Project: Final Report

Cloud Prototype

By Sayali Borase

TABLE OF CONTENTS

1. Abstract ……2

2. Introduction …… 2

2.1 Cloud Computing …… 2

2.2 Socket Programming ……3

3. Project Description and Requirements …… 3

4. Implementation and Execution …….. 4

4.1 Environment Setup …….. 4

4.2 Establish TCP Client-Server Socket Connection …… 4

4.3 Concurrent Servers …… 7

4.4 Client Authentication/Login …… 8

4.5 Read, Write and Delete operation …… 10

4.6 File Access Permission …… 11

4.7 Synchronization …… 12

4.8 Critical Section …….. 13

5. Experiment Result …… 16

6. Conclusion …… 33

7. Source Code …… 33

8. Project Demo …… 33

REFERENCES …… 34

1. Abstract

Cloud computing enable convenient, on-demand resource access to heterogenous devices with the ability to scale up or down their service requirement. The computing resources based on pay-per-use pattern eliminate the requirements for setting up of high-cost infrastructure. In a cloud computing environment, the entire data and resources resides over a set of remote servers, enabling shared access over the network. In this paper, we present a prototype of the cloud environment based on network socket programming using C++ language. We implement two serves on separate machines that concurrently serve client requests which are again hosted on different machines of local area network (LAN). We also employ client authentication, synchronization and file access permissions on the server side. The servers also enforce the critical section check.

2. Introduction

2.1 Cloud Computing

Cloud Computing is the combination of a technology, platform that provides hosting and storage service on the Internet. Cloud services are generally maintained by third parties and consumers of these services do not possess resources in the cloud model but pay for them on a per-use basis [1]. This feature support high scalability, multitenancy and inexpensive on-demand computing resources with good quality of service. Benefits of Cloud computing are enormous. The most important ones are reduced IT cost, business continuity, and speed.

Cloud computing architecture mainly consist of two sections – front end and back end. Its data is replicated and preserved remotely as part of the cloud configuration. Front end includes user interface and the client's computer system that utilizes the cloud resources over the network. On the back end of the system are the various servers and data storage system and that forms the "cloud" of the computing services [4]. The back-end side is also responsible for providing security mechanisms, traffic control, and protocols that connect networked computers for communication. Application are processed remotely on cloud server that provides necessary infrastructure, processing capacity and data storage. In short, all applications are controlled, managed, and served remotely by a cloud server and client system access those application as an when needed.

Based on the cloud computing architecture, we develop a simple prototype using client-server socket programming in C++. The model consists of two servers which runs on virtual system A and virtual system B respectively. Two clients which are hosted on local system, communicates with server over local area network (LAN).

2.2 Socket Programming

According to definition [4], "A socket is one endpoint of a two-way communication link between two processes running on the network". An endpoint is a combination of an IP address

and a port number. Client and servers communicate by means of multiple layers of network protocols. In this paper, we will be using TCP/IP protocol suite.

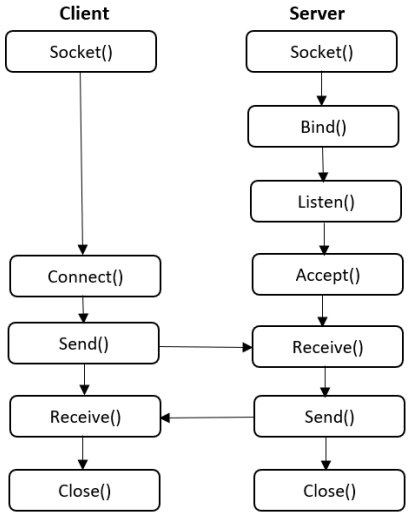


Figure 1: Client-server socket [5]

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. The client tries to establish connection with server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

3. Project Description and Requirements

We divide the project requirements into following functionalities or modules:

- **Concurrent servers:** Two clients may request services from the same server at the same time. This implies that the server should be able to handle two clients request simultaneously.
- **Client authentication:** Each server maintains separate file for storing user login data. For every connection with client, server verify if it’s a new user or returning user (by checking user login file). If it’s a new user, server prompt for username and password and store it in file. In case of incorrect login information (for returning user), server prompt to re-enter the password and disconnect if user exceeds three attempts.
- **Read, write and delete operation:** Client can perform read, write and delete operations on the file saved on remote server. Clients send ‘exit’ to terminate the connection with the server.

- **File access permission:** Each client has the persimmon to read each other's file. However, a client can only write and delete the content of its own file.
- **Synchronization**: In order to maintain consistency, both the servers synchronize the information in two scenarios.
    - Server sends the login information to other server in case of new client logs into the server.
    - Server synch the file content with other server in case of write and delete operation.
- **Critical section:** Concurrent access to shared resources on both the servers may lead to unexpected or erroneous behavior. This functionality of this module is to verify the user data is consistent at any point of time. Both the clients can read the data at the same time. A client cannot read/write/delete the data when one client is writing or deleting data to the server. Also, the client cannot read/write/delete the data when a server is synchronizing with another server.

