet, e.g., public GitHub repo. To share code between team members, create a private repo in github.umn.edu.

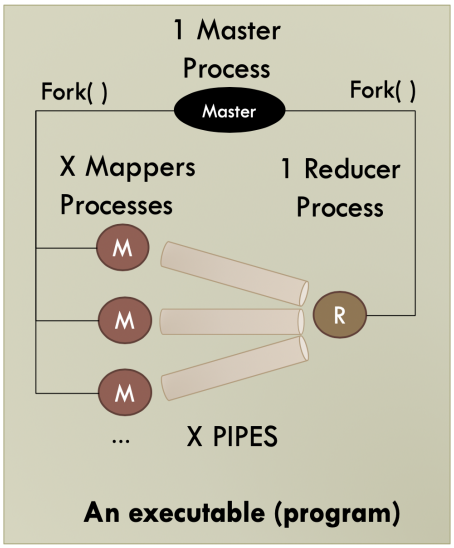
2. Background and Objective

Socket programming is a way of connecting two nodes (sockets) on a network to communicate with each other. One node (Server socket) listens on a particular IP address and port number, while other nodes (Clients sockets) reaches out to the server socket to build a connection. Through the programming assignment 4 (PA4), you will extend the word counting application of programming assignment 2 (PA2) using the socket programming. Similar to PA2, there will be a master process, multiple mapper processes, and a single reducer process. However, for communication, you will use socket-based TCP connections instead of pipes as shown in Figure 1 and Figure 2.

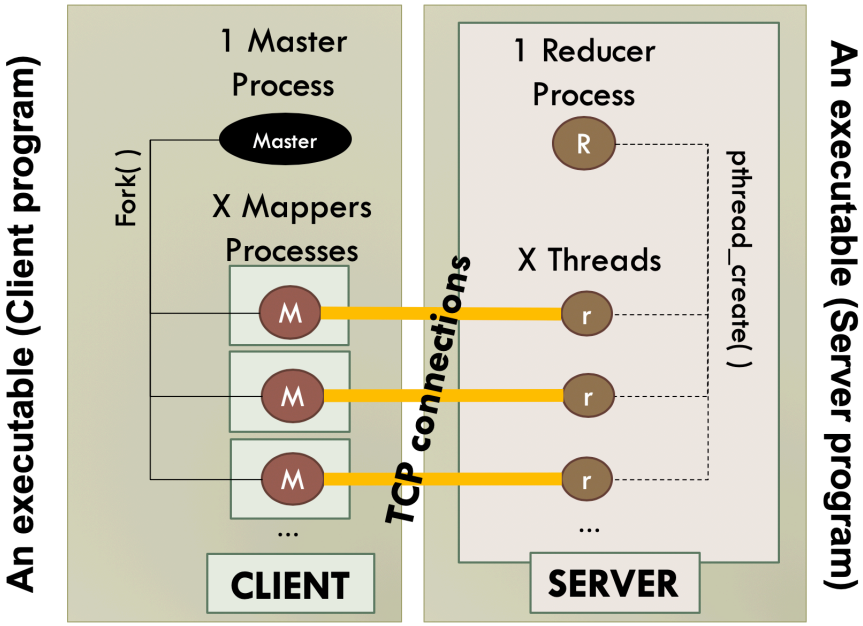
3. Project Overview

In PA4, you will make two executables (two separate independent programs), **server program** and **client program**. Master process is the main process of the client program. Master process spawns multiple mapper processes similar to PA2. Reducer process is the main process of the server program. Note that the reducer process is no longer spawned by the master process. The reducer process spawns a thread whenever it establishes a new TCP connection with a client. It is called **a multi-threaded server.** Server, Reducer and Reducer process are interchangeably used in this document.

The client program has two types of clients - **Master clients** and **Mapper clients**. Master process and mapper processes can be a client. Master client implementation is extra credit. Details of master client can be found in the section 5. Extra credit. Now, Let's fist focus on the mapper clients. Each mapper process initiates a TCP connection to the server. Once the connection is established, the mapper client sends **deterministic requests**, a fixed set of requests, to the server. Mappers, Mapper clients, and Master's child processes indicate the same thing. The relationship of client processes and server threads are shown in Figure 2.



