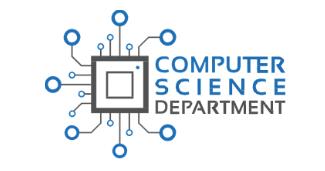
**AMERICAN UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**FACULTY OF ARTS AND SCIENCES**



Fall 2020 – 2021

CSI 402 – Operating Systems

NetPrint

**By**

Mohamad Nizar Daouk

Patrick Balian

**Submitted to**

Mr. Antoine Aouad

**12/01/2021**

**Table of contents**

[Chapter 1 - Problem Statement 3](#_Toc61604821)

[1.2 – Segmentation of the Problem 3](#_Toc61604822)

[Chapter 2 – Handling Requests 4](#_Toc61604823)

[2.1 – Client-Server Model 4](#_Toc61604824)

[2.1.1 – Client-Server Relation 4](#_Toc61604825)

[2.2 – Socket Programming 5](#_Toc61604826)

[2.2.1 – Establishing a Connection 5](#_Toc61604827)

[2.3 – Client-Server communication 6](#_Toc61604828)

[2.3.1 – Protocol Design 6](#_Toc61604829)

[Chapter 3 – User Credentials 8](#_Toc61604830)

[Chapter 4 – Priorities 9](#_Toc61604831)

[4.1 – Printing 9](#_Toc61604832)

[Chapter 5 – Multithreading 10](#_Toc61604833)

[5.1 – Mutual Exclusion 10](#_Toc61604834)

[Chapter 6 – User Interface 12](#_Toc61604835)

# Chapter 1 - Problem Statement

Design a printer queue that is responsible for handling the printing requests coming from different users. You have to take into consideration that users have different levels of privileges and priorities.

Each user has an identification number and a password, in addition to printing priorities. One good idea is to design the queue using an array or pointers while preserving the first-in first-out concept of the queue. For every printing request received, the program should check the privileges of that request and whether it can be moved forward in the queue to be served by the printer prior to serving the other requests.

Using the programming language of your choice (preferably C++), write the printer queue that would handle the user request. The program must allow for requests coming from different users or from one user.

## 1.2 – Segmentation of the Problem

Let’s adopt a divide and conquer strategy, looking closer at the problem statement provided by the instructor, we can segment it into 5 subproblems.

1. Our Printer needs to be able to handle requests coming from different users.
2. Users should be identified by an ID number associated with a password.
3. Users have privileges that limit them from performing certain actions.
4. Users have priorities giving their print requests precedence.
5. The system will be able to handle multiple users simultaneously.

In chapter 1 we discussed the problem statement and how it can be divided into 5 parts, which we will tackle in the following chapters.

# Chapter 2 – Handling Requests

In order to handle requests from different users, we decided to use a Client-Server architecture.

In our scenario, the printer would be the server listening and accepting connections from the clients which are our users.

## 2.1 – Client-Server Model

In a Client-Server architecture, the server is the one that provides for the client, the consumer. The server owns and provides high-end, computing services to the client on demand. The most notable of them are application access, storage, file sharing, printer access and/or direct access to the server’s raw computing power.  
On the other hand, a client does not share any of its resources, it only requests content or service from the server.  
In a Typical scenario, servers would be idly listening and waiting for a request from a client that initiates communication sessions.

### 2.1.1 – Client-Server Relation

The Client-Server relation is one to many, and it is presented in the figure below.

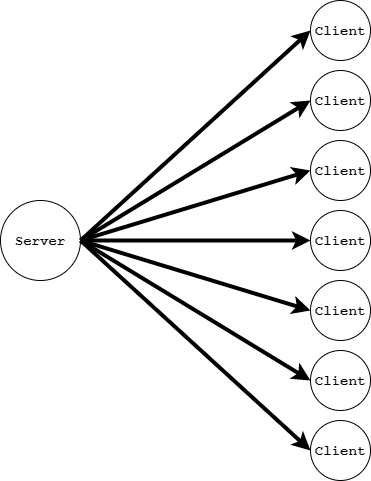


Figure 2 - Client-Server one-to-many relation

## 2.2 – Socket Programming

Sockets are a key factor in our approach as they enable us to communicate between processes.

A socket programming interface provides the routines required for inter-process communication between applications, either on the local system or spread in a distributed, TCP/IP based network environment. Once a peer-to-peer connection is established, a socket descriptor is used to uniquely identify the connection. The socket descriptor itself is a task specific numerical value.

One end of a peer-to-peer connection of a TCP/IP based distributed network application described by a socket is uniquely defined by:

* Internet address
  + for example, 127.0.0.1 (in an IPv4 network) or FF01::101 (in an IPv6 network).
* Communication protocol
  + User Datagram Protocol (UDP)
  + Transmission Control Protocol (TCP)
* Port
  + A numerical value, identifying an application. We distinguish between "well known" ports, for example, port 23 for Telnet, port 80 for http …
  + User defined ports

### 2.2.1 – Establishing a Connection

To establish a connection between a server and a client, sockets must be initialized in a specific order.

