Московский Авиационный Институт

(Национальный Исследовательский Университет)

Кафедра 806 «Вычислительная информатика и программирование»

Факультет: «Информационные технологии и прикладная математика»

Курсовая работа

«Сервер сообщений»

3 семестр

|  |  |
| --- | --- |
| Группа: | М8О-208Б-18, №12 |
| Студент: | Коростелев Дмитрий Васильевич |
| Преподаватель: | Миронов Евгений Сергеевич |
| Оценка: |  |
| Дата: | 20.04.2020 |

Москва, 2020

**Содержание**

1. **Задание**..................................................................................................**3**
2. **Объяснение работы сервера сообщений**.........................................**3**
   1. **ServerClass.h**.............................................................................**3**
   2. **DataBaseClass.h**........................................................................**6**
   3. **SocketMenedger.h**.....................................................................**7**
3. **Объяснение работы класса Client**....................................................**8**
4. **Дополнительные модули и функции**……………………..………**10**
   1. **ООП оболочка библиотеки winsock**………………………**10**
   2. **ThreadMenedger**………………………………………….…**12**
   3. **Архитектура обработки команд**…………...……………...**12**
   4. **CharStream**………………………………………………......**13**
5. **Плюсы и минусы**..............................................................................**13**
6. **Заключение**........................................................................................**13**
7. **Задание**

Реализовать сервер сообщений, который поддерживает несколько очередей сообщений, а также библиотеку для работы с сервером вида:

a. var connection = Connect(string serverAddress); // var connection = Connect(«127.0.0.1:8312»);

b. SendMessage(connection, queue, message)

c. RecieveMessage(connection, queue)

Сам сервер должен поддерживать cli, то есть запуск должен осуществляться следующим образом:

a. server.out 8312 (поднимает сервер сообщений на порту 8312)

b. Сервер должен работать на нативных сокетах

c. Должна быть возможность отправлять сообщения большого размера (например, 1ГБ)

Репозиторий GitHub: *https://github.com/Dmitry4K/MessageServer*

1. **Объяснение работы сервера сообщений**

**2.1. ServerClass.h**

Сервер сообщений – модульная программа, которая принимает различные запросы от других устройств, формирует ответ на них и отсылает его.

Ответом на запросы может быть переданная клиенту строка, либо файл, в зависимости от того, какой ресурс запросил клиент, также сервер может принять строку, либо файл.

Весь сервер сообщений – соответствующий класс, заголовочный файл которого представлен ниже.

Для обеспечения многопоточности (одновременной работы с несколькими пользователями) требовалось реализовать некоторые функции, которые будут распределять между собой входящие запросы и параллельно их обрабатывать.

Весь сервер работает на одном процессе на одном сокете, запросы обрабатываются на разных потоках. В связи с этим имеем ограничение в количество одновременно подключенных пользователей ~1000.

Основные модули сервера – хранилище базы данных (класс со своими методами и логикой), менеджер подключенных сокетов (хранит активные подключенные к серверу сокеты), приемщик входящих сообщений и исполнитель команд.

