Computer Networks Lab#1: Chat Room Project

1 Introduction

This is the first lab for COMP461205: Computer Networks. In this project, you will implement a chat room by completing four stages:

1. In Stage 1, you will learn how to perform network programming using socket interfaces. You will implement a well-encapsulated, connection-oriented `TCPSocket` by completing several simple member functions.
2. In Stage 2, you will understand the client-server programming model and learn how to use mutexes properly in multi-threaded concurrent programming to implement a server.
3. In Stage 3, you will gain a deeper understanding of multi-threading synchronization methods and implement a message queue.
4. In Stage 4, you will learn about TCP packet sticking issues, and you will need to implement a codec to separate each message correctly.

In the chat room, any message sent by one person will be received by all other participants in the chat room. This allows everyone in the chat room to communicate with each other.

Your task is not to implement the entire project. Instead, you only need to write a small portion of the C++ code. The quality of this code will reflect your understanding of the lab.

2 Important Notes

Independent Work: The code you submit must be written by you. You are allowed to discuss with your classmates, but you cannot view their code or show your code to them. If you discuss the implementation of any stage with other students, please indicate this in the comments within your code.

Operating System Requirements: Your code will be tested on Ubuntu 22.04 LTS with `g++` 11.4.0, so it is recommended that you complete the lab in this environment. However, you may also use other Linux distributions (if Ubuntu, version 18.04 or above is recommended) or macOS. If your computer uses Windows, you can install a virtual machine to perform the lab.

Using CMake for Project Build Management: Ensure that the CMake version on your system is 3.8 or higher to support the C++17 standard. You can use either GCC or Clang for compilation.

Using Git for Version Control: It is recommended to use Git for managing your project, as this will help you track progress and maintain a history of your code. However, please be aware that if you host your code on a public platform (such as GitHub), ensure your repository is private.

3 A Brief Overview

Before diving into the detailed tasks, it's a good idea to have a rough understanding of the overall project. You need to carefully consider how participants in the chat room establish connections and communicate. Once you've decided on a solution, you can then focus on the specific implementation methods to ensure the project functions correctly.

3.1 Network Topology

In this lab, the TCP protocol will be used as the transport protocol. This means that if process a and process b want to communicate directly, a connection must first be established between them. For ease of discussion, we treat each process in the chat room as a node and each direct connection as an edge. This allows us to abstract the chat room application as an undirected graph.

Assume there are n processes in the chat room. We need to design a topology that ensures smooth application performance while minimizing resource usage. One simple idea is to establish a connection between every pair of nodes, forming a complete graph. However, this would require $O(n^2)$ edges and consume excessive resources; for example, adding just one more node would necessitate $O(n)$ additional edges.

The approach you will adopt is to introduce a special node s, where the remaining n nodes each establish a connection with s and communicate with other nodes indirectly through it. This reduces the number of edges to $O(n)$ and reduces the cost of dynamically adding or removing nodes. Figure 1 is a comparison of topologies. Typically, s is referred to as the server, while the other nodes are called clients, and this model is known as the client-server model. In this model, the server passively handles client requests and provides services to them. In the chatroom, when client a wants to send a message, it sends the message to the server. The server, upon receiving the message, forwards it to all other clients, allowing everyone to receive a's message.

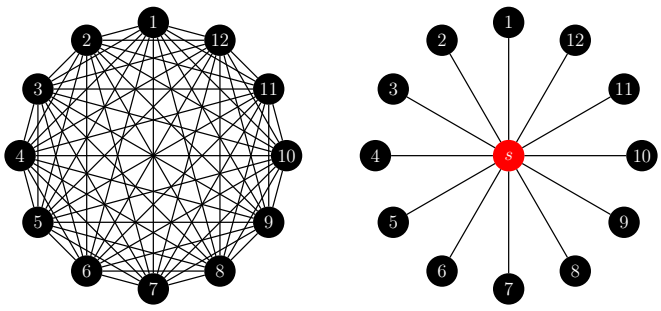

Figure 1: Comparison of Topologies. The left side shows a complete graph, while the right side shows a star topology with an additional node s.

