Socket Programming

学号： N/A
姓名： N/A
专业： 计算机科学与技术

1 Introduction

1.1 Features

This is a simple socket programming project that implements a P2P file transfer system. The system consists of a server and multiple clients. The server is responsible for managing the clients and the files they share. The clients can download files from each other, if their servers are online. The project is implemented in C++ and uses the Windows/Posix socket API for network communication.

The features of the system include:

- The user can use command-line interface to interact with the system.
- The server can handle multiple active clients simultaneously.
- The client may connect to multiple servers at the same time.
- The connection supports both IPv4 and IPv6.
- The project can be compiled and run on both Windows and Posix systems (Linux, macOS).

1.2 Usage

1.2.1 Commands

The client program can be run with the following command:

- `client-start`
  Start the client program.
- `client-stop`
  Stop the client program.
- `client-connect server=<ip> port=<port> domain=ipv4|ipv6`
  Connect to a server.
- `client-disconnect server=<ip> port=<port>`
  Disconnect from a specific server.

-
  ```
  client-request server=<ip> port=<port>
                 remote=</file/path/on/server>
                 local=</file/path/on/client>
  ```
  Request a file from a server and save it to the local path.

The server program can be run with the following command:

- `server-start`
  Start the server program.

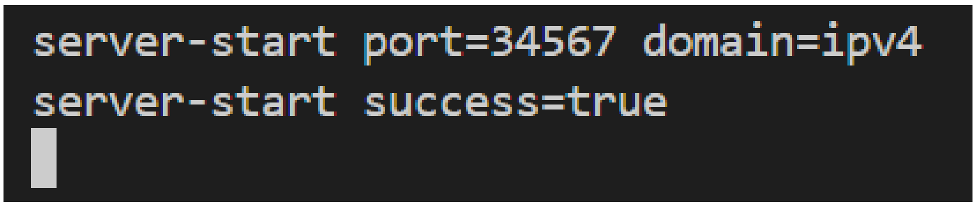
- `server-stop`
  Stop the server program.

Other commands include:
- `help`
  Show the help message.
- `exit`
  Exit the program.

1.2.2 Example

Here we start two hosts A and B on the same machine:

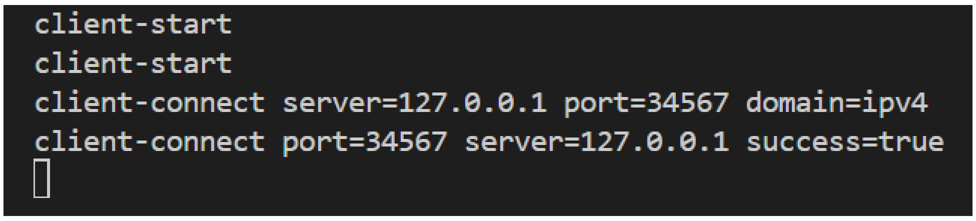
1. We first start the server on host A by running `server-start`.



```
server-start port=34567 domain=ipv4
server-start success=true
```

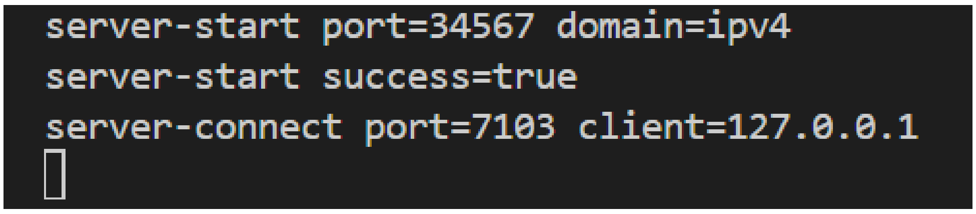
Figure 1: Start server on host A.