|  |
| --- |
| #pragma once  #define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS  #include<iostream>  #pragma comment (lib,"Ws2\_32.lib")  #include <winsock2.h>  #include<Windows.h>  #include<string>  #include<utility>  #include<vector>  #include<queue>  #include<mutex>  #include<thread>  #include<sstream>  #include"DataBaseClass.h"  #include"SocketMenedger.h"  #include"MyServerParser.h"  #include"../MySockets/MySockets.h"  #include"../Additional/MyProtClasses.h"  #include"../MyCharStream/MyCharStream.h"  const int ACTIVE\_SOCKETS\_COUNT = 200;  const int DEFAULT\_SLEEP\_TIME = 10; //100  const int DEFAULT\_COUNT = 1000; //10  const int OFF = 0;  const int ON = 1;  struct MyCommandClass;  struct ServerClass;  const int CHECK\_TIME = 10000;  const int ACTIVE = 1;  const int NOT\_ACTIVE = 0;  const int IN\_USED = 2;  struct ThreadMenedgerClass {  private:  ServerClass\* Server = nullptr;  std::vector<std::pair<std::thread, int\*>> container;  std::queue<int> used;  std::mutex door;  int state = NOT\_ACTIVE;  std::thread TimeChecker;  friend void ThrMenExecuteFunction(MyCommandClass\*, int\*, ServerClass\* Server);  friend void TimeCheckerFunction(ThreadMenedgerClass\* menedger);  public:  void SetServer(ServerClass\* Server);  ThreadMenedgerClass();  ThreadMenedgerClass(ServerClass\* Server);  ~ThreadMenedgerClass();  void Add(MyCommandClass\*);  };  void ThrMenExecuteFunction(MyCommandClass\*, int\*, ServerClass\* Server);  void TimeCheckerFunction(ThreadMenedgerClass\* menedger);  struct ServerClass {  private:  MySocketClass HostSocket;  SocketMenedger ActiveSockets;  DataBaseClass Data;  MyParserClass Parser;  ThreadMenedgerClass Threads;  std::map<std::string, MyCommandClass\*> CommandMap;  std::thread ExecuteThread;  std::thread ReceiveThread;  std::thread AcceptThread;  int ExecuteThreadState = OFF;  int ReceiveThreadState = OFF;  int AcceptThreadState = OFF;  MyProtQueue<MyCommandClass\*> Commands;  void BlockThread();  public:  std::mutex DataMutex;  std::mutex CoutMutex;  ServerClass(int argc, char\* argv[]);  ServerClass(const std::string& adr);  ServerClass();  ~ServerClass();  void Start();  MySocketClass& GetSocket();  DataBaseClass& GetData();  friend void ReceiveFunction(ServerClass\* Server);  friend void ExecuteFunction(ServerClass\* Server);  friend void AcceptFunction(ServerClass\* Server);  friend MyCommandClass;  };  void ReceiveFunction(ServerClass\* Server);  void ExecuteFunction(ServerClass\* Server);  void AcceptFunction(ServerClass\* Server);  struct MyCommandClass {  int count = 0;  std::vector<char\*> params;  std::string name;  int socket = 0;  virtual void execute(ServerClass\* node) const = 0;  virtual void argument\_parsing(MyCharStreamClass& stream) = 0;  virtual void copy(MyCommandClass\*&) const = 0;  ~MyCommandClass() {  for (int i = 0; i < params.size(); ++i) {  delete[] params[i];  }  }  }; |

ServerClass поддерживает несколько конструкторов: конструктор, который принимает номер порта, на котором будет работать сервер (сервер запускается по локальному адресу). После вызова данного конструктора сервер блокирует текущий поток и начинает слушать запросы на присоединение, конструктор который принимает аргументы командной строки. Этот конструктор получает порт из командной строки если тот задан, в ином случае, сервер не запускается и выводит сообщение о неправильном формате командной строки.

Запуск сервера начинается с вызова метода Start(), этот метод автоматически вызывается, если конструктор успешно отработал свои инструкции. Внутри метода Start() происходит следующее – к хранилищу базы данных привязывается имя файла, подключается библиотека для работы с сокетами windows, создается и связывается сокет. Затем запускается несколько крайне важных функций на отдельных потоках - ReceiveFunction, ExecuteFunction, AcceptFunction.

Функция AcceptFunction – принимает входящие запросы на подключение к серверу, как только, какое-нибудь устройство подключилось к серверу, подключенный сокет отправляется в SocketMendger, сервер начинает слушать полученный сокет.

ReceiveFunction – принимает входящие запросы из подключенных сокетов посредством SocketMenedger, то есть ReceiveFunction по очереди пытается через заданное кол-во миллисекунд принять какое-нибудь сообщение из всех подключенных сокетов. Если сообщение получено, оно парсится вложенным классом MyServerParser, создается экземпляр команды с параметрами для ее обработки, затем команда отправляется в очередь сообщений Commands, на последующую обработку ExecuteFunction.

