Report 1
BI12-409 Hồ Công Thành
March 2024

1. Design protocol

The predefined file transfer protocol utilizes a message structure with headers for packet type, sequence number (for order), and potentially file size (during handshake). The server listens for connections and acknowledges them. The client initiates a connection and sends a message containing the filename (and size if needed) to initiate the handshake. File transfer occurs with the client sending sequenced data packets, and the server acknowledging received packets.
Error handling can be implemented through timeouts and re-transmission requests. The procedure of sending and receiving files repeats until no more files need to be loaded on the server.

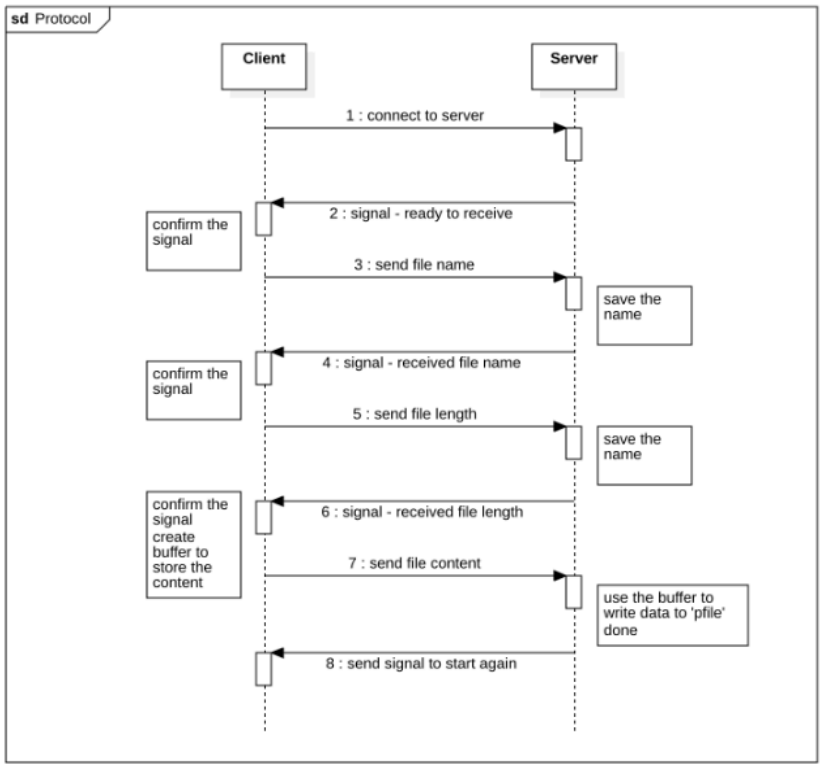

Figure 1: Protocol

2. Organizing system

Like in the protocol, the system has one server, client, and socket behavior utilizing the handshake behavior. On a specified port, the server waits for incoming connections. A handshake takes place to verify the connection and exchange the filename and sometimes the size of the file after a client connects. The server will send a signal to confirm the handshake if the client verifies the file transfer can be established. Subsequently, the client sends the file contents in chunks inside of predefined message packets. During the handshake, these packets might contain headers for the data type, sequence number, and sometimes file size. In addition to the acknowledgment packets that are received, the server also implements error-handling techniques like timeouts and re-transmissions. If successful, the server will send a confirmation signal to the client. A last server acknowledgment precedes the connection being closed, and a defined" end of transfer" message indicates completion. This organization provides basic error correction capabilities together with orderly and dependable data transport.