4. Implementation and Execution

We briefly discuss each component mentioned in section 3 and our approach to implement the functionality associated with each component.

4.1 Environment Setup:

- Server A is implemented on Ranger system (linux system) and server B on Shemp (Windows server 2008). Clients A and B are implemented on local host (Windows 10).
- We are using TCP/IP protocol suit for communication over LAN.
- We use C++ to implement functionality (preferably g++ complier).
- We use following libraries that support socket programming in both windows and linux.

```
#ifdef _WIN32
#include <winsock2.h>
#include <ws2tcpip.h>
#else
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif

#pragma comment(lib, "ws2_32")
```

4.2 Establish TCP Client-Server Socket Connection:

i. Socket Addresses:

IPv4 socket address is named sockaddr_in and is defined in <netinet/in.h> header. The POSIX definition is given figure below.

```
struct sockaddr_in {
        uint8_t sin_len; /* length of structure (16)*/
        sa_family_t sin_family; /* AF_INET*/
        in_port_t sin_port; /* 16 bit TCP or UDP port number */
        struct in_addr sin_addr; /* 32 bit IPv4 address*/
        char sin_zero[8]; /* not used but always set to zero */
};
```

ii. Socket Connection:

Below are the steps for establishing a TCP socket at the client side:

- Create a socket using socket() function.
- Connect the socket to the address of the server using the connect() function. Here, the clients need to specify the host IP address and the port to which it wants to connect.
- Send and receive data by means of the send() and recv() functions.

And, the steps involved in establishing a TCP socket at the server side are as follows:

- Create a socket with the socket() function.
- Bind the socket to an address using the bind() function.
- Listen for connections with the listen() function.
- Accept a connection with the accept() function system call. This call typically blocks until a client connects with the server.
- Send and receive data by means of send() and recv().

iii. Socket() function:

This function specifies type of communication protocol (TCP based on IPv4). It returns socket descriptor or -1 on error.

```
// Create a socket for the client
// If sockfd<0 there was an error in the creation of the socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
{
    cerr << "Error in establishing the client 1 socket";
    exit(1);
}
```

iv. Connect() function:

The function returns 0 if the it succeeds in establishing a connection, -1 otherwise.

```
//Connection of the client to the socket
status = connect(sockfd, (sockaddr*)&servaddr, sizeof(servaddr));
if (status < 0)
{
    cerr << "Error in connecting to server socket";
    close(sockfd);
    exit(1);
}
cout << "Connected to the server" << endl;
```

v. Bind() function:

This function assigns a local protocol address to a socket. The address is the combination of IPv4 address along with the TCP port number. It returns 0 if it succeeds, -1 on error. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

```
//preparation of the socket address
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

//bind the socket
bindStatus = bind (listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));
if(bindStatus < 0)
{
    cerr << "Error in binding the server socket";
    exit(1);
}
cout << "Bind to port: " << SERV_PORT << endl;
```

vi. Listen() function:

This function converts an unconnected socket into a passive socket. LISTENQ is the maximum number of connections that a server can accept. Once the queue is full, system rejects any additional connection requests. For this project, we are allowing up to client connection. However, this can be altered as required.

```
//listen to the socket by creating a connection queue, then wait for clients
listen (listenfd, LISTENQ);

cout << "Server running...waiting for connections" << endl;
```

vii. Accept() function:

This function retrieves a connection request. A new file descriptor listenfd is connected to client that calls the connect(). The original socket descriptor remains available to receive additional connect request.

```
//accept a connection
connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen);
if(connfd < 0)
{
    exit(1);
}
cout << "Connection accepted from " << inet_ntoa(cliaddr.sin_addr) << ":" << ntohs(cliaddr.sin_port) << endl;
```

viii. Send() and Recv() function:

These functions specify options to read and write data. We declare a character array(buffer) to send and received bulk data between client and server.

```
memset(&buffer, 0, sizeof(buffer)); //clear the buffer
strcpy(buffer, msg.c_str());
bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);
cout << "> Awaiting client response for username..." << endl;

memset(&buffer, 0, sizeof(buffer)); //clear the buffer
bytesRead += recv(connfd, (char*)&buffer, sizeof(buffer), 0);
```

ix. Close() function:

This function closes a socket and terminate a TCP socket.

4.3 Concurrent Servers:

We need to design parallel TCP/IP socket server that will support multithreading or multiprocessing environment. This means, a server can receive multiple client requests (in our case, 2 clients) at the same time and server each client in parallel so that no client will have to wait.