3.2 Socket

A socket, or network socket, provides a method for network communication. The application layer can use the services of the transport layer, such as the TCP protocol, via the API provided by sockets. Refer to Section 6.1.3 in Chapter 6 of the textbook to see how two processes in a network establish connections and communicate.

In Linux/macOS, everything is treated as a file, and a socket is no exception. When the `socket()` function is successfully called, it returns a file descriptor, denoted as fd_s, which is essentially an index for a file[1]. After a connection request is accepted via the `accept()` function, another file descriptor, fd_a, is returned. By reading from and writing to fd_a, you can achieve information transmission. Meanwhile, `accept()` can still be called on fd_s, as they represent two different sockets.

Assume process a establishes a reliable TCP connection with another process b in the network using the socket fd_a, while process b has a corresponding socket fd_b. You can think of the connection between them as a two-way pipe, with one end being fd_a and the other being fd_b. When process a calls `send()` to transmit some bytes, the operating system simply copies the bytes into the send buffer in the kernel. In blocking mode, the process is blocked until the message is successfully copied. When and how the data is sent, and in what segment size, are handled by TCP. Similarly, when calling `recv()`, it merely checks the receive buffer in the kernel. If the buffer is empty, the process blocks (in blocking mode); otherwise, it reads bytes from the buffer. See Figure 2.

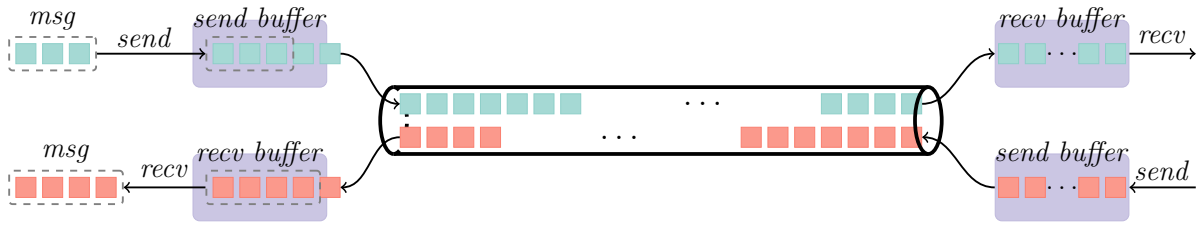

Figure 2: Communication flow between two ends using socket programming.

3.3 Concurrent Programming

Since the server needs to interact with n clients, there are n logical control flows. To respond to asynchronous client requests, we need an application-level concurrency technique to build a concurrent program. Several methods can achieve concurrent programming, such as processes, I/O multiplexing, and threads. In this project, we will use multithreading to support concurrency. You don't need to worry about creating or destroying threads, as that has already been handled for you. Instead, you will focus on another important issue: thread synchronization. Since different threads share variables, such as global variables, simultaneous access to the same memory address can lead to synchronization errors.

[1]In the following context, the term "file descriptor" refers to the file it points to, unless otherwise stated.

You will use the `mutex` class, available since C++11, to implement a mutual exclusion lock for thread synchronization.

4 Getting Started

Please ensure the following:

- The lab environment is Linux/macOS. If you don't have a Linux/macOS system, it is recommended to use a VM for this lab.
- The compiler supports C++17. You can check the `gcc` version by running `gcc --version`; `gcc 7` or above fully supports C++17.
- CMake version is 3.8 or higher.

Next, follow these commands in sequence to fetch and build the starter code:

1. Run `git clone https://github.com/KingSiong/chat-room.git` to download the source code prepared for this lab.
2. Enter the lab directory: `cd chat-room`.
3. Read the `CMakeLists.txt` file and generate the corresponding build files under `build/`: `cmake -S . -B build`.
4. Enter the `build/` directory: `cd build`.
5. Build the project according to the `Makefile`: `make`.

5 Chat Room

This lab involves implementing application-level concurrency using multithreading and building a network application–a chat room–using socket programming. In this section, you will first understand the overall structure and logic of the code. Then, you only need to focus on implementing a few member functions of several encapsulated classes, which form the core parts of the chat room.

