ThreadPool.hpp
Librerie usate:

```
#include <vector> // data struct
#include <queue> // data struct
#include <thread> // per il threading
#include <mutex> // sincronizzazzione
#include <condition_variable> // sincronizzazzione
#include <functional> // funzioni generiche o lambda function
```

La librearia <thread> fornisce il supporto al multithreading.
al suo interno è presente la classe std::thread
e i metodi associati come join(), detach(), hardware_concurrency()
viene usata per eseguire più operazione contemporaneamente.

La libreria <mutex> fornisce il supporto per la sincronizzazione tra thread.
al suo interno è presente la classe std::mutex, std::timed_muted, std::recursive_mutex
e i metodi associati come lock(), unlock(), try_lock()
Viene usata per evitare la race condition, specialmente quando diversi thread tentano
di accedere contemporaneamente ad una risorsa condivisa.

La librearia <condition_variable> fornisxce la sincronizzazione dei thread basata su condizioni
contiene le classi std::condition_variable, std::condition_variable_any,
e i metodi wait(), notify_one(), notify_all()
viene sfruttata per sospendere l'attività di un thread fino a quando una condizione non è soddisfatta,
ma anche per verificare quali sono i thread in attesa e che posso riprendere l'esecuzione

classe ThreadPool:

```
class ThreadPool{
    private:
        std::vector<std::thread> threads; // vettore dei thread gestiti dal pool
        std::queue<std::function<void()>> tasks; // coda di task (funzioni) da eseguire
        std::mutex queueMtx; // mutex per sincronizzare l'accesso alla coda
        std::condition_variable condition; // variabile di condizione per notificare i thread
        bool stop; // flag per la chiusura del pool
    public:
        ThreadPool(size_t threadsCount);
        void enqueue(std::function<void()> task); // meotdo per aggiungere una task alla coda
        void waitForCompletion() {
            for (std::thread &thread : threads) {
                if (thread.joinable()) thread.join();
            }
        }
        ~ThreadPool();
};
```

ThreadPool.cpp
COSTRUTTORE:
La prima funzione è il costruttuore della classe.
Il parametro threadsCounter specifica il numero di thread che devono essere creati.
La sua funzione prinicipale è quella di creare un pool di N thread (N = threadsCounter)

Per ogni thread viene eseguita una lambda contenente [this] che fa parte del thread.
Inoltre ogni thread esegue un while(true) in cui:
- Attende fino a quando nella coda non sono presenti task o fino a quando non viene attivato il flag

stop
- se il flag stop è attivo e la coda di task è vuota, il thread termina (esegue il return)
- se nella coda ci sono sono task, estrae il primo e lo esegue

Sincornizzazione:
- std::unique_lock<std::mutex>: Consente di ottenere un lock sul queueMtx (mutex) per accedere in sicurezza alla coda dei task.
- condition.wait(): I thread rimangono "in attesa" finché la condizione stop || !tasks.empty() non è vera
- Pop della coda: Una volta ottenuto un task, il thread lo rimuove dalla coda ed esegue la funzione task().

METODO ENQUEUE():
La funzione passata per parametro non presenta parametri e non ritorna nulla, ma bensì permette l'esecuzione del job

Aggiunta di una task alla coda:
- std::lock_guard<std::mutex>: Impedisce l'accesso concorrente alla coda dei task. Quando il lock viene distrutto (fuori dal blocco {}), il mutex viene rilasciato.
- tasks.push(task): Aggiunge il nuovo task alla coda.

- condition.notify_one(): Sveglia uno dei thread che potrebbe essere in attesa sulla condizione condition.wait().

DISTRUTTORE:
imposta la flag stop a true, segnalando ai thread che devono interrompersi.
- condition.notify_all(): Sveglia tutti i thread in attesa, così possono controllare il flag stop.
- QUando la condizione thread.joinable() è vera, interromper il thread con thread.join(), la distruzione della classe attende fino a quel momento.

Thread Pool = struttura che gestisce un gruppo fisso di thread riutilizzabili per eseguire più task. Invece che "distruggere" i thread e ricrearli, quest'ultimi restano utilizzabioli e pronti per eseguire nuove task.