There are two ways we can achieve this:

- **Multithreading (Multithreaded approach):**

In this, we create a new thread for every new client connected to the server. This is included in header file <pthread.h>
The following steps describes the concurrent server process [7]:

- When a connection request arrives at a concurrent server, the server's main thread create a thread and pass client request to that thread with its ID.
- The thread processes the client request, generate the report, and send it back to client.

This approach requires single process to spawn multiple threads to create socket connection.

```
// For each client request creates a thread and assign the client request to it to process
// So the main thread can entertain next request
if( pthread_create(&tid[i], NULL, socketThread, &newSocket) != 0 )
    cout << "Failed to create thread" << endl;
```

- **Fork (Multiprocessing approach):**

In this, we create one child process for each client.
It returns 0 if its child and the process ID of the child in parent; otherwise, -1 on error. In fact, the function fork() is called once but returns twice. It returns once in the calling process (called the parent) with the process ID of the newly created process (its child). It also returns in the child, with a return value of 0. The return value tells whether the current process is the parent or the child [5].

- After the fork() the program concurrently executes two processes.
- Parent process virtual address space is replicated in the child Including the states of variables, mutexes
- The child inherits copies of the parent's set of open file descriptors
- To avoid any unpredictable result, we need to synchronize the execution of two or
- more processes

```
// Support multiple clients
if ( (childpid = fork ()) == 0 )
{
    // If it's 0, it's child process
    cout << "Child process created for dealing with client requests" << endl;

    // Avoid creating zombie processes
    if(childpid != 0)
    {
        signal(SIGCHLD, waitParent);
        while(1);
    }

    // Close listening socket
    close (listenfd); // Child closes listening socket
```

- **Forking a process with synchronization:**
  - The parent process forks a child process to perform a computation, goes on in parallel, and then it reaches an execution point where it needs to use the output data of the child process
  - The waitpid(pid_t) call allows the caller process to wait for a specific child process. It also allows the system to release the resources associated with the child process.
  - If a child terminates, without wait() performed, it remains in a "zombie" state which consumes a slot in the kernel process table [9].

  For this project, we are using combination of mutex and flock file locking mechanism to achieve synchronization between shared variables during concurrent execution.

  Note: Fork() function only works in Linux environment and requires multiple processes to create socket connection.

4.4 Client Authentication/Login:

Client can only access the read/write/delete services after successful login. Both Server A and server B maintains userInfoA.txt and userInfoB.txt respectively. The file contains user login information such as username, password, and a key. The key is used for managing the file permissions. These details are discussed in Section 4.6. The login file also contains a flag (WRITE_FLAG_OFF) associated with every file. We will discuss the significance of the flag Section 4.8.

Sample login data at serverA:

```
user1 pass1111 key13 user1fileA.txt WRITE_FLAG_OFF
user2 pass2222 key68 user2fileA.txt WRITE_FLAG_OFF
```

Once the connection established, client needs to authenticate to access the services.
When a user logged in for the first time, server checks if the user already exist in the file. If not, server consider it as a new user and store (append) login details in a file. Server generates a random unique key and an empty file for user1 with a file name "user1fileA.txt". Client can access this file to read, write and delete the contents in a file. We assume one file per user .

In case of returning user, server verify its login details and allow three attempts before it disconnects the client.

With every unsuccessful login attempt, server notifies client about the remaining attempts.

We wrote a function that generates 4-digit random numeric key which remains unique for each client process. The length of the key can be altered by changing MAX_LIMIT and MIN_LIMIT values as required.

```
    // User is not present in a file (New user)
    // Add user details to file
    // Generate a unique key
    pthread_mutex_lock(mutex);
    srand((int)time(&t) % getpid()); // seed based on current process ID
    key = rand() % (MAX_LIMIT - MIN_LIMIT) + MIN_LIMIT;
    pthread_mutex_unlock(mutex);
```

After generating a key we also create a new empty file for the user (following the name convention userfileA.txt, where A means, file hosted at serverA) and make an row entry in the login file as shows in below code.

```
if(userPresent == 0)
{
    // User is not present in a file (New user)
    // Add user details to file
    // Generate a unique key
    srand((int)time(&t) % getpid()); // seed based on current process ID
    key = rand() % (MAX_LIMIT - MIN_LIMIT) + MIN_LIMIT;

    // Generate a new file
    ofstream outUserFile;
    string userFileName = user + "fileA.txt";
    outUserFile.open(userFileName, ios::out);
    if(outUserFile.is_open())
    {
        outUserFile << "file is empty";
    }
    outUserFile.close();

    // Initialize the writeDeleteFlag to OFF
    writeDeleteFlag = "WRITE_FLAG_OFF";

    ofstream outFile;
    outFile.open(loginFile, std::ios_base::app | std::ios_base::out);
    if(outFile.is_open())
    {
        outFile << user << " " << pass << " " << key << " " << userFileName << " " << writeDeleteFlag << endl;
    }
    outFile.close();

    cout << ">";
    cout << "Client " << user << " is logged in" << endl;
```

Figure 2 explains the basic flow of authentication process.

