ThreadPool.hpp
Libraries used:

```
#include <vector> // data structure
#include <queue> // data structure
#include <thread> // for threading
#include <mutex> // synchronization
#include <condition_variable> // synchronization
#include <functional> // generic functions or lambda functions
```

The <thread> library provides support for multithreading.
It contains the class std::thread
and associated methods like join(), detach(), hardware_concurrency().
It is used to perform multiple operations simultaneously.

The <mutex> library provides support for thread synchronization.
It contains the classes std::mutex, std::timed_mutex, std::recursive_mutex,
and associated methods like lock(), unlock(), try_lock().
It is used to prevent race conditions, especially when multiple threads try
to access a shared resource simultaneously.

The <condition_variable> library provides thread synchronization based on conditions.
It contains the classes std::condition_variable, std::condition_variable_any,
and methods like wait(), notify_one(), notify_all().
It is used to suspend a thread's activity until a condition is met,
and also to check which threads are waiting and can resume execution.

ThreadPool class:

```
class ThreadPool {
    private:
        std::vector<std::thread> threads; // vector of threads managed by the pool
        std::queue<std::function<void()>> tasks; // task queue (functions) to execute
        std::mutex queueMtx; // mutex to synchronize access to the queue
        std::condition_variable condition; // condition variable to notify threads
        bool stop; // flag for stopping the pool

    public:
        ThreadPool(size_t threadsCount);
        void enqueue(std::function<void()> task); // method to add a task to the queue
        void waitForCompletion() {
            for (std::thread &thread : threads) {
                if (thread.joinable()) thread.join();
            }
        }
        ~ThreadPool();
};
```

ThreadPool.cpp
CONSTRUCTOR:
The first function is the constructor of the class.
The parameter threadsCounter specifies the number of threads to be created.
Its main function is to create a pool of N threads (N = threadsCounter).
For each thread, a lambda containing [this] is executed as part of the thread.
Each thread executes a while(true) loop in which:
- It waits until the queue has tasks or until the stop flag is triggered.
- If the stop flag is active and the task queue is empty, the thread terminates (executes return).

- If there are tasks in the queue, it extracts and executes the first task.

Synchronization:
- std::unique_lock<std::mutex>: Allows obtaining a lock on the queueMtx (mutex)
to safely access the task queue.
- condition.wait(): The threads remain "waiting" until the condition stop || !tasks.empty() is true.
- Queue pop: Once a task is obtained, the thread removes it from the queue and executes the function task().

ENQUEUE METHOD:
The function passed as a parameter has no parameters and does not return anything,
but it allows the job to be executed.
Adding a task to the queue:
- std::lock_guard<std::mutex>: Prevents concurrent access to the task queue.
When the lock is destroyed (out of the {} block), the mutex is released.
- tasks.push(task): Adds the new task to the queue.
- condition.notify_one(): Wakes up one of the threads that might be waiting on the condition condition.wait().

DESTRUCTOR:
Sets the stop flag to true, signaling to the threads to stop.
- condition.notify_all(): Wakes up all waiting threads so they can check the stop flag.
- When the thread.joinable() condition is true, the thread is stopped with thread.join(),
and the class destruction waits until then.

Thread Pool = a structure that manages a fixed group of reusable threads to execute multiple tasks.
Instead of "destroying" the threads and recreating them, they remain available and ready to execute new tasks.

Socket.hpp
Libraries:
#include <cstring>: used for string manipulation like in Consente
#include <sys/socket.h>: for socket management
#include <netinet/in.h>: for network structures (e.g., sockaddr_in)
#include <unistd.h>: for system functions (e.g., close())

Socket class:
```
class Socket {
    protected:
        int sockfd; // socket file descriptor
        struct sockaddr_in address; // structure containing address and port info

    public:
        Socket();
        virtual ~Socket();
        void bind(int port); // assigns the socket to a specific port
        void listen(int backlog = 5); // sets the socket to listen with a backlog of pending connections
        void closeSocket();
        void sendData(int clientSocket, const std::string& message); // sends data to the specified client
        std::string receiveData(int clientSocket, int bufferSize = 1024); // receives data from the clientSocket
        int getSocketfd() const {
            return sockfd;
        }
};
```

file descriptor: is a unique integer used as a handler to refer to the socket in subsequent operations.
It also allows the program to identify the socket and perform operations on it such as:
- send()
- recv()
- close()
- reading and writing

Socket.cpp
CONSTRUCTOR:
Creates a socket using the socket() function:
- AF_INET: specifies the use of the IPv4 protocol
- SOCK_STREAM: indicates the use of a TCP socket (connection-oriented)
- Specifies the default protocol, which is TCP

If the socket creation fails, the program will terminate with an error code, usually due to a lack of network resources.

DESTRUCTOR:
Closing the socket is essential to avoid wasting network resources.