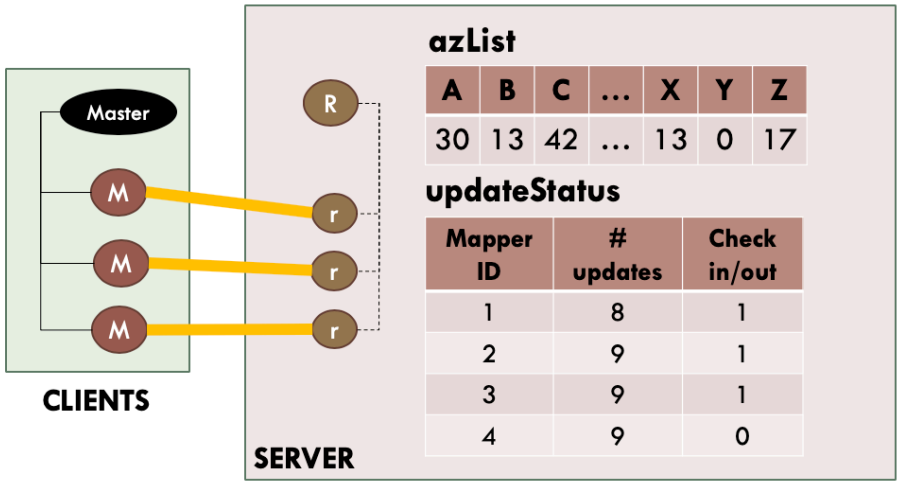
< Figure 1: Process relationship of **programming assignment 2** >



< Figure 2: Process and thread relationships of **programming assignment 4** >

3.1 The server program

The server program is a multi-threaded server. The server is responsible for listening and building connections from clients. The server waits for connections on the specified port. When it receives a connection, it should spawn a new thread to handle that connection. A thread is responsible for handling requests from a client by reading client's messages from a socket set up between them, doing necessary computation (refer to section 4. Communication Protocol) and sending responses to the client back through the socket. The thread should exit after it close the client connection.



< Figure 4: Information and data structure the server maintains >

Server maintains two important lists - ***azList*** and ***updateStatus***. azList is a 1-D integer list and updateStatus is a 2-D integer list, as shown in Figure 4. The server saves the sum of all the word count results from mapper clients in the *azList*. *updateStatus* table has 3 essential attributes - **MapperID, Number of updates, check in/out flag**. A new entry of the *updateStatus* table is created when the server receives a **CHECKIN** request from a new mapper client (refer to the details of request commands in section 4.1 Request). You can feel free to add more columns to the table if needed.

Table 1: Attributes of updateStatus Table

Attribute Name	E
MapperID	Unique mapper ID, which is greater than 0
Number of updates	Incremented by 1 when receiving an **UPDATE_AZLIST** request from a mapper client. The value should be the same with the number of files the mapper client processes.
Check In / out flag	1 – When a mapper client is checked in. 0 – When a mapper client is checked out.

3.2 The client program

Client program is responsible for initiating a connection with the server on the IP address and port number given to it. It consists of 3 phases.

[Phase 1] File Path Partitioning - By Master Client

This is exactly the same with phase 1 of PA2. You are given the codes (phase1.c and phase1.h) for phase1. This code will generate a folder “MapperInput” and “Mapper_<mapperID>.txt” so-called mapper files in the folder. The mapper files contain a list of file paths. MapperID starts from 1 and increments by 1. Please refer to the PA2 write-up for further details of phase 1.

[Phase 2] Deterministic Request Handling - By Mapper Clients

Master process assigns a unique mapperID to mapper processes while it spawns mapper processes. The mapper ID starts from 1, and increments by 1. Mapper processes run **in parallel**. Each mapper client sets up a TCP connection to the server, and sends the following a fixed set of requests **sequentially** in a deterministic manner. A TCP connection is used to send all the following requests. Mapper clients can access their own mapper file in MapperIntput folder.

1. CHECKIN
2. UPDATE_AZLIST (If the mapper client has multiple file paths in its mapper file, this request is sent to the server multiple times. For example, Mapper_1.txt contains 10 lines of file paths, the mapper client sends 10 UPDATE_AZLIST requests to the server with the word count result of each file.
3. GET_AZLIST
4. GET_MAPPER_UPDATES
5. GET_ALL_UPDATES
6. CHECKOUT

For mapper clients to send any requests to the server, they should be checked in first. After check-in, they can send the other types of requests to the server. After exchanging the messages with the server, they should check out, close the TCP connection, and exit. Master waits until all mapper processes are terminated. You can find details of requests in section 4.

[Phase 3] Dynamic Request Handling - By Master Clients

Phase 3 is extra credit. You will add the master client functionality. After the master process should make sure all mapper processes are terminated, it sends any requests dynamically by reading commands from a file named **"commands.txt"**. It uses separate TCP connections to send the requests. You can find the details in Section 5. Extra Credits.