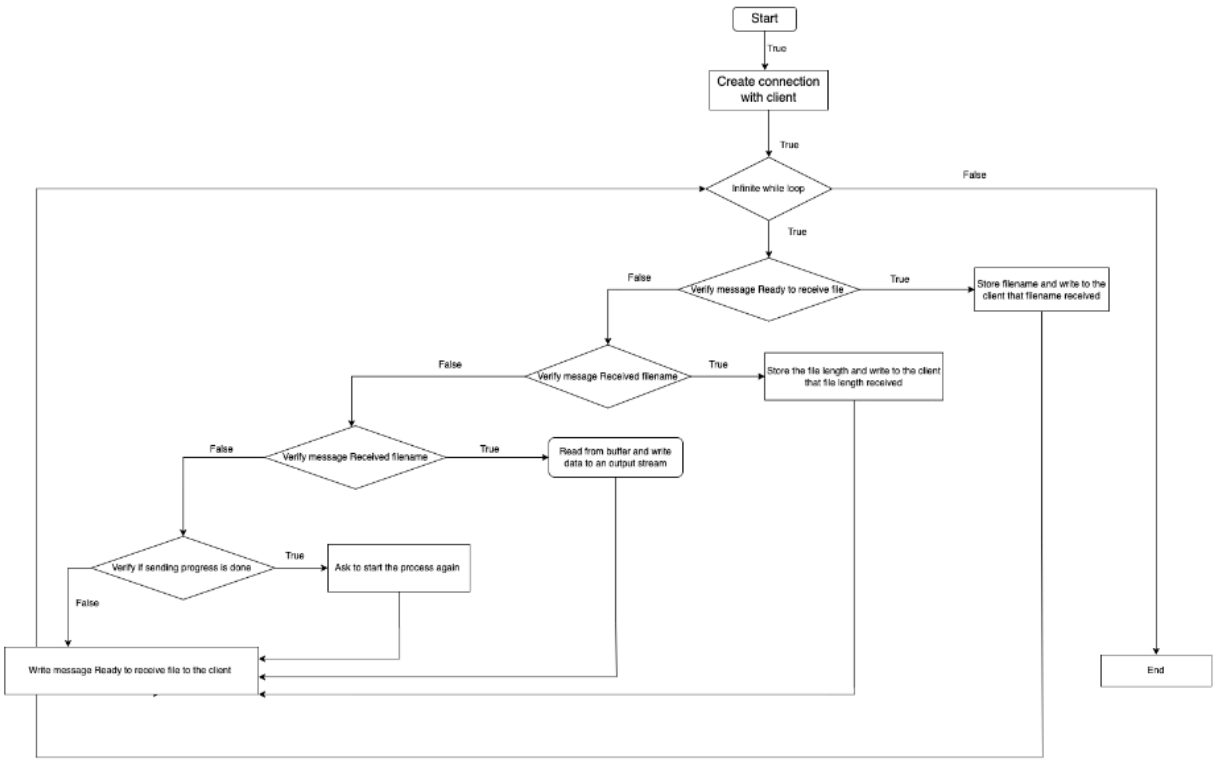

Figure 2: Server

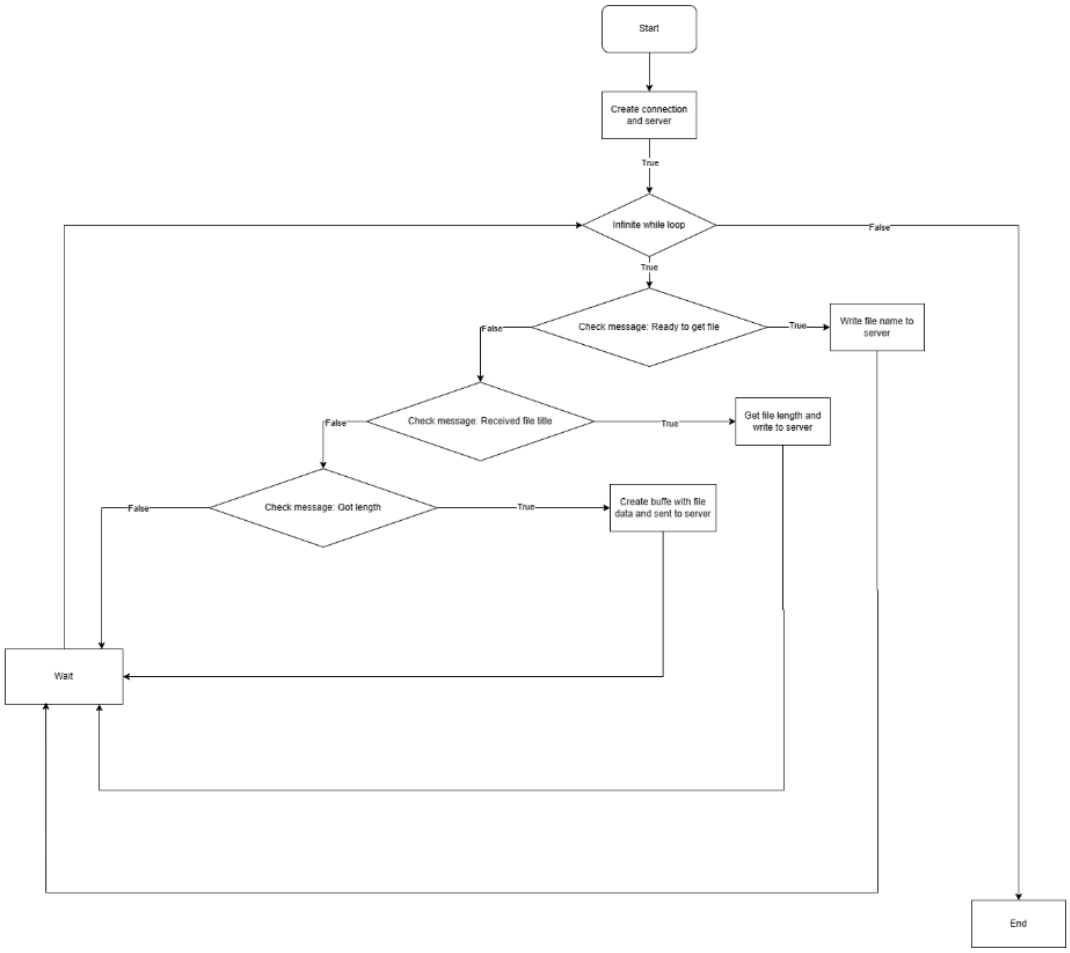

Figure 3: Client

3. Implementation

3.1 Server

Here are some of the code snippets for the server.

```
// Bind socket to address
if (bind(server_socket, (struct sockaddr*)&server_address,
sizeof(server_address)) == -1) {
  std::cerr << "Error binding socket" << std::endl;
```

```
  close(server_socket);
  return 1;
}
```

This code binds the server socket to the configured server address (server address) using the bind() system call. The bind() function associates the socket with a specific IP address and port number on the server. If the binding fails, an error message is printed, and the program exits.

```
// Listen for incoming connections
if (listen(server_socket, 1) == -1) {
  std::cerr << "Error listening for connections" << std::endl;
  close(server_socket);
  return 1;
}
```

After binding the socket, the server calls listen() to start listening for incoming client connections. The second argument (1) specifies the maximum length of the queue of pending connections. If the listen() call fails, an error message is printed, and the program exits.

```
// Accept a connection
int client_socket;
struct sockaddr_in client_address;
socklen_t client_address_size = sizeof(client_address);
client_socket = accept(server_socket, (struct sockaddr*)&client_address,
&client_address_size);
if (client_socket == -1) {
  std::cerr << "Error accepting connection" << std::endl;
  close(server_socket);
  return 1;
}
```

The server uses the `accept()` system call to accept an incoming client connection. The `accept()` function blocks until a client connects, and it returns a new socket descriptor (client socket) for communicating with the client. The client's address information is stored in the client address structure. If the `accept()` call fails, an error message is printed, and the program exits.