Socket.hpp
Librerie:
#include <cstring>: usata oper la manipolazione delle stringhe come in Consente
#include <sys/socket.h>: per la gestione della socket
#include <netinet/in.h>: per le strutture di rete (esempio: sockaddr_in)
#include <unistd.h>: per l'uso di funzioni di sistma (esempio: close())

classe Socket:
```
class Socket {
    protected:
        int sockfd; // file descriptor del socket
        struct sockaddr_in address; // struttuira che conteiene l'indirizzo e le informazione sulla porta
    public:
        Socket();
        virtual ~Socket();
        void bind(int port); // assegna il socket a una sepcifica porta
        void listen(int backlog = 5); // imposta il socket in modalità di ascolto con una coda di donnessioni in attesa
        void closeSocket();
```

```
        void sendData(int clientSocket, const std::string& message); // invia dati al lient specificato
        std::string receiveData(int clientSocket, int bufferSize = 1024); // riceve dati dal clientSocket

        int getSocketfd() const {
            return sockfd;
        }
    };
```

file descriptor: è un intero univoco e serve come handler per riferirsi a quel sopcket nelle operazioni successive.
consente inoltre al programma di identificare il socket e di esgeuire operazioni su di esso come:

- send()
- recv()
- close()
- lettura e scrittura

Socket.cpp
COSTRUTTORE:
crea una socket utilizzando la funzione socket():

- AF_INET: specifica l'uso del protocollo IPv4
- SOCK_STREAM: indica l'uso di un socket TCP (connection oriented)
- specifica ilk protocollo predefinto, in questo caso TCP

se la creazione della socket è fallita, si trminerà il programma con un codice di errore. Solitamente accade quando nopn ci sono più risorse di rete

DISTRUTTORE:
chiudere il socket è fondamentale per non sprecare risorse di rete

METODO BIND()
configura l'indirizzo (address) del socket:

- sin_family = AF_INET: Specifica il protocollo IPv4.
- sin_addr.s_addr = INADDR_ANY: Indica che il server accetta connessioni da tutte le interfacce di rete (localhost, rete locale, ecc.).
- sin_port = htons(port): Converte il numero di porta in formato big-endian, come richiesto dalle funzioni di rete.

viene chiamata poi la bind per associare il socket a un indirizzo IP e una porta specifica.

METODO LISTEN()
mette il socket in "ascolto" per le connessioni in arrivo
backlog: specifica la lunghezza massima della coda di connessioni in attesa
= 5: può gestire fino a 5 connessioni in attesa prima di rifiutarne delle altre

METODO CLOSESOCKET():
chiude il socket in modo esplicito.

METODO SENDDATA():
la sua funzione è quella di inviare un messaggio al client.
Accetta 2 parametri:

- clientSocket: file descriptor del socket con il client connessioni

- message: stringa da inviare

si utilizza la funzion send, che invia i dati al client:
- message.c_str() converte la stringa C++ in una stringa C (un array di caratteri).
- message.size() specifica la lunghezza del messaggio.
- L'ultimo parametro è 0 (flag predefinito).

METODO RECEIVEDATA():
la sua funzione è quella di ricevere i dati dal client connesso.
accetta 2 pòarametri:
- clientSocket: file descriptor del socket con il client.
- bufferSize: la dimensione massima di dati che possiamo ricevere.

crea un buffer di bufferSize e con la memset() lo pulisce.
Usa la funzione recv() per la ricezione dei dati, la sua funzione è quella di memorizzare i dati nel buffer, se i dati sono più grandi di bufferSize allora verranno presi sono i primi byte fino al bufferSize.
Infine restituisce il numero di byte ricevuti:
- clientSocket: il file descriptor del socket con il client da cui vogliamo ricevere i dati.
- buffer: un array di caratteri (buffer) in cui verranno memorizzati i dati ricevuti.
- bufferSize: la dimensione massima (in byte) dei dati da ricevere.
- 0: flag (opzioni) per il comportamento di recv() (0 = comportamento predefinito).