4. Communication Protocol

Communication protocol is an application-layer protocol formed of requests and responses. **Both requests and responses are integer arrays**. Requests are sent from the client, received by the server. After the server does necessary computation, it responds to clients. You can find the details of the requests and responses in the section.

4.1 Request

The request structure is as follows. You can find the relevant definitions in the given protocol.h. You can use them in your implementation.

Table 2: Request Structure

Field Name	Size (# of Integer)	Purpose
Request Command	1	Specifies request command
MapperID	1	Specified mapperID (-1 for master client)
Data	26	Relevant data for the command. If there is no data, fill with zeros

Table 3: Request Command Codes

Request Command	Command Name	Data	Request permission (Who can send)
1	CHECKIN	Zeros	Mapper clients
2	UPDATE_AZLIST	Word count result of a file	Mapper clients
3	GET_AZLIST	Zeros	Mapper clients Master clients (extra credit)
4	GET_MAPPER_UPDATES	Zeros	Mapper clients
5	GET_ALL_UPDATES	Zeros	Mapper clients Master clients (extra credit)
6	CHECKOUT	Zeros	Mapper clients

Details of each request are as follows.

- **1. CHECKIN**
  - [Client] Mapper clients should send this request before sending other types of requests.
  - [Server] Server creates a new entry in *updateStatus* table for a new mapper client if corresponding entry does not exist in the table. If there is an existing entry in the table, the server simply changes the check in/out field to checked-in (1).

- **2. UPDATE_AZLIST**
  - [Client] Mapper clients send it to the server with **PER-FILE** word count results. If there are multiple file paths in the mapper file, this request should be sent as many as the same number of files paths. If there is no files in the mapper file, mapper clients SHOULD NOT send this message.
  - [Server] Server sums the word count results in the *azList*, and increases the number of update field of *updateStatus* table by 1 for the corresponding mapper client. The number of updates of a particular mapper client should be the same with the number of files paths in the mapper's mapper file.

- **3. GET_AZLIST**
  - [Client] These requests can be sent by both mapper clients and master clients (extra credit). If mapper clients want to send this request to the server, they should be already checked in. On the other hand, master clients can send it without check-in (extra credit)..
  - [Server] Server returns the current values of the *azList*

- **4. GET_MAPPER_UPDATES**
  - [Client] Only mapper clients can send this request. They should be already checked in.
  - [Server] Server returns the current value of "number of updates" field of *updateStatus* table for the corresponding mapper ID.

- **5. GET_ALL_UPDATES**
  - [Client] These requests can be sent by both mapper clients and master clients (extra credit). If mapper clients want to send this request to the server, they should be already checked in. On the other hand, master clients can send it without check-in (extra credit).
  - [Server] Server returns the sum of all values of “number of updates” field in the *updateStatus* table.

- **6. CHECKOUT**
  - [Client] This request is the last request sent from a mapper client. After getting a response, the mapper client closes its TCP connection and terminates its own process.
  - [Server] Server updates check in/out field of the*updateStatus* table to checked-out (0).

4.2 Response

The response structure is as follows. You can find the relevant definitions in the given protocol.h. You can use them in your implementation.

Table 3: Response Structure

Field Name	Size (# of Integer)	Purpose
Request code	1	Specifies type of request
Response code	1	Specified response code
Data	1 or 26	Relevant data from server

Table 4: Response Code

Response Code		Meaning	Purpose
0	RSP_OK	Success	For successful requests
1	RSP_NOK	Error	For unsuccessful requests

Table 5: Data Returned on Success

Request Command	Request Name	Data Returned
1	CHECKIN	1 value (mapperID)
2	UPDATE_AZLIST	1 value (mapperID)
3	GET_AZLIST	26 word count values (*azList* values) at that moment the request is received regardless of mapperID
4	GET_MAPPER_UPDATES	1 value (The value of the number of updates field of *updateStatus* table for the corresponding mapperID)

5	GET_ALL_UPDATES	1 value (Sum of all entries of the number of updates field in the *updateStatus* table)
6	CHECKOUT	1 value (mapperID)

Servers should handle various error cases such as:

- When receiving a request with unknown request code
- When mapper ID is not greater than zero
- When there is no corresponding entry in the *updateSatus* table
- When a mapper client sends CHECKIN request, if it is already checked-in.
- When a mapper client sends CHECKOUT request, if it is not checked-in.
- Handling request permission etc.