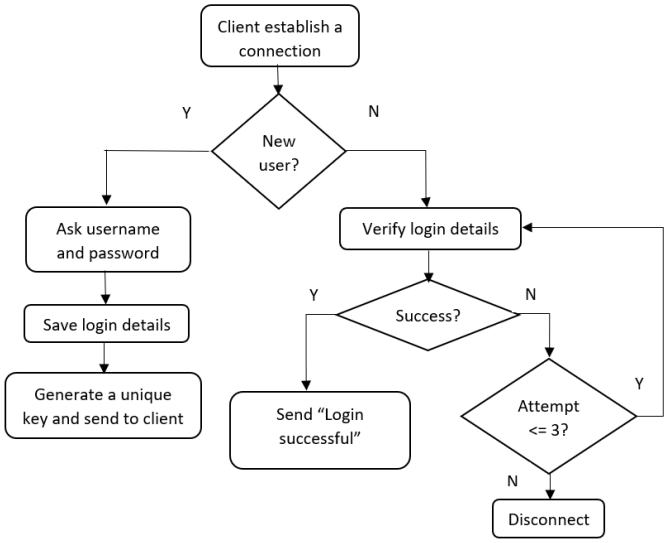


Figure 2: Client authentication

4.5 Read, Write and Delete operation:

A default file is created for each user when client logs in for the first time. Therefore, both the servers maintain a separate files for each client. For example, aliceuserA.txt file belong to Alice on serverA and so on. Apart from that, each server keeps a temp file which is used during the synchronization process after user exits.

Server allow to access file services only after successful login. Server provide four options that client can choose from – Read, Write, Delete, and Exit. Client can type "exit" to exit at any point of time during the active session.

- **Read operation:**

Both the clients can read each other's file at the same time. However, read operation is not allowed on a file when owner is making changes to the file.

Below is the flow of program for read operation:

- Client request for a read operation and specify filename. For eg: read aliceuserA.txt
- Server checks two conditions:
    - Whether there are any changes currently happening to the file. Basically, server checks if the file is edited by the owner of the file.
    - Whether the server is synchronizing with the remote server.
- Server permits read access if no such changes.
- Server fetches file content and display it to the user.
- In the opposite scenario, server denies read permission and send message indicating that a client cannot access the file at this moment.

- **Write/Delete operation:**

Client can only write/delete the contents of its own file.

Below is the flow of program for write/delete operation:

- Client request for a write/delete operation and specify filename.
- Server checks two conditions:
    - Whether the requesting client is the owner of the file. To verify client authorization, server ask for a unique key - which was assigned to client when it logs in for the first time.
    - Whether the server is synchronizing with the remote server Permits access only if the entered key matches with the one stored at server side.
- Server permits write/delete access if no such changes.
- Server fetches file content and display it to the user.
- Server prompts for the new data to be entered to the file.
- In case of write operation, file content is appended to the existing content. In case of delete, the file is overwritten by the new content.
- Server acknowledges the write/delete operation.

4.6 File Access Permission:

Each client is assigned a unique key when it logs in for the first time. When a client request for a write or delete operation, server prompts for a key. Server then verifies the key with the one stored at server side. If a match found, then client is given access to modify the file. In this way, each client can only modify its own file. (Requesting client should be owner of the file). Please note, server do not check for file permission for read operation.