BIND METHOD:
Configures the socket address:
- sin_family = AF_INET: Specifies the IPv4 protocol.
- sin_addr.s_addr = INADDR_ANY: Indicates that the server accepts connections from all network interfaces (localhost, local network, etc.).
- sin_port = htons(port): Converts the port number to big-endian format as required by network functions.

Then bind() is called to associate the socket with a specific IP address and port.

LISTEN METHOD:
Puts the socket in "listening" mode for incoming connections.
backlog: specifies the maximum length of the connection queue
= 5: it can handle up to 5 pending connections before rejecting others.

CLOSESOCKET METHOD:
Explicitly closes the socket.

SENDDATA METHOD:
This method sends a message to the client.
It accepts 2 parameters:
- clientSocket: the file descriptor of the client socket
- message: the string to send

It uses the send function to send the data to the client:
- message.c_str() converts the C++ string to a C string (array of characters).
- message.size() specifies the length of the message.
- The last parameter is 0 (default flag).

RECEIVEDATA METHOD:
This method receives data from the connected client.
It accepts 2 parameters:
- clientSocket: the file descriptor of the client socket
- bufferSize: the maximum size of data to receive

A buffer of bufferSize is created, and memset() is used to clear it.
It uses recv() to receive data, storing it in the buffer. If the data is larger than bufferSize,
only the first bytes up to bufferSize are taken.

Finally, it returns the number of bytes received:
- clientSocket: the file descriptor of the client socket to receive data from
- buffer: a character array (buffer) where the received data will be stored
- bufferSize: the maximum size (in bytes) of data to receive
- 0: flag (options) for recv() behavior (0 = default behavior).

ServerSocket.hpp
ServerSocket class:

```
class ServerSocket : public Socket {
    private:
        ThreadPool threadpool; // creates a thread pool object to manage it

    public:
        ServerSocket(int port, size_t threadsCount); // constructor initializes port and thread count
        int acceptConnection(); // method that accepts the client connection
        void handleClient(int clientSocket); // manages the client
        ThreadPool& getThreadPool() { // getter for the thread pool, inaccessible from main because it is private
            return threadpool;
        }
};
```

ServerSocket.cpp
CONSTRUCTOR:
Initializes the thread pool. The constructor accepts two parameters:
- port: the port where the server will listen for incoming connections
- threadsCounter: the number of threads to use for managing clients.
This is passed to the ThreadPool constructor to handle the thread pool for multithreading.
bind(): associates the socket with a specific port.
listen(5): the server starts listening for connections on the port with a backlog of 5 pending connections.

ACCEPTCONNECTION METHOD:
- int addrlen = sizeof(address): calculates the byte size of the address structure,
used to store the address of the client trying to connect to the server.
The addrlen variable indicates to the accept() function how much space is available to store the client's address.
The accept() function will populate this structure with the client's address (IP and port).

- int newSocket = accept(sockfd, (struct sockaddr*)&address, (socklen_t*)&addrlen): accepts an incoming connection on the server socket.
- sockfd: listening socket associated via bind().
- (struct sockaddr*)&address: casts the address structure to a generic sockaddr* type.
accept() accepts a pointer to sockaddr as a parameter to support both IPv4 and IPv6.
Inside it will contain the address of the client that has connected (IP and port).
- (socklen_t*)&addrlen: accept() needs to know the length of the address structure, used to determine how much space is available to store the client's address.
addrlen is updated with the size of the client's address.

The function returns a new socket (newSocket) (used primarily for communication with the newly connected client) that will be used as the identifier for the socket associated with the client connection.

HANDLECLIENT METHOD:
This function takes a clientSocket as input, which is the socket associated with the client that connected.

The server handles communication with the client on a separate thread.

ClientSocket.hpp
ClientSocket class:

```
class ClientSocket : public Socket {
    public:
        void connectToServer(const std::string& serverAddress, int port); // method to connect to the server
};
```

ClientSocket.cpp
Library:
#include <arpa/inet.h>: used to manipulate IP addresses, with the function inet_pton() converting an IP address from string to binary format.

CONNECTTOSERVER METHOD:
It accepts 2 parameters:
- const std::string& serverAddress: the server's address, i.e., the IP in string form that the client is trying to connect to
- int port: the server's port to connect to

address.sin_port = htons(port): htons() --> function that converts the port number to "network byte order",
so the bytes are sent correctly across the network.

inet_pton(AF_INET, serverAddress.c_str(), &address.sin_addr): Converts the server's IP address (in string form)
to a binary format that can be used by sockaddr_in.
The inet_pton function returns a positive value if the conversion is successful or a negative value (<= 0) if there is an error.

connect(sockfd, (struct sockaddr*)&address, sizeof(address)): Attempts to establish a connection with the server.
The sockfd socket descriptor is used to perform the connection, passing the address of the server and its length.