The project's file structure is shown in Figure 3. In the `client/` directory, you'll find the `Client` class and some utility functions. Similarly, the `server/` directory contains the `Server` class, and `utils/` holds useful functions and utility classes required for the project. The `demo/` directory contains applications implementation, and `tests/` includes a series of test programs. Please do **NOT** modify the content in `tests/`.

In the chat room application, the client will eventually create two peer threads for sending and receiving data (see `client/utils.hh`). On the server side, a sending thread and several receiving threads will be created to handle messages from each client (see `server/utils.hh`).

In this lab, you don't need to worry about the overall logic; you should focus on an object-oriented approach. You will go through four stages, each requiring you to write

only a small piece of C++ code (with a total of less than 200 lines) to implement some interfaces.

```
chat-room
├── CMakeLists.txt
├── client
│   ├── client.cc
│   ├── client.hh
│   └── utils.hh
├── demo
│   ├── ...
├── server
│   ├── server.cc
│   ├── server.hh
│   └── utils.hh
├── tests
│   ├── ...
└── utils
    ├── codec.cc
    ├── codec.hh
    ├── common.hh
    ├── msg_queue.cc
    ├── msg_queue.hh
    ├── tcp_socket.cc
    ├── tcp_socket.hh
    └── thread_pool.hpp
```

Figure 3: File structure of `chat-room`.

Warning: This lab will enable `-O2` optimization. Please follow C++ standards when writing code and avoid undefined behavior, such as non-void functions without return values. Otherwise, it may lead to unexpected errors or crashes.

5.1 Stage 1: TCP Socket

In the first stage, you will need to complete the member functions of a `TCPSocket` class that provides a full-duplex, reliable stream based on TCP. To achieve this, you need to understand several system call functions. You can run `man 2 sys-call-name` for more information or refer to `https://man7.org/linux/man-pages/dir_section_2.html`.

- `socket` creates a communication endpoint, returning a file descriptor that can be used to establish connections or transmit messages.

  ```
  int socket(int domain, int type, int protocol);
  ```

  the `domain` argument specifies the protocol family or communication domain that determines the addressing scheme used by the socket. We use `AF_INET` for communication over IPv4. `type` specifies the communication semantics. We use `SOCK_STREAM` to provide sequenced, reliable, two-way, connection-based byte streams. And we use `IPPROTO_TCP` as `protocol` to support this type.

- `setsockopt` is used to configure various options at different levels for a socket. It allows you to control socket behavior, such as enabling reusable addresses.

```
int setsockopt(int sockfd, int level, int optname,
        const void *optval, socklen_t optlen);
```

`sockfd` indicates the file descriptor of the socket you want to configure. You only need to manipulate options at the sockets API level, and `level` is specified as `SOL_SOCKET`. `optname` is the specific option you want to set (e.g., `SO_REUSEADDR`, `SO_RCVBUF`). The arguments `optval` and `optlen` are used to access option values.

- `bind` is used to bind a name to a socket. This is particularly important for server applications, where the server needs to bind its socket to a known address or port so that clients know how to connect to it.

  ```
  int bind(int sockfd, const struct sockaddr *addr,
          socklen_t addrlen);
  ```

  `sockaddr` is a generic structure used to represent a socket address. It's often used for casting to more specific address structures like `sockaddr_in` or `sockaddr_un`. Here you should use `sockaddr_in` for IPv4 addresses, whose structure is defined as follows:

  ```
  struct sockaddr_in {
      sa_family_t     sin_family;     /* AF_INET */
      in_port_t       sin_port;       /* Port number */
      struct in_addr  sin_addr;       /* IPv4 address */
  };
  ```

- `listen` is used by a server to indicate that it is willing to accept incoming connection requests on a socket.

  ```
  int listen(int sockfd, int backlog);
  ```

  The `backlog` argument defines the maximum number of pending connections that can be queued up before the kernel starts rejecting new connections. This also indicates that the `listen()` call is non-blocking.