ExecuteFunction – работает с очередью Commands, логика данного модуля такова – если очередь не пуста, забираем полученное сообщение, и пытаемся отправить его на обработку в ThreadMenedger.

**2.2 DataBaseClass.h**

Хранилище базы данных – класс DataBaseClass, хранит в себе вектор очередей сообщений, каждой сообщение представляет из себя экземпляр MessageClass. Присутствуют методы чтения и записи базы на диск в виде текстового файла.

|  |
| --- |
| #pragma once  #include<iostream>  #include<fstream>  #include<vector>  #include<queue>  #include<map>  #include<string>  #include<list>  const int FILE\_TYPE = 0;  const int STRING\_TYPE = 1;  const int INGURED\_TYPE = 2;  const int UNKNOWN\_TYPE = -1;  const std::string WAY\_TO\_DATA = "D:\\dev\\OSKP2\\data.txt";  const std::string FOLDER = "D:\\dev\\OSKP2\\server\_storage\\";  class MessageClass {  std::string Data;  int Type;  public:  MessageClass();  MessageClass(const std::string&, int t);  bool Empty();  int GetType();  std::string GetData();  void Write(std::ofstream& file);  void Read(std::ifstream& file);  };  class DataBaseClass {  private:  std::string File;  std::map<std::string, std::queue<MessageClass>> Data;  std::list<MessageClass\*> Files;  void ReadQueue(std::ifstream& file);  void WriteQueue(std::ofstream& file, const std::string& qid, std::queue<MessageClass>& q);  public:  DataBaseClass();  ~DataBaseClass();  DataBaseClass(const DataBaseClass&) = delete;  std::queue<MessageClass>& GetQueueByName(const std::string&);  void AddQueue(const std::string&);  std::map<std::string, std::queue<MessageClass>>::iterator isExistQueue(const std::string&);  std::map<std::string, std::queue<MessageClass>>::iterator End();  bool RemoveQueue(const std::string&);  void Upload(const std::string&);  void Write(const std::string&);  void NewBase(const std::string&);  }; |

MessageClass состоит из строки, текст сообщения и типа сообщения(файл, либо строка).

DataBaseClass – хранит в себе путь до файла откуда будет прочитана или же записана база данных. Инструментарий класс – набор методов на картой очередей, с помощью этих методов легко оперировать всей базой данной: можем добавлять очереди, добавлять сообщения в очередь со специальным идентификатором, удалять целые очереди, либо сообщения в очереди, названия методов в точности отражают их функционал, единственное, что может оказаться неочевидным это то, что если при вызове метода AddMessageInQueue не будет найдена очередь, эта очередь будет создана и в нее будет положено сообщение.

**2.3 SocketMenedger**

Для хранения активных сокетов используется класс SocketMenedger, содержит в себе вектор сокетов и очередь. Данный класс, своего рода умный аллокатор, нужен для грамотного распределения памяти, то есть, если удаляется какой-то сокет в середине вектора, номер ячейки, из которой был удален сокет помещается в очередь. При последующем добавление в SocketMeneder, новый сокет будет помещен в тот номер, что хранится в очереди, если очередь пустая, то новый сокет добавляется в конец.

1. **Объяснение работы класса Client**

ClientClass\* – класс для работы с сервером, позволяет отправлять и принимать различные сообщения на сервер. Хранит свой сокет и сокет сервера

ClientClass(const char\*) – подключается к серверу по адресу , переданному в данный конструктор.