3.2. Client

Here are some of the code snippets for the client.

```
// Connect to the server
if (connect(client_socket,
(struct sockaddr*)&server_address, sizeof(server_address)) == -1) {
  std::cerr << "Error connecting to server" << std::endl;
  close(client_socket);
  return 1;
}
```

The client calls the `connect()` system call to establish a connection with the server. The `connect()` function attempts to make a connection on the specified socket (client socket) to the server address (server address). If the connection fails, an error message is printed, and the program exits.

```
// Send the filename to the server
if (send(client_socket, filename, strlen(filename), 0) == -1) {
  std::cerr << "Error sending filename" << std::endl;
  close(client_socket);
  return 1;
}
```

After establishing the connection, the client sends the filename to the server using the `send()` system call. The `send()` function writes the filename (including the null-terminator) to the socket (client socket). If the `send()` call fails, an error message is printed, and the program exits.

```
// Send the file data to the server
char buffer[1024];
while (infile.read(buffer, sizeof(buffer)).gcount() > 0) {
  if (send(client_socket, buffer, infile.gcount(), 0) == -1) {
    std::cerr << "Error sending data" << std::endl;
    infile.close();
    close(client_socket);
    return 1;
  }
}
```

The client reads the file data in chunks of 1024 bytes using `std::ifstream::read()` and sends it to the server using the `send()` system call. The loop continues until the end of the file is reached (infile.gcount() ¿ 0). If the `send()` call fails, an error message is printed, and the program exits after closing the file and socket