2. Then we start the client on host B by running `client-start` and connect to the server on host A by running `client-connect`:



```
client-start
client-start
client-connect server=127.0.0.1 port=34567 domain=ipv4
client-connect port=34567 server=127.0.0.1 success=true
```

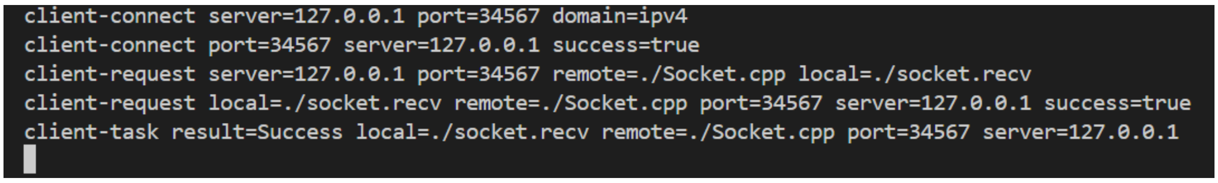
Figure 2: Start client on host B and connect to server on host A.



```
server-start port=34567 domain=ipv4
server-start success=true
server-connect port=7103 client=127.0.0.1
```

Figure 3: The server receives the connection from the client.

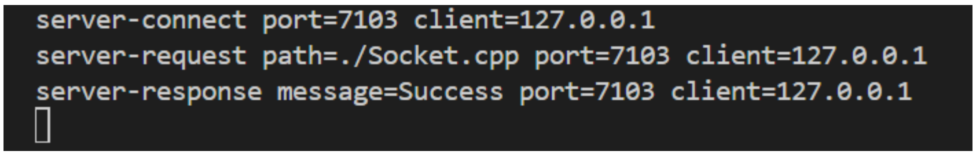
3. We can request a file `Socket.cpp` from the server on host A by running `client-request`, and save it to the local path:



```
client-connect server=127.0.0.1 port=34567 domain=ipv4
client-connect port=34567 server=127.0.0.1 success=true
client-request server=127.0.0.1 port=34567 remote=./Socket.cpp local=./socket.recv
client-request local=./socket.recv remote=./Socket.cpp port=34567 server=127.0.0.1 success=true
client-task result=Success local=./socket.recv remote=./Socket.cpp port=34567 server=127.0.0.1
```

Figure 4: Request a file from the server.

Here the `client-request` echo informs the user that the request is successfully sent to the server, and the `client-task` echo informs the user that the file is successfully received and saved to the local path.

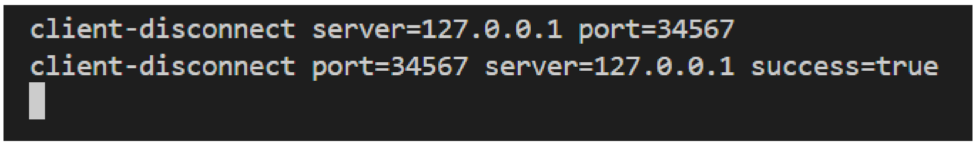


```
server-connect port=7103 client=127.0.0.1
server-request path=./Socket.cpp port=7103 client=127.0.0.1
server-response message=Success port=7103 client=127.0.0.1
```

Figure 5: The server receives the request and sends the file to the client.

The server also logs the request and response in the console.

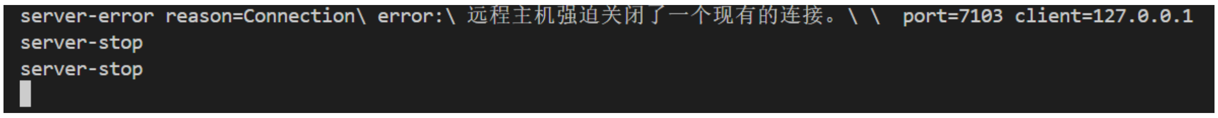
4. We then disconnect from the server on host A by running `client-disconnect`:



```
client-disconnect server=127.0.0.1 port=34567
client-disconnect port=34567 server=127.0.0.1 success=true
```

Figure 6: Disconnect from the server.

Then we can see that the server logs the unexpected disconnection in the console. Here the backslashes is used to escape the whitespaces in the message. Then we manually stop the server by running `server-stop`.



```
server-error reason=Connection\ error:\ 远程主机强迫关闭了一个现有的连接。\ \  port=7103 client=127.0.0.1
server-stop
server-stop
```

Figure 7: The server logs the unexpected disconnection.

1.3 Compilation

The project can be compiled with the following commands:

- `make BUILDTYPE=debug`
  Compile the project in debug mode.
- `make BUILDTYPE=release`
  Compile the project in release mode.