4.3 Log Printout

The server program prints the following logs in terminal. The client programs print the following logs in a log file **"log_client txt"** in the "log" folder. (refer to section 6. Folder Structure). Please stick closely to this output format. The following items are minimal requirements to print, so you can print additional logs or messages for failure cases. In addition, you can use a function createLogFile() to initialize a client log file in the client.c. You can find example logs in the PA4_Appendix document, and other expected outputs and logs in the "Testcases/ExpectedResult" folder.

- **When the server is ready to listen.**
  - [Client] None
  - [Server] Print `"server is listening\n"`.
- **Establish a TCP connection between a client and server.**
  - [Client] Print `"[%d] open connection\n",` mapperID.
  - [Server] Print `"open connection from %s:%d\n",` client_ip, client_port.
- **Close the TCP connection between a client and server.**
  - [Client] Print `"[%d] close connection\n",` mapperID.
  - [Server] Print `"close connection from %s:%d\n",` client_ip, client_port.
- **1. CHECKIN**
  - [Client] Print `"[%d] CHECKIN: %d %d\n",` mapperID, Response Code (response[1]), Data (response[2])
  - [Server] Print `"[%d] CHECKIN\n",` mapperID (request[1])
- **2. UPDATE_AZLIST**
  - [Client] Print `"[%d] UPDATE_AZLIST: %d\n",` mapperID, Total number of messages sent to server. **Print out this log after a mapper client sends all UPDATE_AZLIST requests to the server.**
  - [Server] None
- **3. GET_AZLIST**
  - [Client] Print `"[%d] GET_AZLIST: %d <26 numbers>\n",` mapperID, Response Code (response[1]), and all data received from the server (26 numbers) in the same line (1 space between numbers)
  - [Server] Print `"[%d] GET_AZLIST\n",` mapperID (request[1])

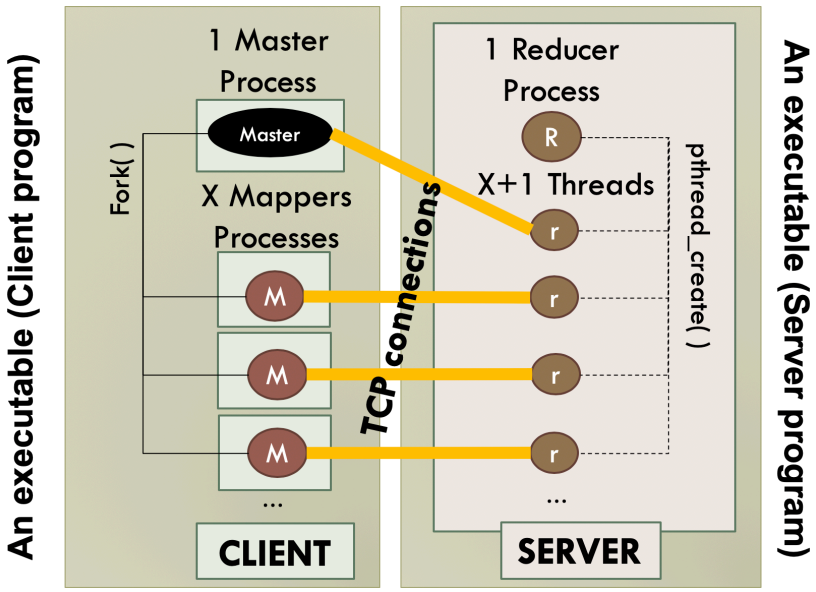
- **4. GET_MAPPER_UPDATES**
  - [Client] Print `"[%d] GET_MAPPER_UPDATES: %d %d\n",` mapperID, Response Code (response[1]), Data (response[2])
  - [Server] Print `"[%d] GET_MAPPER_UPDATES\n",` mapperID (request[1])
- **5. GET_ALL_UPDATES**
  - [Client] Print `"[%d] GET_ALL_UPDATES: %d %d\n",` mapperID, Response Code (response[1]), Data (response[2])
  - [Server] Print `"[%d] GET_ALL_UPDATES\n",` mapperID (request[1])
- **6. CHECKOUT**
  - [Client] Print `"[%d] CHECKOUT: %d %d\n",` mapperID, Response Code (response[1]), Data (response[2])
  - [Server] Print `"[%d] CHECKOUT\n",` mapperID (request[1])