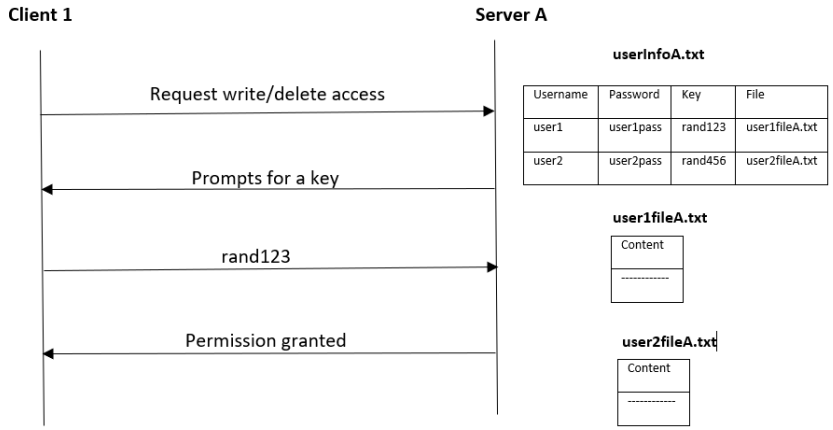


Figure 5: File Access Permission Check

```
// Write opration
if(arr[0] == "write")
{
    int keyVal;

    // Critical section check - each client can write to its own file
    // Ask user for a key to check if user have permission to access file
    serverMsg = "Please enter a key";
    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    strcpy(buffer, serverMsg.c_str());
    bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);
    cout << "> Awaiting client " << user << " response... for key" << endl;

    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    bytesRead += recv(connfd, (char*)&buffer, sizeof(buffer), 0);
    cout << "Client " << user << " message: " << buffer << endl;
    keyVal = atoi(buffer);

    inFile.open(loginFile, ios::in);
    if(inFile.is_open())
    {
        while (inFile >> userText >> passText >> key >> userFileText >> writeDeleteFlag)
        {
            if(userFileText == arr[1] && key != keyVal)
            {
                strError = "You have no permission to write to this file!";
                writeAllowed = false;
            }
        }
    }
    inFile.close();
```

```
// Delete operatio
if(arr[0] == "delete")
{
    int keyVal;

    // Critical section check - each client can modify (update/delete) contents of its own file
    // Ask user for a key to check if user have permission to access file
    serverMsg = "Please enter a key";
    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    strcpy(buffer, serverMsg.c_str());
    bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);
    cout << "> Awaiting client " << user << " response... for key" << endl;

    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    bytesRead += recv(connfd, (char*)&buffer, sizeof(buffer), 0);
    cout << "Client " << user << " message: " << buffer << endl;
    keyVal = atoi(buffer);

    inFile.open(loginFile, ios::in);
    if(inFile.is_open())
    {
        while (inFile >> userText >> passText >> key >> userFileText >> writeDeleteFlag)
        {
            if(userFileText == arr[1] && key != keyVal)
            {
                strError = "You have no permission to update this file!";
                writeAllowed = false;
            }
        }
    }
    inFile.close();
```

4.7 Synchronization:

Server ensures the data consistency in two cases:

- **When new entry is made to the login file, userInforA.txt and userInfoB.txt**
  Server append the user details to the file, when a client logs in for the first time. We assume that a client will "exit" after the first login so the local server can synchronize the content of the file at the remote server. In this case, loginFile maintaining the user login details is synchronized.

- **When client performs write or delete operation on a file**
  After client exit the active session, server checks for a type of operation user performed on a file. In case of write or delete operation, server initiate the synchronization process. Here, the content of the file is synchronized.

In both cases, local server copies the file contents to temp file and send it to the remote server. Remote server then copies the contents to the respective file. This is how both the servers synchronizes the data. Clients need to wait until the server finishes the synchronization.

Figure [3] explains the synchronization process.

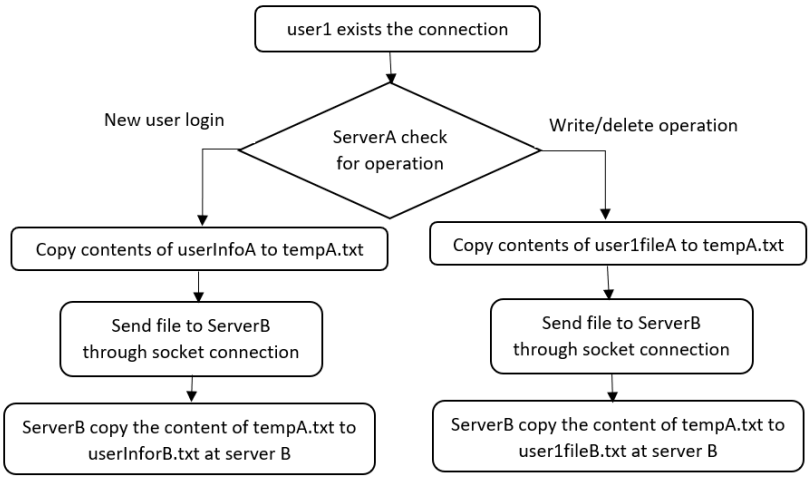
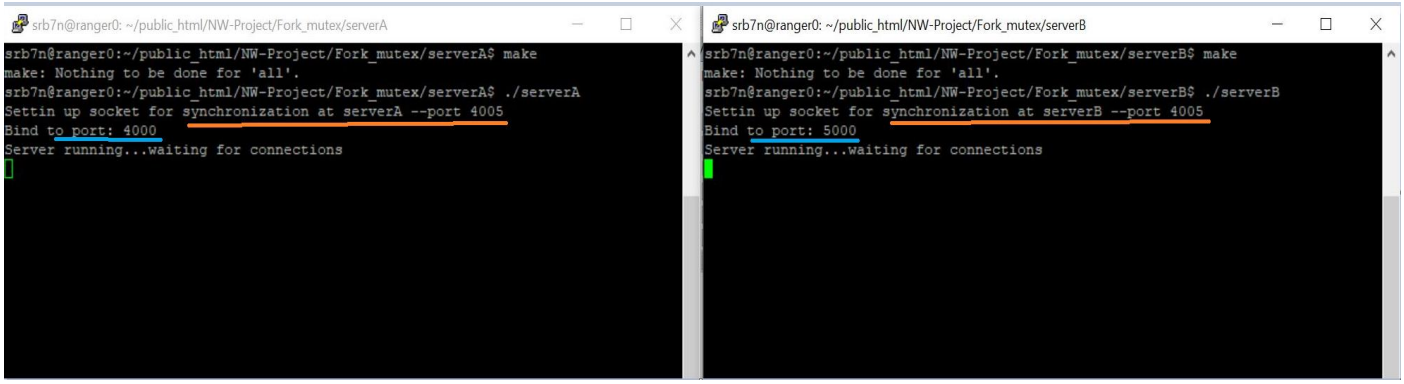