If you want to compile the project on Windows, you need to install the MinGW compiler and substitute the `make` command with `mingw32-make`.

The makefile itself is compatible with both platforms, and you can compile the project on Windows and Posix systems without any modification.

Be aware that the project uses the C++20 standard, and you need to upgrade to Ubuntu 24.04 LTS to have the latest GCC compiler that supports C++20.

The project is garanteed to compile and run on the following platforms:
- Windows 11 with gcc version 14.2.0 (x86_64-posix-seh-rev0, Built by MinGW-Builds project)
- WSL2 Ubuntu 24.04.1 LTS with gcc version 13.2.0 (Ubuntu 13.2.0-23ubuntu4)

Using other platforms or compilers may cause unexpected errors.

2 Implementation

2.1 Socket

The `Socket` class provides a cross-platform wrapper for the Windows/Posix socket API. It supports both IPv4 and IPv6, both TCP and UDP, and both blocking and non-blocking receiving.

The `Socket` class is provides the following methods:
- `Socket(Domain, Type)`
  Create a socket with the specified domain (IPv4/IPv6) and type (TCP/UDP). The class cannot be copy-constructed or copy-assigned to prevent resource leaks.
- `~Socket()`
  Deconstructor will automatically close the socket.
- `connect(IP, Port)`
  Connect to a server with the specified IP and port.
- `bind(Port)`
  Bind the socket to a specific IP and port.
- `listen()`
  Start listening for incoming connections.

- `accept(block)`
  Accept an incoming connection, either blocking or non-blocking.
- `send(data)`
  Send data to the connected server, if the socket uses TCP.
- `send(data, IP, Port)`
  Send data to a specific IP and port, if the socket uses UDP.
- `recv(block)`
  Receive data from the connected server, either blocking or non-blocking, if the socket uses TCP.
- `recv(block, IP, Port)`
  Receive data from a specific IP and port, either blocking or non-blocking, if the socket uses UDP.

All functions, except for the deconstructor, will throw an exception if any error occurs, instead of setting the `WSALastError` or `errno` variable. The exception will contain the error message fetched from the system, using the error code.

For Windows platform only, the `Socket` class maintains an instance count, so that `WSAStartup` and `WSACleanup` are called only once in the program.

By this design, we have completely wrapped the platform differences in the `Socket` class, and the user can use the class without worrying about the platform.

2.2 Protocols

The `Socket` class only provides the basic socket communication, yet we need to implement the protocols on top of the socket to achieve the file transfer system.

The protocols include:

- `GetProtocol`
  The protocol to request a file from the server.
- `FileProtocol`
  The protocol to send a file to the client.
- `ErrorProtocol`
  The protocol to send an error message to the client.

The protocols are all derived from the `Protocol` class. A protocol provides its own `serialize` method using virtual functions, and the `Protocol` class provides a static `deserialize` method to parse the received data into a protocol object. The object contains the type information and will be later cast to the specific protocol type.

Considering that the TCP connection is by nature a byte stream, it is possible to receive: 1) a part of the message, 2) exactly one message, or 3) multiple messages. To solve this problem, we add a header to the message, which contains the length of the message. The `Protocol` class provides a static `distill` method to extract exactly one message from the byte stream, if there is any.

This design can be easily extended to support more protocols, thus making the system more flexible and scalable.

2.3 Client

The client provides the following methods:

- `connect(Host)`
  Connect to a server with the specified host.
- `disconnect(Host)`
  Disconnect from a specific server.
- `request(Host, Task)`
  Request a file from a server and save it to the local path.

The client has these threads:

- The main thread is responsible for handling external function calls, and it will push the host or task to the corresponding queue.
- The monitor thread is responsible for starting and cleaning up the worker threads. When a task is pushed to the queue, or any worker thread is finished, the monitor thread will be notified with a condition variable. Then the monitor will clean up the finished worker threads, deal with the return states of them, and start new worker threads if there are any tasks in the queue.
- The worker threads are responsible for sending the request to the server, receiving the file, and saving it to the local path. The worker thread will save the `Success` message or any error message to the return state, and notify the monitor thread when it is finished.

