# CSci 4061: Introduction to Operating Systems
# Spring 2021
# Project #4 - Socket Programming

Instructor: Abhishek Chandra

Due: 11:59 pm, May 3, 2021

## 1. Background

Network sockets, or simply "sockets", enable processes to communicate with each other. In many regards, sockets appear to process to be simply another file, but a file that permits a process to pass data to another process, usually on another machine. In some sense, sockets can be viewed as being similar to pipes, except the two communicating processes may be on different machines, connected by a network (such as the Internet). Like pipes, sockets have special operations for opening and closing a socket, and a few other operations. Typically, one process, the server, listens on a socket that has a particular IP address and port number. Another process, the client, uses its socket to establish a connection with the server's socket.

In this programming assignment, you will extend the "word length statistics" application covered in previous projects. There will be a server and multiple clients. Mappers are implemented as client processes and reducers as a single server process. Clients (mappers) read files, compute word length statistics for the files, and send the data to the server. The server (reducer) accumulates the word length statistics sent by clients. The server and clients will communicate via sockets-based TCP connections, rather than files or pipes.

## 2. Project Overview

In this programming assignment, you will make two executables (two separate independent programs), **server** and **client**. For server program, you will use multithreading to implement a multi-threaded server. For client program, you will generate multiple client processes. Each client process reads a file, processes "word length statistics" and sends a message to the server through TCP connections. The server spawns a thread whenever it establishes a new TCP connection with a client. The statistics sent by clients are accumulated in a shared resource named "Result Histogram". You can reuse your codes for computing word length statistics since the definition of a word for this assignment is the same with the one used in previous projects. Space and newline character are the only delimiters. For example, here is an example sentence: *Thi's is. a te_xt* 0h gr8!!!* The words in this sentence are {*Thi's*, is., a, *te_xt**, 0h, gr8!!!} and the length of the words are {5, 3, 1, 6, 2, 6}.

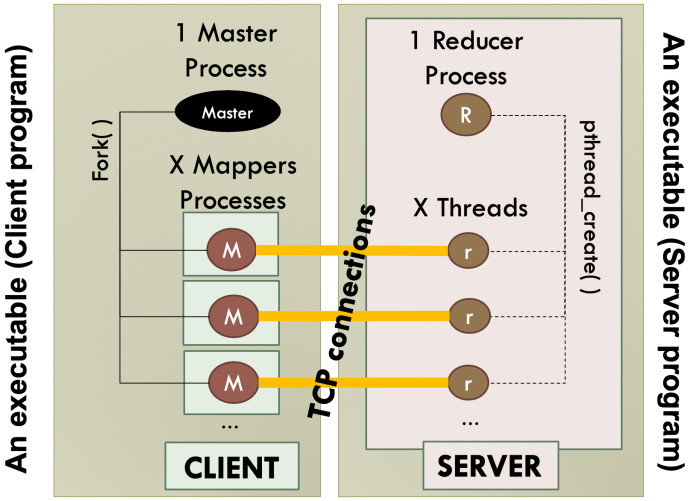
### 2.1. Messages

There are four different types of requests sent from a client to the server. When clients send a request to the server in order to update or get data, one of the following request codes should be specified in the request message.

- UPDATE_WSTAT: to update PER-FILE word length statistics.
- GET_MY_UPDATES: to get the current number of updates of a client.
- GET_ALL_UPDATES: to get the current number of updates of all clients.
- GET_WSTAT: to get the current word length statistics from the server.

When the server response to the request, one of the following response codes and requested data should be specified in the response message.

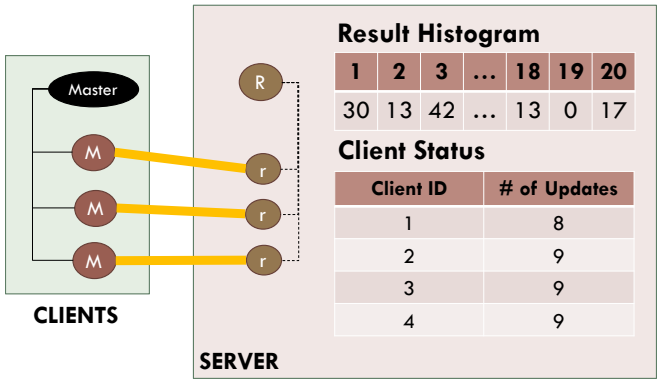
- RSP_NOK: to indicate unsuccessful request. When a client receives a response with RSP_NOK, the server and the client should close their TCP connection.
- RSP_OK: to indicate successful request.