Figure 3: Synchronization at Server A

After both servers finished binding to the port, we run a synchronization setup where both servers create a socket connection at port 4005. Both servers keep listening at port 4005 which is dedicated specifically for synchronization purpose. When serverA received data at this port, it knows that the data is coming from serverB for synchronization.



Below is the sample code that demonstrates synchronization with another server when client "exits" the socket connection. After user exits, server initiates the synchronization based on the user actions (New user will cause changes in login file and write/delete operation will cause changes in userfile) as shown in Figure 3. To ensure the critical section check, the synchronization code is executed within the mutex lock system.

```
if(!strcmp(buffer, "exit"))
{
    cout << "Client " << user  << " has quit the session" << endl;

    // Check if new user - synchronize user login details with serverB
    // Check if user made changes to the file - synchronize file content with serverB
    cout << "Please wait! Server is synchronising" << endl;

    pthread_mutex_lock(mutex);

    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    FILE *fp1, *fp2;
    fp1 = fopen(loginFile.c_str(), "r");
    while(!feof(fp1))
        fread(buffer, sizeof(buffer), 1, fp1);

    // Send login file to serverB
    send(connfdSync, (char*)&buffer, strlen(buffer), 0);

    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    fp2 = fopen("user1fileA", "r");
    while(!feof(fp2))
        fread(buffer, sizeof(buffer), 1, fp2);

    // Send user file to serverB
    send(connfdSync, (char*)&buffer, strlen(buffer), 0);

    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
    recv(connfdSync, (char*)&buffer, sizeof(buffer), 0);
    if(!strcmp(buffer, "synchronization complete"))
        cout << "Synchronization complete" << endl;

    pthread_mutex_unlock(mutex);

    cout << ">";
    break;
}
```

4.8 Critical Section:

Here, we are using the concept of locking. With multithreading and locking, only one thread can hold a lock at a time and then execute the set of code without interrupt. When a thread enters the critical section, other thread waits until the first thread finishes. To

achieve this, we use mutually exclusive lock (mutex) which guarantees atomic operation. <Pthread.h> library in C++ supports mutex.

- **Mutex**: Mutexes are used to prevent data inconsistencies due to operations by multiple processes upon the same memory area performed at the same time or to prevent race conditions where an order of operation upon the memory is expected.

  We allocate a shared memory using 'shm_open' and 'mmap' to allow multiple processes to access shared region in a synchronized way. 'PTHREAD_PROCESS_SHARED' allows the mutex to be shared between multiple processes.

```
des_mutex = shm_open(MUTEX, O_CREAT | O_RDWR | O_TRUNC, S_IRWXU | S_IRWXG); // Allocate shared memory

if (des_mutex < 0)
{
    cerr << "Failure on shm_open on des_mutex";
    exit(1);
}

if (ftruncate(des_mutex, sizeof(pthread_mutex_t)) == -1)
{
    cerr << "Error on ftruncate to sizeof pthread_attr";
    exit(-1);
}

mutex = (pthread_mutex_t*) mmap(NULL, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE, MAP_SHARED, des_mutex, 0);

// Set mutex shared between processes
pthread_mutexattr_t mutexAttr;
pthread_mutexattr_setpshared(&mutexAttr, PTHREAD_PROCESS_SHARED);
pthread_mutex_init(mutex, &mutexAttr);
```

```
pthread_mutex_lock(mutex);
pthread_cond_signal(condition); // Acquire a lock
/* Critical section */
pthread_mutex_unlock(mutex); // Unlock


pthread_mutexattr_destroy(&mutexAttr);
pthread_mutex_destroy(mutex); // Deallocate mutex
shm_unlink(MUTEX);
```

We implement the critical section check in the following scenarios:

- Clients cannot access any services at the servers when serverA is synchronizing data with serverB.
  Note: In this project, we are dealing with small size files containing less than 5 record. As per the observation, synchronization is not taking more than a fraction of seconds. So, it is our assumption that it is unlikely for any client to establish sock connection or request any services in such short duration.