- `accept` is used by a server to accept an incoming connection from a client. When a client tries to connect to a server, the server can call `accept()` to complete the connection process and obtain a new socket for communication with that client.

  ```
  int accept(int sockfd, struct sockaddr *_Nullable restrict addr,
          socklen_t *_Nullable restrict addrlen);
  ```

- `connect` is used in socket programming by a client to establish a connection with a server.

  ```
  int connect(int sockfd, const struct sockaddr *addr,
          socklen_t addrlen);
  ```

  `addr` is a pointer to a `sockaddr` structure that contains the server's address and port to which the client wants to connect.

- `send` is easy to understand, which allows you to transmit data over a connection-oriented (TCP) or connectionless (UDP) socket.

  ```
  ssize_t send(int sockfd, const void *buf, size_t len, int flags);
  ```

- `recv` is used to receive data from a socket.

  ```
  ssize_t recv(int sockfd, void *buf, size_t len, int flags);
  ```

  By default, both `send()` and `recv()` operate in blocking mode. Please keep this configuration for this lab.

In Stage 1, you only need to modify `utils/tcp_socket.cc`. The sections you need to complete will have `TODO` comments. Once you have implemented the `TCPSocket` class, run `make`, and then run `./A` and `./B` in separate terminals. You will see an end-to-end interactive chat system. Afterward, run `make stage1` to automatically test your implementation. If everything is working correctly, you will see the following:

```
[100%] Testing Stage #1: TCPSocket...
Test project /home/chat-room/build
    Start 1: t_socket_member_function
1/2 Test #1: t_socket_member_function .........   Passed    0.44 sec
    Start 2: t_socket_p2p
2/2 Test #2: t_socket_p2p .....................   Passed    0.28 sec

100% tests passed, 0 tests failed out of 2
```

5.2 Stage 2: Server

Please open `server/server.cc`, where you need to implement 4 interfaces for `Server`:

```
// send ``msg'' to some clients: ``client_socks''
void send(std::vector<TCPSocketPtr> &client_socks,
        const std::string &msg);

// send ``msg'' to all clients connected to this server
void send_all(const std::string &msg);

// add the client ``client_sock'' to ``_client_socks''
void add_client(TCPSocketPtr client_sock);

// delete the client ``client_sock'' from ``_client_socks''
// and shutdown the corresponding socket
void del_client(TCPSocketPtr client_sock);
```

You need to manage the clients connecting to the server using a `std::set`, which might seem straightforward. However, you should be aware that many functions of the `std::set` container are not thread-safe, meaning you need to handle thread synchronization issues. One solution is to use a mutex (mutual exclusion). C++11 introduced mutex-related classes, with all relevant functions provided in the `<mutex>` header file. `std::mutex` is the basic mutex class in C++11, and you can create a mutex by constructing a `std::mutex` object. Its member functions `lock()` and `unlock()` are used to lock and unlock the mutex, respectively. However, in practice, it is better not to call these member functions directly, as this requires calling `unlock()` at every critical section exit, including handling exceptions. Instead, C++11 provides a RAII mechanism through the template class `std::lock_guard` for managing mutexes.

In this lab, we recommend using `std::unique_lock`, which is more flexible compared to `std::lock_guard`. It allows for explicit locking and unlocking of `std::mutex` objects. If you are not yet familiar with how these classes work, consider writing one or two simple programs to help you understand them. After completing Stage 2, you can verify your implementation by running `make stage2`.

5.3 Stage 3: Message Queue

Consider the following issue: If multiple receiving threads on the server receive messages simultaneously, which message should the server process first? What happens to the remaining messages? This issue arises because message reception by different threads is asynchronous, and you cannot assume that a particular client's message will always arrive first. Therefore, messages should not be processed in a specific order based on assumptions.

Moreover, the speed at which messages are received is uncertain, so a buffer is needed to temporarily store a large number of messages that may arrive in a short time. Ideally, we would like to process these messages in the order they arrived. This means that the buffer should function as a first-in, first-out (FIFO) queue, known as a message queue, as shown in Figure 4.