< Figure 1: Process and thread relationships >

## 2.2 The server program

The server program is a multi-threaded server as shown in Figure 1. The server is responsible for listening for incoming socket connection requests and establishing connections with clients. The server waits for connections on the specified port. When it receives a connection request, it should spawn a new thread to handle that connection. The thread is responsible for handling a request from a client by reading client's message from a socket set up between them, doing necessary computation (refer to Section 3. Communication Protocol) and sending a response to the client back through the socket. The thread should exit after it processes a request and closes the client connection. For example, a client sends to the server four different types of messages stated in Section 2.1, and for different threads handle individual messages using separate TCP connections between the client and server. For the extra credit, a thread handles all the multiple requests from a client. Please refer to Section 5. Extra Credits if you attempt it.




< Figure 2: Information and data structure the server maintains >

Server maintains two data structures, ***ResultHistogram*** and ***ClientStatus***, as shown in Figure 2. *ResultHistogram* is an integer list to maintain the accumulated word length statistics sent by clients and *ClientStatus* is another integer list to count the number of requests conveying word length statistics from individual clients. You need to apply a synchronization method on those shared resources.

**Table 1: Attributes of *ClientStatus***

| Name | Purpose |
|---|---|
| Client ID | Unique client ID, which starts from 1 and increments by 1 |
| Number of Updates | Incremented by 1 when receiving a request with request code “UPDATE_WSTAT”. It eventually has the same value with the total number of files that each client processes. |

## 2.3 The client program

The client program spawns multiple client processes (mapper processes) as shown in Figure 1. Each client process is assigned a unique client ID when it is spawned. The client ID starts from 1 and increments by 1. The processes run **in parallel**. Each client sends the following messages in that order from 1. UPDATE_WSTAT to 4. GET_WSTAT requests.

1. 0 or N UPDATE_WSTAT request(s) (If there is no file, do not send this request. If you attempt extra credit, you need to send this request multiple times to send word length statistics for multiple files.)
2. 1 GET_MY_UPDATES request
3. 1 GET_ALL_UPDATES request
4. 1 GET_WSTAT request

Clients initiate a TCP connection with the server on given IP address and port number for a single request and close the TCP connection. A new TCP connection should be initiated and terminated for each request. For extra credit, a TCP connection is used to exchange all the requests between a client and a server thread. Please refer to Section 5. Extra Credits if you attempt it.

There is a folder where all source files exist. You need to specify the folder as an input parameter when executing the client executable (Refer to Section 7. Execution Syntax). The folder can be empty or contain multiple files (no subdirectories, no symbolic or hard links). The file names will be 1.txt, 2.txt, …, N.txt. Each client reads a file having the same name as their client ID. For example, if there are 4 client processes and 2 files in the folder, a client with client ID 1 should compute and send a UPDATE_WSTAT request for 1.txt, and a client with client ID 2 processes 2.txt file. Last two clients with client ID 3 and 4 should not send UPDATE_WSTAT request since there is no file for them. If there are 4 client processes and 10 files in the folder, first four files from 1.txt to 4.txt are processed by the four client processes and the remaining 6 files are not computed by the clients nor collected on the server side. In case you attempt extra credits, all files in the folder should be handled by client processes. You can use round-robin method to assign the files to each client process.

# 3. 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.

## 3.1 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) | Comment |
|---|---|---|
| Request code number | 1 | Specifies request code number |
| Client ID | 1 | Specifies client ID |
| Data | 20 | Relevant data for the request.<br>If there is no data, fill with zeros |
| Persistent flag | 1 | Used for Extra Credits. Default value = 0.<br>When it is set to 1, server thread should not close TCP connection nor exit. Refer to Section 5 Extra Credits for details. |

**Table 3: Request Codes**

| Request Code Number | Request Code | Required Data |
|---|---|---|
| 1 | UPDATE_WSTAT | Word length statistics for a single file (20 integer values) |
| 2 | GET_MY_UPDATES | No data required. Fill with zeros |
| 3 | GET_ALL_UPDATES | No data required. Fill with zeros |
| 4 | GET_WSTAT | No data required. Fill with zeros |

Details of each request are as follows.