On the server’s side:

Step 1, socket creation:

int sockfd = socket(domain, type, protocol)

* sockfd : socket descriptor, an integer (like a file-handle)  
  domain: integer, communication domain e.g., AF\_INET (IPv4 protocol) , AF\_INET6 (IPv6 protocol)
* type: communication type
  + SOCK\_STREAM: TCP(reliable, connection oriented)  
    SOCK\_DGRAM: UDP(unreliable, connectionless)
* protocol: Protocol value for Internet Protocol (IP), which is 0. This is the same number which appears on protocol field in the IP header of a packet.

Step 2, Bind:

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

After creation of the socket, bind function binds the socket to the address and port number specified in addr (custom data structure).

Step 3, Listen:

int listen(int sockfd, int backlog);

It puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection. The backlog defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED.

Step 4, Accept:

int new\_socket= accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);

It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. At this point, connection is established between client and server, and they are ready to transfer data.

As for the Client’s side:

Step 1, socket creation:

Socket creation is the same as the server.

Step 2, Connect:

int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

The connect() system call connects the socket referred to by the file descriptor sockfd to the address specified by addr. Server’s address and port is specified in addr.

## 2.3 – Client-Server communication

Now that we have established a connection, the two entities need to communicate with each other.

Communication needs to happen based on a language that both entities can understand.

For this purpose, we develop a simple application-level protocol for client server communication.

### 2.3.1 – Protocol Design

Network protocols are sets of established rules that dictate how to format, transmit and receive data so computer network devices -- from servers and routers to endpoints -- can communicate regardless of the differences in their underlying infrastructures, designs or standards.

To successfully send and receive information, devices on both sides of a communication exchange must accept and follow protocol conventions.

Graphical user interface, text, application

Description automatically generatedIn our case, a 1996 bytes payload size with a 4 bytes header, are enough to fit simple print requests.

Figure 2 2 - Packet composition

Protocol codes, placed in the header, need to be defined in order to identify and process requests accordingly.

We define the following protocol codes:

“LGIN”: sent by the client when logging in, followed by his ID and password.

LGIN-ID-Password

“AKIN”: sent by the server the acknowledge a login request, followed by a binary value indicating if login was successful.

AKIN-1 or AKIN-0

“NWID”: sent by a client with administrative privileges to request a new id for registering a new user.

“DEFS”: sent by the server when replying to a “NWID” request followed by an incremental ID

DEFS-NewID

“RGST”: sent by a client with administrative privileges after a NWID request to register a new user followed by ID, Password, Priority, Privilege level.

RGST-ID-Password-Priority-PrivilegeLevel

“PRNT”: sent by a client with appropriate privilege to request printing a string.

PRNT-String

“AKPT”: sent by the server to acknowledge a print request, followed by a binary value indicating success or failure.

AKPT-1 or AKPT-0

# Chapter 3 – User Credentials

For simplicity’s sake we opted to use a HashMap to store user information.

A class “User” was created with the following data members:

class User {

private:

int m\_id; //Idenftification Number

std::string m\_password; //Password

short m\_priority; //Printing Priority (Low 1, Medium 2, High 3)

short m\_privLevel; //Privileges (Limited User 1 / Normal User 2 / Admin 3)

The HashMap Keys would be the user’s ID and the value the actual User object containing all the information pertaining to one user.

//Register new user

User newUser(newID, newPass, newPriority, newPrivLevel);

Users[newID] = newUser;

Access and modification to a HashMap has a time complexity of O(1) allowing login requests to be processed faster.

# Chapter 4 – Priorities

Print requests need to be handled based on a priority. Having a small finite set of priority values: 1 for low, 2 for normal and 3 for high.

We have decided to implement a hybrid Data Structure combining both HashMap and Vector available in C++ standard library.

This approach has the following advantages:

* Access time complexity of O(1) to priorities, unlike a standard queue where you would have to traverse all of it to find the priorities and then sorting them according to their values taking at worse O(n^2)
* A First In First Out behavior in the vectors related to each priority, no need to sort them as they would all have the same priority inside the same vector.

std::map <short, std::vector<std::string> > printQ = {

{1, {}},

{2, {}},

{3, {}}

};

## 4.1 – Printing

Printing becomes a trivial task now that our data is well structured inside the hybrid Data Structure.

for (int i = printQ.size(); i >= 1; i--) {

if (!printQ[i].empty()) {

for (int z = 0; z < printQ[i].size(); z++) {

std::cout << "[|] Printed: " << printQ[i][z] << " (Priority " << i << ")" << std::endl;

}

printQ[i].clear();

}

}

After Printing out all the print request of one priority we clear its vector and proceed to a lower priority.

# Chapter 5 – Multithreading

To be able to handle multiple users simultaneously the use of threads was inevitable.