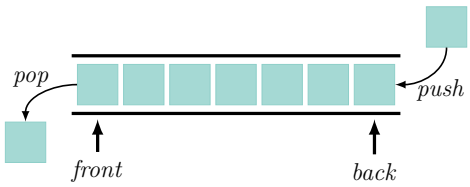

Figure 4: FIFO queue.

In the chatroom, there are n threads that may produce messages. Whenever a thread receives a message, it attempts to push the message to the back of the queue. There is also a thread responsible for processing and sending messages. Whenever this thread finds that the queue is not empty, it attempts to pop a message from the front of the queue and send it to each client. This is a classic producer-consumer problem, which can be solved using semaphores[2]. Simply using mutexes is insufficient for solving this problem because the consumer needs to check whether the queue is empty, and checking followed by locking are not indivisible atomic operations.

In this lab, you are dealing with a model that has n producers and a single consumer, and we assume that the message queue has an *infinite* length. C++11 provides a synchronization primitive, `std::condition_variable`, to help us solve this problem more easily. `std::condition_variable` has two important member functions: `wait()` and `notify_one()`. When a thread tries to acquire a lock on a mutex using `std::unique_lock` and calls `wait()`, the thread will block and release the mutex until another thread calls `notify_one()`, at which point the waiting thread is awakened and reacquires the mutex.

[2]For more information, refer to Section 12.5.4 of *Computer Systems: A Programmer's Perspective*.

Please implement the two functions in `utils/msg_queue.cc` using the methods mentioned above. Afterward, as in the previous stages, run `make stage3` to test your implementation.

5.4 Stage 4: Codec

Although TCP provides reliable data streaming, as you learned in Section 3.2, the `recv()` function simply copies a chunk of data from a section of memory called the receive buffer in the kernel. As a result, the received string might contain multiple messages concatenated together, or some messages might be split at unpredictable points, requiring multiple `recv()` calls before they are fully received. Run `./one-time_recv` and you will see the following output:

```
server sent a msg: hello!
server sent a msg: how are you?
server sent a msg: this is a test.
client received a msg: hello!how are you?this is a test.
```

In this program, the sending thread first sends `hello!`, `how are you?`, and `this is a test.` one by one, and then waits for the thread to complete before starting the receiving thread. As you can see, the three messages are "glued" together and received all at once.

To address this issue, you need to design an encoding and decoding scheme so that each message is encoded before sending, and the received string can be correctly decoded into the original messages regardless of message concatenation or truncation.

For the encoder, it takes a non-empty string as input and returns the encoded string. For the decoder, it is an instance that continuously works, attempting to decode each input string and returning a sequence of decoded messages. This means the current string might be incomplete and temporarily undecodable, but since TCP provides a reliable data stream, the incomplete part of the string along with subsequent input will eventually allow the original messages to be decoded. Formally, given k messages to be encoded, $m_1, m_2, \ldots, m_k \in \Sigma^+$, the encoder implements an injective function $f : \Sigma^+ \mapsto \Sigma^+$, resulting in k encoded strings $f(m_1), f(m_2), \ldots, f(m_k)$. There is a channel ch that takes a sequence of strings as input and outputs another sequence of strings. Specifically:

$$ch((s_1, s_2, \ldots, s_k)) = (t_1, t_2, \ldots, t_p)$$

which satisfies the condition $s_1 + s_2 + \ldots + s_k = t_1 + t_2 + \ldots + t_p$, where "+" denotes string concatenation. Let $(e_1, e_2, \ldots, e_p) = ch((f(m_1), f(m_2), \ldots, f(m_k)))$. The decoder will sequentially read e_i $(1 \leq i \leq p)$ and decode them back into $m_1, m_2, \ldots, m_k$.