- **1. UPDATE_WSTAT**
  - [Client] A client sends **PER-FILE** word length statistics to the serve. If there is no file assigned to a client, the client SHOULD NOT send this request.
  - [Server] When the server receives this request, it accumulates word length statistics into the *ResultHistogram*, and increases *the number of updates* field of the *ClientStatus* by 1 for the corresponding client. The final value of *the number of updates* field of a client should be the same with the number of files processed by the client.
- **2. GET_MY_UPDATES**
  - [Client] A client sends this request to server to get the number of UPDATE_WSTAT requests the client has sent to the server.
  - [Server] Server returns the current value of *the number of updates* field of *ClientStatus* for the corresponding client.
- **3. GET_ALL_UPDATES**
  - [Client] A client sends this request to server to get the total number of UPDATE_WSTAT requests all clients have sent to the server so far.
  - [Server] Server returns the sum of all values of *the number of updates* field of *ClientStatus*.
- **4. GET_WSTAT**
  - [Client] A client sends this request to server to get the current values of the word length statistics accumulated in the server.
  - [Server] Server returns the current values of the *ResultHistogram* to the corresponding client.

## 3.2 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) | Comments |
|---|---|---|
| Request code number | 1 | Specifies a request code number received |
| Response code number | 1 | Specified a response code number |
| Data | 1 or 20 | Relevant data to be sent from server |

**Table 4: Response Code**

| Response Code Number | Response Code | Comments |
|---|---|---|
| 0 | RSP_NOK | For unsuccessful requests. Returned data should be filled with zeros. |
| 1 | RSP_OK | For successful requests |

**Table 5: Data Returned on Success**

| Request Code Number | Request Code | Data Returned |
|---|---|---|
| 1 | UPDATE_WSTAT | Client ID in the request message (1 integer) |
| 2 | GET_MY_UPDATES | The value of *the number of updates* field of *ClientStatus* for the corresponding client (1 integer) |
| 3 | GET_ALL_UPDATES | The sum of all *the number of updates* field of *ClientStatus* (1 integer) |
| 4 | GET_WSTAT | Word length statistics of *ResultHistogram* at that moment that the request is received (20 integers) |

Servers should handle various error cases by responding RSP_NOK such as:

- When receiving a request with unknown request code
- When client ID is not greater than zero

## 4. 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 logs and expected results by using client and server executables included in the PA4 package. README files in each client and server folders describe how to use the executables for your testing purpose.

- **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",` client ID (request[1]).
  - [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 (successful execution)\n",` client ID (request[1]) when all responses that clients receive are RSP_OK. If clients get a RSP_NOK response, they should close the TCP connection and print `"[%d] close connection (failed execution)\n",` client ID (request[1])
  - [Server] Print `"close connection from %s:%d\n",` client_ip, client_port.
- **1. UPDATE_WSTAT**
  - [Client] Print `"[%d] UPDATE_WSTAT: %d\n",` client ID (request[1]), the total number of UPDATE_WSTAT requests sent to server. **Print out this log after a client sends all UPDATE_AZLIST requests to the server.**
  - [Server] Print `"[%d] UPDATE_WSTAT\n",` client ID (request[1])
- **2. GET_MY_UPDATES**
  - [Client] Print `"[%d] GET_MY_UPDATES: %d %d\n",` client ID (request[1]), Response Code (response[1]), Data (response[2])
  - [Server] Print `"[%d] GET_MY_UPDATES\n",` client ID (request[1])
- **3. GET_ALL_UPDATES**
  - [Client] Print `"[%d] GET_ALL_UPDATES: %d %d\n",` client ID (request[1]), Response Code (response[1]), Data (response[2])
  - [Server] Print `"[%d] GET_ALL_UPDATES\n",` client ID (request[1])
- **4. GET_WSTAT**
  - [Client] Print `"[%d] GET_WSTAT: %d <20 numbers>\n",` client ID (request[1]), Response Code (response[1]), and all data received from the server (20 numbers) in the same line (1 space between numbers)
  - [Server] Print `"[%d] GET_WSTAT\n",` client ID (request[1])