5. Extra Credits

You can get extra credits by implementing the master client's dynamic request handling (Phase3 of client program). Unlike the deterministic request handling by mapper clients, the master client dynamically sends requests by reading request commands from a command file named **"commands.txt"**. The master client initiates a new TCP connection to the server after it makes sure all mapper processes are terminated. **When the master client sends a request, the mapperID field of the request should be set to -1.** Requests from master client does not require check-in and check-out unlike mapper clients, so master client should set up a new TCP connection for each request, and close after getting a response. For example, if you use the following commands.txt, server will get successful response only for the request commands 3 (GET_AZLIST) and 5 (GET_ALL_UPDATES). For other request, it will get unsuccessful responses from the server. **You can assume that 2 (UPDATE_AZLIST) doesn't appear in the commands.txt.**

```
1
3
4
5
6
7
```

< example commands.txt file >



< Figure 5: Extra Credit – Master Client >

6. Folder Structure

Please strictly conform with the folder structure. The conformance will be graded.
You should have two project folders named "**PA4_Client**" and "PA4_Server". "PA4_Client" should contain "**include**", "**src**", "**log**", "**Testcases**", and "**MapperInput**" folders. "PA4_Server" should contain **include**" and "**src**". You can feel free to modify the provided Makefiles.

- "**include**" folder: All .h header files
- "**src**" folder: All .c source files
- "**log**" folder: log_client.txt.
- "**Testcases**" folder: 5 TestCase folders and ExpectedResult folder are provided for your testing. Your program will be tested with additional hidden TestCases. THe TestCases are located in this folder. **Please DO NOT include this folder in the final deliverable.**
- "**MapperInput**" folder: This folder is created as part of phase1 of the client program. Please do not include this folder in the final deliverable.
- **Each** top folders ("**PA4_Client**" and "**PA4_Server**") should contain the following files.
    - Makefile
    - executable
    - commands.txt (only in PA4_Client if you attempt extra credit)

7. Execution Syntax

The usage of your server program is as follows. The executable name should be "**server**".

```
./server <Server Port>
```

- <Server Port> is any unsigned integer to be used as a port number

The usage of your client program is as follows. The client executable name should be "**client**".

```
./client <Folder Name> <# of Mappers> <Server IP> <Server Port>
```

- <Folder Name> the name of the root folder to be traversed.
- <# of Mappers> the number of mapper processes spawned by master.
- <Server IP> IP address of the server to connect to.
- <Server Port> port number of the server to connect to.

7. Assumptions and Hints

- You should use TCP sockets.
- Maximum number of mapper clients per a master client is 32.
- Maximum number of concurrent connections at the server side is 50.
- Mapper IDs should be greater than zero, and unique in a client program.
- A client connects to a single server at any time.
- A server considers multiple client requests at a time.
- The server program is not terminated unless it is killed by a user.
- Requests and response messages are integer arrays.
- Given phase1 code handles symbolic folders and files.
- Your server will get any types of requests including both successful and unsuccessful requests, so you need to handle various errors at the server side.

- Commands.txt file contains any integers (>0) except for 2.
- Start to work on a local host for both client and server programs, then try it on multiple machines later.

8. Submission

One student from each group should upload to Canvas, a zip file containing the two project folders ("**PA4_Client**" and "**PA4_Server**") and a **README.md** that includes the following details:

- Team names and x500
- How to compile the client and server programs
- How to run the client and server programs
- Your and your partner's individual contributions
- Any assumptions outside this document
- If you have attempted extra credit

The README.md file does not have to be long, but must properly describe the above points. Your source code should provide **appropriate comments** for functions. At the top of your README.md file and each C source file please include the following comments:

```
/*test machine: CSELAB_machine_name * date: mm/dd/yy
* name: full_name1 , [full_name2]
* x500: id_for_first_name , [id_for_second_name] */
```

9. Grading Policy

1. (5%) Correct README content
2. (5%) Appropriate code style and comments
3. (10%) Conformance check
    a. (5%) Folder structure and executable names
    b. (5%) Log format for client logs and server logs
4. (30%) TCP connection setup between clients and server
    a. (10%) Set up a TCP connection between mapper client and the server
    b. (10%) Use threads at the server side
    c. (10%) Set up multiple TCP connections between mappers and the server
5. (30%) Deterministic requests handling between Mapper clients and the server
    a. (15%) Mapper client side
    b. (15%) Server side
6. (10%) Error Handling
7. (10%) Successful executions of client and server programs in different machines.
8. (10%) Extra credit – Master client's dynamic request handling
    a. (5%) Set up a TCP connection between master client and the server
    b. (5%) Read request commands, and process send and receive the commands