Threads in C++ have the following syntax:

std::thread thread\_obj(TargetFunction,FunctionArgs);

Once a connection has been accepted by the server, a thread is created targeting a function that handles connections and that takes as argument the new socket resulting from the server accepting the new connection.

while (true) {

SOCKET new\_socket = accept(s, (struct sockaddr\*)&client, &c);

ncon++;

connections[ncon] = new\_socket;

if (new\_socket == INVALID\_SOCKET) {

std::cout << "[-] Error has occured\n[-]Could not accept connection" << std::endl;

return WSAGetLastError();

}

std::thread thread\_obj(handleConnections, connections[ncon]);

thread\_obj.detach();

}

At the start of the server a printA() thread would have started running In parallel with the main thread.

The sole purpose of this thread is to print what has been accumulated inside the vectors of the HashMap.

## 5.1 – Mutual Exclusion

In order to have uninterrupted printing we had to implement mutual exclusion using Semaphore, that way we would avoid two threads using the HashMap at the same time.

Semaphores were employed inside 2 functions, printA() which reads from the HashMap, prints the request then clears the vectors and while inserting a new print request into the HashMap.

if (reqCode == "PRNT") {

//std::cout << "entered print" << std::endl;

std::string toPrint = splitOn(reqStr, 1);

while (semaphore < 0) {

std::cout << "[-] Waiting.." << std::endl;

Sleep(100);

}

semaphore--;

//handle the print request

//critical part

std::cout << "[+] Entered critical section" << std::endl;

printQ[Users[id].getPriority()].push\_back(toPrint);

semaphore++;

void printA(){

while (true) {

//std::cout << "entered printA" << std::endl;

Sleep(5000);

while (semaphore < 0) {

std::cout << "[-] Waiting.." << std::endl;

Sleep(500);

}

semaphore--;

//std::cout << "entered printA crit" << std::endl;

//handle the printing

for (int i = printQ.size(); i >= 1; i--) {

if (!printQ[i].empty()) {

for (int z = 0; z < printQ[i].size(); z++) {

std::cout << "[|] Printed: " << printQ[i][z] << " (Priority " << i << ")" << std::endl;

}

printQ[i].clear();

}

}

semaphore++;

}

}

That way when multiple users request to print at the same time, the requests will be added to the HashMap one at a time, while doing so the print thread would wait.

Once the print thread has begun printing, threads that want to insert new request into the HashMap will have to wait until printing is done.

# Chapter 6 – User Interface

A picture containing text, monitor, black, screen

Description automatically generated

Whenever NetPrint is first launched by a client, the user is presented with a console containing ASCII art of NetPrint and some options that may be entered, include logging in and quitting.

A prompt is available for entering the chosen option, which can be performed by entering the first letter of these options, l or q.

Quit will terminate the program, while logging in will initialize a connection to the NetPrint server and prompt the user for the ID and password as shown in the image below. A computer screen capture

Description automatically generated with low confidence

On the other hand, the server program looks as such:

A picture containing text, monitor, screenshot, black

Description automatically generated

Upon launching the server, some ASCII art and output detailing the fact that the server is running is displayed.

Upon entering the login information, the server displays the following information to signal that a LGIN request has been received with and ID of 0 and a password of niz12345, along with it replying with an AKIN response.

A computer screen capture

Description automatically generated with low confidence

Upon logging in, the user privilege level and account type is presented on the top with the available options accordingly. Again, entering the first letter of each option is enough to proceed to the next menu.

A computer screen capture

Description automatically generated with low confidence

Entering ’r’ will open the register user menu where a password for the new account is prompted for. Entering the password will send a request to the server and await a reply for the new ID of the user. After receiving the new ID, the user may now also enter the priority and privilege levels of the new user.

A computer screen capture

Description automatically generated with low confidence

The server-side implementation will display the received new ID request and the DEFS response sent.A picture containing text, monitor, screenshot, black

Description automatically generated

After entering the privilege and priority levels, the user is returned to the main menu, and the server displays the following output to signal that it successfully received the register user request.

A computer screen capture

Description automatically generated with low confidence

Entering P will bring up the print menu, where data is waited for printing.

A screenshot of a computer

Description automatically generated with medium confidence

On the server, the following output is presented:

A screenshot of a computer

Description automatically generated with medium confidence

The output can be understood as that a new user with ID 1 has logged in and sent a print request to the server with the following bytes as data: “hello this is user 1”.

The server outputs some additional information for when it enters the critical section and replies with an acknowledgement that it received the print request. Upon printing successfully, a “[|] Printed: hello this is user 1 (Priority 2)” is presented as well to signal that the printing job is completed.

If at the same time, both users sent a print request, then the server output would display this:A computer screen capture

Description automatically generated with low confidence

The above image shows that even though both users sent a print request at the same time, proper printing procedure will occur accordingly.

Upon logging out, the client is returned to the login screen, while the server log shows that it received a logout request and that the connection with the user with ID 0 has been terminated.

A screenshot of a computer

Description automatically generated with medium confidence