The feedback from the client can be got from:
- When connecting, disconnecting or requesting a file, the client will throw an exception if any error occurs.
- When a task is finished, the client provides a callback function to notify the user. The callback function will be called with the task information and the return state.

2.4 Server

The server provides the following methods:
- `start()`
  Start the server and listen for incoming connections.
- `stop()`
  Stop the server and close all connections.

The server has these threads:
- The main thread is responsible for handling external function calls `start` and `stop`.
- The listener thread is responsible for accepting incoming connections. It will start a new worker thread for each connection. The worker thread will be detached, as we do not need its return state.
- The worker threads are responsible for receiving the request from the client, sending the file, and logging the request and response. The worker thread will not close the connection after sending the file, as the client may request more files in the same connection.

The feedback from the server can be got from:
- The server provides callback functions for connecting, disconnecting, requesting, responding, client error, and server error. The callback functions will be called with the connection information and the return state.

2.5 User Interface

The user interface provides a command-line interface for the user to interact with the system. It will parse the input command and call the corresponding function in the client or server. It catches all exceptions and prints the error message to the console.

3 Experiences and Challenges

3.1 Cross-platform

It is challenging to implement a cross-platform project, as the Windows and Posix socket API are quite different.

- The Windows API uses `WSAStartup` and `WSACleanup` to initialize and clean up the socket, while the Posix API do not need such operations.
- The Windows API uses `SOCKET (unsigned long long)` as the socket type, while the Posix API uses `int`.
- The Windows API uses `WSAGetLastError` to get the error code, while the Posix API uses `errno`.
- The Windows API indicates an error by returning `INVALID_SOCKET (ull max)`, while the Posix API returns `-1`.
- The Windows API and the Posix API have different mechanisms for setting the socket to non-blocking mode.

3.2 Non-blocking I/O

The project needs to use non-blocking I/O to gracefully terminate the client and server. When either stops, the `mRunning` flag will be set to false. If the worker uses blocking I/O, it will not be able to check the flag in time and thus not able to terminate.

The Posix API provides the `fcntl` function to set the socket to non-blocking mode, while the Windows API provides the `ioctlsocket` function. It is tricky that the `recv` function returns an error if the socket is set to non-blocking mode and there is no data to receive, instead of saying 0 bytes was received. We need to check the error code and handle it properly.

3.3 Multi-threading

The project uses multi-threading to handle multiple clients simultaneously. Mutex and condition variable and thread are used to synchronize the threads. It is challenging to design the monitor model, and let the program gracefully stops without crashing.

It is worth mentioning that debugging multi-threading programs is quite difficult. Not all errors can be reproduced because of the randomness of timing. We need to use the `std::this_thread::sleep_for` function to simulate the delay and increase the probability of error occurrence.

3.4 Exception Handling

Using exceptions rather than error codes is a good practice in C++ programming. However, it has some drawbacks:

- In deconstructors, we cannot throw exceptions, as it immediately crashes the program, or at least causes memory leaks.
- In multi-threading programs, we cannot throw exceptions across threads. We have to handle the exceptions thread-locally and pass the error message to the main thread using a return state.
- In the interface, we need to handle different kinds of exceptions and print the error message to the console.
- The `try-catch` blocks are rather ugly and make the code less readable.

4 Conclusion

The project implements a P2P file transfer system using the Windows/Posix socket API. This project allows the user to interact with the system using a command-line interface. The project is cross-platform and multi-threaded, and it supports both IPv4 and IPv6 connections.

In this project, we have learned how to use the Windows/Posix socket API, how to program in a cross-platform way, how to use mutex and condition variable to synchronize threads, and how to use exceptions to handle errors.

The project is not perfect, and there are still some improvements to be made:

- The project does not support browsing files on the server, uploading files or chunked transfer. Yet these features can be easily added by extending the protocols.
- The project does not support encryption or authentication. We need further study to implement these features.
- The project does not support the GUI interface. More time is needed to provide a user-friendly interface. As the project runs on the command line, it should be compatible with all front-end frameworks like QT, electron or python, etc.

In conclusion, the project is a good start for socket programming, and it provides a solid foundation for further study in network programming. I would like to thank for the tutorials provided by the assignment script, the stackoverflow community, and TAs for their help. Without their help, I would not be able to finish the project.