СlientClass::Connect(const char\*) - подключается к серверу по адресу , переданному в данный метод, возвращает ошибку SOCKET\_ERROR, если не подключился, иначе – сокет.

int СlientClass::Receive(const std::string& qid, std::string& dest); – получить сообщение из очереди с идентификатором qid, результат будет записан в dest.

int СlientClass::SendText(const std::string& qid, const std::string& text); - отправить строку на сервер в очередь qid.

int СlientClass::SendFile(const std::string& qid, const std::string& file); - отправить файл на сервер в очерель qid. Передача файлов осуществляется посредством их деления на более маленькие тексты(пакеты) фиксированной длины.

|  |
| --- |
| #pragma once  #include<iostream>  #define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS  #pragma comment (lib,"Ws2\_32.lib")  #include <winsock2.h>  #include<Windows.h>  #include<string>  #include<utility>  #include<map>  #include"../Additional/SimpleTimer.h"  #include"../MySockets/MySockets.h"  #include"../Additional/MyProtClasses.h"  #include"../MyCharStream/MyCharStream.h"  #include"MyClientParser.h"  struct MyCommand;  const int DEFAULT\_SLEEP\_TIME = 10; //500  const int DEFAULT\_COUNT = 1000; //10  const int OFF = 0;  const int ON = 1;  const int FILE\_TYPE = 0;  const int STRING\_TYPE = 1;  const std::string DEFAULT\_FOLDER = "D:\\dev\\OSKP2\\client\_storage\\";  const int PACK\_SIZE = 10000;  struct MyCommandClass;  class ClientClass {  private:  MySocketClass HostSocket;  Timer MyTimer;  std::map<std::string, MyCommandClass\*> CommandMap;  std::thread ReceiveThread;  int ReceiveThreadState = OFF;  MyParserClass Parser;  std::queue<MyCommandClass\*> Commands;  int Send(const std::string& text);  std::string Folder = DEFAULT\_FOLDER;  inline void GetPacks();  public:  ClientClass(const std::string& adr);  ~ClientClass();  void Start();  int Connect(const std::string& adr);  int Disconnect();  int State();  const MySocketClass& GetSocket() const ;  int SendText(const std::string& qid, const std::string& text);  int SendFile(const std::string& qid, const std::string& file);  int Receive(const std::string& qid, std::string& dest);  const std::string& GetFolder() const;  void SetFolder(const std::string& f);  friend void ReceiveFunction(ClientClass\* Server);  };  void ReceiveFunction(ClientClass\* Server);  struct MyCommandClass {  int count = 0;  std::vector<char\*> params;  virtual void execute(ClientClass \*node) const = 0;  virtual void argument\_parsing(MyCharStreamClass& stream) = 0;  virtual void copy(MyCommandClass\*&) const = 0;  ~MyCommandClass() {  for (int i = 0; i < params.size(); ++i) {  delete[] params[i];  }  }  }; |

1. **Дополнительные модули и функции**

**4.1. ООП оболочка библиотеки winsock**

Для осуществления передачи сообщений между сервером и пользователями использовались стандартные windows сокеты. Функции которые предоставляет данная библиотека легла в основу MySocketClass - класс, с помощь. которого можно легко подключаться к сокетам, считывать и отправлять на них сообщения.

|  |
| --- |
| #pragma once  #define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS  #include<iostream>  #pragma comment (lib,"Ws2\_32.lib")  #include <winsock2.h>  #include<Windows.h>  #include<mutex>  //Состояния сокетов  const int EMPTY = 0;  const int CONNECTED = 1;  const int BINDED = 2;  //Переменный для хранения информации библиотеки winsock  static bool IS\_WSA\_STARTED = false;  static WSADATA W\_DATA;  static int SOCKET\_COUNT = 0; //Кол-во созданных в процессе экземпляров MySocketClass  class MySocketClass {  private:  std::string adr; //подключенный адресс  int handle = 0; //handle, который получаем при вызове socket()  int state = EMPTY; //Состояние сокета  //std::mutex door; //Можно добавить мьютекс, чтобы сделать сокет потоко-безопасным  public:  MySocketClass();  ~MySocketClass();  const std::string& GetAdr() const;  int GetSocketHandle() const;  int Connect(const std::string& way);  int Bind(const std::string& way);  int Close();  //методы отправки сообщений  int Send(int h, const char\*, int len);  int Send(const char\*, int len);  int Send(const char\*); //только для классических си строк !  int Send(const std::string& m);  int Send(int h, const std::string& m);  //методы приема сообщений  int Recieve(int h, char\*&);  int Recieve(char\*&);  int Recieve(std::string& m); //только для классических строк  int Recieve(int h, std::string& m); //только для классических строк  int State() const ;  }; |