ServerSocket.hpp
classe ServerSocket:

```
class ServerSocket : public Socket {
    private:
        ThreadPool threadpool; // crea un oggetto threadpool, in modo da poterlo gestire
    public:
        ServerSocket(int port, size_t threadsCount); // costruttore in cui si inizializzano la porta e il numero di thread
        int acceptConnection(); // metodo che accetta la connesisone con il client
        void handleClient(int clientSocket); // gestisce il client

        ThreadPool& getThreadPool() { // getter per il threadpool, inacessibile dal main perchè privato
            return threadpool;
        }
};
```

ServerSocket.cpp
COSTRUTTORE:
inizializza il threadpool. il costruttore accetta due parametri:
- port: la porta su cui il server ascolterà le connessioni in ingresso.
- threadsCounter: il numero di thread da utilizzare per gestire i client.
Viene passato al costruttore della classe ThreadPool che gestisce il pool di thread per il multithreading.

bind(): associa il socket ad una porta specifica
listen(5): il server inizia ad ascoltare le connessioni sulla porta con un backlog di 5 connessioni massime in attesa

METODO ACCEPTCONNECTION():
- int addrlen = sizeof(address): calcola dimensione in byte della struttura address,
usata per memorizzare l'indirizzo del client che sta cercando di connettersi con il server
La variabile addrlen serve a indicare alla funzione accept() quanto spazio è disponibile per

memorizzare l'indirizzo del client.
La funzione accept() popolerà questa struttura con l'indirizzo del client che si è connesso.
- int newSocket = accept(sockfd, (struct sockaddr*)&address, (socklen_t*)&addrlen): accetta una connessione in entrata su un socket server
- sockfd: socket di ascolto, associato tramite la bind();
- (struct sockaddr*)&address: esegue un cast della struttura address a un tipo generico sockaddr*. accept() accetta un puntatore a sockaddr come parametro per supportare sia IPv4 che IPv6
al suo interbno sarà presente l'indirizzo del client che si è connesso (IP e porta)
- (socklen_t*)&addrlen: accept() deve sapere la lunghezza della struttura address, usata per sapere quanto spazio si ha per memorizzare l'indirizzo del client.
addrlen al termine viene aggiornata con la dimensione dell'indirizzo del client

Infine questa funzione ritorna un nuovo socket (newSocket) (prinicipalmente usato per la comunicazione con il client che si è appena connesso), che servirà come identificatore del socket associato alla connessione con il client

METODO HANDLECLIENT():
questa funzione prende in input un clientSocket, che è il socket associato al client che si è connesso.
Il server gestisce la comunicazione con il client su un thread separato.

CientSocket.hpp
```
class ClientSocket : public Socket {
    public:
        void connectToServer(const std::string& serverAddress, int port); // metodo per effettuare la connessione al server
};
```

ClientSocket.cpp
liberiria:
#include <arpa/inet.h>: utilizzata per manipolare gli indirizzi IP, con la funzione inet_pton() converte un indirizzo IP da stringa a formato binario

METODO CONNECTTOSERVER():
accetta 2 parametri:
- const std::string& serverAddress: indica l'indirizzo del server, quindi l'IP sottoforma di stringa del server a cui il client tenta di connettersi
- int port: la porta del server alla quale connettersi

address.sin_port = htons(port): htons() --> funzione che converte il numero della porta in formato "network byte order", in modo che i byte siano inviati correttamente nella rete

inet_pton(AF_INET, serverAddress.c_str(), &address.sin_addr): Converte l'indirizzo IP del server (che è in formato stringa) in un formato binario che può essere utilizzato da sockaddr_in.
La funzione inet_pton restituisce un valore positivo se la conversione ha successo, o un valore negativo (<= 0) se c'è stato un errore.

connect(sockfd, (struct sockaddr*)&address, sizeof(address)): Tenta di stabilire la connessione con il server. Il descrittore di socket sockfd viene utilizzato per eseguire la connessione, passando l'indirizzo del server e la sua lunghezza.