To help you understand how the decoder works, you can think of it as a combination of a DFA (Deterministic Finite Automaton) $M = (Q, \Sigma, \delta, q_0, F)$ and an injective function $f^{-1} : \Sigma^+ \mapsto \Sigma^+$. In M, $F = \{q_0\}$, and f^{-1} is the inverse function of the encoding function f. When the decoder receives a string s, M processes each character of s one by one and transitions states. Each time M reaches an accepting state, it indicates that starting from the initial state q_0, given an input string w, M has returned to q_0. In this case, w is an accepted word. Let $w_1, w_2, \ldots, w_k$ be the words accepted by M after processing s. The decoder returns the sequence $(f^{-1}(w_1), f^{-1}(w_2), \ldots, f^{-1}(w_k))$.

Here is an example:

1. Suppose the messages sent are `hello` and `computernetwork`.
2. These messages are encoded as `hello#` and `computernetwork#`, respectively.
3. The received strings are `hell`, `o#computer`, and `network#`.

In this case:

1. Firstly the decoder receives `hell`, it cannot decode the original message, so it returns ().
2. Then the decoder receives `o#computer`, it returns the sequence (`hello`).
3. Finally, the decoder receives `network#`, it returns the sequence (`computernetwork`).

The following two pieces of C++ style pseudo-code illustrate how the encoder and decoder work.

- The encoder at the sender:

```
Codec encoder;
while ((string msg = gen()) != "") { // to generate a msg
    auto encoded_msg = encoder.encode(msg);
    send(encoded_msg);
}
```

- The decoder at the receiver:

```
Codec decoder;
while ((string str = recv()) != "") {
    auto msgs = decoder.decode(str);
    for (const auto &msg : msgs) {
        print(msg);
    }
}
```

The testing program will follow these conventions:

- **Multiple Sub-Tests**: Each test point will have several unrelated sub-tests.
- **String Length Constraints**: For the t-th sub-test, the i-th encoded string is denoted as $s_i^{(t)}$. It is guaranteed that the total length of encoded strings $\sum_i |s_i^{(t)}|$ will not exceed 10^6, and the combined total length across all test cases $\sum_t \sum_i |s_i^{(t)}|$ will not exceed 10^8.
- **Encoder Constraints**: For an encoder, the encoded string $f(s)$ for input s will be considered valid if $|f(s)| \leq \max\{2|s| + 1, |s| + 9\}$.
- **Decoder Constraints**: When implementing the decoder, you will not know which strings have been encoded by the encoder. Although encoding and decoding methods belong to the same class, they will be tested with separate instances.

- **Time Limit**: The time limit for execution is 10 seconds.
- **Character Set Size**: The character set size $|\Sigma|$ is 256, meaning all 256 possible byte values may appear in the test cases.
- **Bonus Test Point**: There is a bonus test point. If your encoding meets the constraint $|f(s)| \leq \lceil 3|s|/2 \rceil$, you will pass this test. Don't worry if you can't pass the bonus test case; it won't affect your final score for this lab.

Please open `utils/codec.hh` and `utils/codec.cc` to implement a codec. You can add any necessary member variables in `utils/codec.hh`. Test your implementation using `make stage4`.

Note: For each decoder instance, it will continuously process a complete data stream. You may need information from previous processing steps when handling a string at any given time. For example, since the string received by the decoder each time may not be enough to decode a complete message, part of it may need to wait for subsequent input. You will need to add some member variables to store the string that has not yet been fully decoded.

Tip: If you need some ideas for this stage, the textbook's Section 3.1.2 might be helpful.

5.5 Final Review and Execution

Congratulations on completing all the stages! You can test all the stages by running `make all-stage`. If everything is working correctly, you can start the server with `./server` and then log into the chat room by running `./client`. To allow people on your local network to join the chat room, you can modify the `DEFAULT_IP` in `demo/server_main.cc`.

6 Submission

- Compress `utils/tcp_socket.cc`, `server/server.cc`, `utils/msg_queue.cc`, `utils/codec.hh` , `utils/codec.cc`, and your report into a ZIP file and upload it.
- Submit a PDF report of approximately two pages, which should only include the final test results and your understanding, reflections, or issues encountered during each stage, along with the solutions.
- **Optional**: You can also include feedback or suggestions regarding the experiment in the report.