## 5. Extra Credits: Processing Multiple Input Files with Persistent Connection

You can get extra credits (10%) by processing multiple input files per client using a persistent connection between a client and the server. You enter an input file folder as an input argument of the client executable. Clients should process all files in the folder for the extra credit. Files should be assigned to individual client in a round-robin fashion. Clients should send all messages from UPDATE_WSTAT to GET_WSTAT to the server using a persistent connection. That is, a server thread should handle all messages sent from a client using a single TCP connection. For example, if there are 50 files in the folder and 10 client processes, each client should process 5 files and send 5 UPDATE_WSTAT, 1 GET_MY_UPDATES, 1 GET_ALL_UPDATES, and 1 GET_WSTAT messages to the server via a single TCP connection.

Hint 1: You can use the modulo operator to assign source files to individual clients in a round-robin fashion.

Hint 2: You can utilize ***Persistent flag*** field of request message so that the client can inform the server whether the TCP connection between them should persist or be terminated. When this field of a request message is set to 1, the server thread SHOULD NOT terminate the TCP connection nor exit after responding to the request message. The default value of the field is 0 for nonpersistent connection; the server thread should terminate and exit after responding to the request message.

## 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**”, and “**Testcases**” folders. “PA4_Server” should contain “**include**” and “**src**”. Your programs should be successfully compiled by using the provided Makefile.

- “**include**” folder: All .h header files
- “**src**” folder: All .c source files
- “**log**” folder: log_client.txt.
- “**Testcases**” folder: There are 5 Testcases are provided for your testing. Your program will be tested with additional hidden testcases. You can get expected results by running client and server executables included in the PA4 package. README files in each client and server folders describe how to use the executables for your testing. **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 files, with the names used in the distributed Makefile

## 7. Execution Syntax

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

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

- <Server Port> is 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 Clients> <Server IP> <Server Port>
```

- <Folder Name> the folder path where source files are located.
- <# of Clients> the number of client processes to be spawned.
- <Server IP> IP address of the server to connect to.
- <Server Port> port number of the server to connect to.

## 8. Assumptions and Hints

- You should use TCP sockets.
- Maximum number of clients is 20.
- Maximum number of concurrent connections at the server side is 50.
- 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.
- There will be no symbolic or hard links in the input directory.
- 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.
- We will use the GCC version installed on the CSELabs machines(i.e. 9.3.0) to compile your code. Make sure that the code you submit compiles and run on CSELabs: your code compile simply by running make.
- Please make sure that your program works on the CSELabs machines e.g., KH 4-250 (csel-kh4250-xx.cselabs.umn.edu). You will be graded on one of these machines.

## 9. Submission

### 9.1 Interim Submission

You need to implement single GET_WSTAT message handling between a client and server. Minimum requirements are as follows:

- Single TCP connection is established and terminated between a client and server.
- A client sends a GET_WSTAT message to the server.
- The server sends a response to the client with any dummy data. Zeros are okay.
- Logs regarding TCP connection handling and GET_WSTAT message should be printed out to terminal. Do screen capture the logs and submit it with the source codes (server.c and client.c).

Interim submission should be a zip containing the captured logs and two .c files.
Due for the interim submission is **April 21, 2021 (Wed), 11:59pm.**

### 9.2 Final 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 that includes the following details:

- Team member names and x500 addresses
- Your and your partner's individual contributions
- Any assumptions outside this document
- Whether you have attempted extra credit or not

The README 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 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] */
```

Due for the final submission is **May 3, 2021 (Mon), 11:59pm.**

## 10. Grading Policy (tentative)

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. (40%) TCP connection setup between clients and server
    a. (10%) Set up a TCP connection between a client and the server
    b. (10%) Use processes at client side
    c. (10%) Use threads at server side
    d. (10%) Set up multiple TCP connections between clients and the server
5. (30%) Request handling between clients and the server
    a. (15%) Client side
    b. (15%) Server side
6. (10%) Error Handling
7. (10%) Extra credit
    a. (5%) Process multiple input files
    b. (5%) Use a persistent connection