- Clients cannot access any services at the servers when owner of the file is making changes (write/delete operation) to the file. Server-side services includes login functionality, read, write, delete operation.

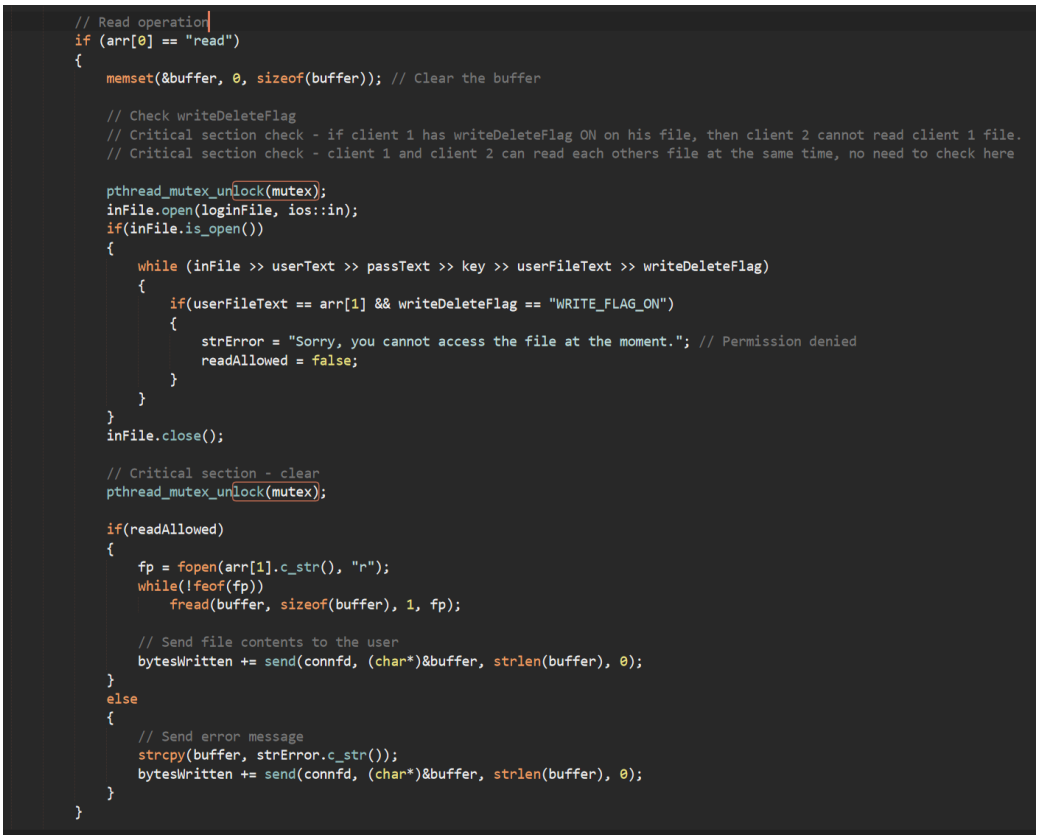
- Both the clients can read from the same file at the same time.

- **Read operation:**

If clinet1 request a read access for client 2 file, server first check the value of write flag for the requested file.
This flag can have one of the two states:
WRITE_FLAG_OFF = File is not being written by the owner at the moment.
WRITE_FLAG_ON = File is currently being written by the owner.

Server denies the read access if the write flag is ON for the requested file.

To allow asynchronous access to shared resources (such as file), the code is executed within the mutex lock system.



```
// Read operation
if (arr[0] == "read")
{
    memset(&buffer, 0, sizeof(buffer)); // Clear the buffer

    // Check writeDeleteFlag
    // Critical section check - if client 1 has writeDeleteFlag ON on his file, then client 2 cannot read client 1 file.
    // Critical section check - client 1 and client 2 can read each others file at the same time, no need to check here

    pthread_mutex_unlock(mutex);
    inFile.open(loginFile, ios::in);
    if(inFile.is_open())
    {
        while (inFile >> userText >> passText >> key >> userFileText >> writeDeleteFlag)
        {
            if(userFileText == arr[1] && writeDeleteFlag == "WRITE_FLAG_ON")
            {
                strError = "Sorry, you cannot access the file at the moment."; // Permission denied
                readAllowed = false;
            }
        }
    }
    inFile.close();

    // Critical section - clear
    pthread_mutex_unlock(mutex);

    if(readAllowed)
    {
        fp = fopen(arr[1].c_str(), "r");
        while(!feof(fp))
            fread(buffer, sizeof(buffer), 1, fp);

        // Send file contents to the user
        bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);
    }
    else
    {
        // Send error message
        strcpy(buffer, strError.c_str());
        bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);
    }
}
```

- **Write operation and Delete operation:**

When owner of the file requests for a write access, server update the value of write flag to WRITE_FLAG_ON in the loginfile. This file is then sent to other server for synchronization. Other clients cannot access the file when flag in ON.