**4.2. ThreadMenedger**

Для выполнения команд используется специальаный класс ThreadMenedger, основа которого - вектор с потоками. У данного класса есть только один метод - Add(MyCommandsClass\*), который принимает указатель на команду. Эта инструкция запускает отдельный поток для обработки переданной команды и отправляет команду на обработку в этот поток. Кроме того, через некоторое время класс в отдельном потоке проверяет уже созданные потоки. Таким образм при заврешении одной инструкции в потоке, этот поток будет вновь использован для выполнения новой инструкции.

**4.3. Архитектура обработки команд**

Так как сервер считывает с сокета некоторую строку, следовало подумать над тем, как связать текст, с какой-нибудь функцей которая будет обрабатывать ту или иную строку. В ранних версиях это работы я использовал примитивную структуру if - else, но она оказалось не удобной в том плане, что код становился громоздким, часто повторяющимся и нечитаемым. Для решения этой проблемы была применена структура std::map<std::string, MyCommandsCLass\*>, благодаря которой можно легко добавлять или убирать поддерживаемые сервером команды.

*Пример*: CommandMap["text"] = new TextCommand();

Как видно, таким образом можно легко связать экземпляр команды с определенной строкой. Выполняются же эти команды, когда наступает их очередь. Происходит разыменование и выполняется нужная инструкция. Стоит отметить, что все команды наследуются от одного абстрактного класса MyCommandClass, что и делает данную систему удобной для дальнейшего расширения.

|  |
| --- |
| struct MyCommandClass {  int count = 0;  std::vector<char\*> params;  std::string name;  int socket = 0;  virtual void execute(ServerClass\* node) const = 0;  virtual void argument\_parsing(MyCharStreamClass& stream) = 0;  virtual void copy(MyCommandClass\*&) const = 0;  ~MyCommandClass() {  for (int i = 0; i < params.size(); ++i) {  delete[] params[i];  }  }  }; |

Чтобы релизовать одну команду нужно написать тело функций execute (что делает команда), argument\_parsing (как доставать параметры), copy (как создавать копию команды).

**4.4. CharStream**

По заданию требовалось, чтобы сервер имел возможность передавать файлы всех типов, а не только текстовые, а также иметь возможность удобно парсить си строки в которых могут находится куски сжатых файлов, было принято решение реализвовать свой потоковый ввод, который имеет схожий функционал что и std::istringstream, однако MyCharStreamClass поддерживает управление сжатой информацией.

1. **Плюсы и минусы**

Плюсы - система команд, которая позволяет легко "обучить" сервер новым командам. Относительная простота. Модульность.

Минусы - обработка команд выполняется в потоках, что сильно ограничивает кол-во активных пользователей.

1. **Заключение**

В ходе выполнения данного проекта был реализован простой сервер сообщений, с соответствующим классом для работы с этим сервер. Данный сервер может послужить неплохим инструментом для коммуникации между небольшой группой людей, или стать основой для полноценного сервера для обработки большого количества запросов при дополнительных модификациях.

В ходе выполнения курсового проекта, стало очевидным, что проектирование больших программ и методов передачи между программами сообщений играет крайне важную роль при разработке, так как при не грамотной реализации одного из этих пунктов приходится множество раз поводить рефакторинг, в следствии чего увеличивается время разработки.