Once user finished writing to the file, write flag is set back to WRITE_FLAG_OFF.

```
if(writeAllowed)
{
    // Update the value of writeDeleteFlag = WRITE_FLAG_ON
    // Send this file to other server for Synchronization (No read should be permitted to other users)
    fstream fFile, temp;
    string line;
    writeDeleteFlag = "WRITE_FLAG_ON";

    fFile.open("userInfoA.txt", ios::in | ios::app | ios::out);
    temp.open("temp.txt", ios::in | ios::app | ios::out);

    while(getline(fFile, line))
    {
        size_t found = line.find("user1");
        if(found == string::npos)
        {
            temp << line << endl;
        }
    }
    temp << user << " " << pass << " " << keyVal << " " << arr[1] << " " << writeDeleteFlag << endl;
    fFile.close();
    temp.close();
    remove("userInfoA.txt");
    rename("temp.txt", "userInfoA.txt");
```

```
// Ask user to enter file content
serverMsg = "Enter a file content";
memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
strcpy(buffer, serverMsg.c_str());
bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);
cout << "Awaiting client " << user << " response... for file content" << endl;

// Receive the file content and write to the file
memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
bytesRead += recv(connfd, (char*)&buffer, sizeof(buffer), 0);

fp = fopen(arr[1].c_str(), "w+b");
fwrite(buffer, sizeof(buffer), 1, fp);

// Acknowldge the write operation
memset(&buffer, 0, sizeof(buffer)); // Clear the buffer
serverMsg = "Write operation successful";
strcpy(buffer, serverMsg.c_str());
bytesWritten += send(connfd, (char*)&buffer, strlen(buffer), 0);

// Write complete
// Update the value of writeDeleteFlag = WRITE_FLAG_OFF
// Send this file to other server for Synchronization (Read operation can be permitted to other users)
writeDeleteFlag = "WRITE_FLAG_OFF";

fFile.open("userInfoA.txt", ios::in | ios::app | ios::out);
temp.open("temp.txt", ios::in | ios::app | ios::out);
```

5. Conclusion

In this paper, we study the concept of cloud computing and how it works. We implemented a prototype of the cloud environment based on TCP/IP protocol. We were successfully able to implement concurrent servers that listen to two client connections at same port and server them. Furthermore, we also employ client authentication, critical section check and file access permissions on the server side. Finally, to maintain data consistency at server side, we set up synchronization process between servers.

In future, we will implement a shared database for the concurrent servers and expand current design to support multiple clients (making the design more scalable).

REFERENCES

[1] G. Motta, N. Sfondrini and D. Sacco, "Cloud Computing: An Architectural and Technological Overview," *2012 International Joint Conference on Service Sciences*, Shanghai, 2012, pp. 23-27.doi: 10.1109/IJCSS.2012.37.

[2] Nazir M, *"Cloud Computing: Overview & Current Research Challenges," 2012 IOSR Journal of Computer Engineering (IOSR-JCE)"*, Volume 8, pp. 14-22.

[3] S. Namasudra, P. Roy and B. Balusamy, "Cloud Computing: Fundamentals and Research Issues," *2017 Second International Conference on Recent Trends and Challenges in ComputationalModels(ICRTCCM)*,Tindivanam,2017,pp.7-12.doi: 0.1109/ICRTCCM.2017.49

[4] "All about Sockets". Retrieved from http://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

[5] "Socket Programming". Retrieved from http://www.cs.dartmouth.edu/~campbell/cs50/socketprogramming.html

[6] "Critical Section in Synchronization". Retrieved from http://www.geeksforgeeks.org/g-fact-70

[7] "Client/server socket programs: Concurrent server socket programs" Retrieved from https://www.ibm.com/support/knowledgecenter/SSLTBW_2.1.0/com.ibm.zos.v2r1.hala001/orgcssocpcos.htm

[8] "Inter Process Communication (IPC)" retrieved from https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_tutorial.pdf

[9] "Multi-Process Programming in C retrieved from" https://home.deib.polimi.it/fornacia/lib/exe/fetch.php?media=teaching:aos:2016:aos201617_multiprocess_programming_updated20161223.pdf