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

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*

2. 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 ответ. 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 ~1500.

Основные модули сервера – хранилище базы данных (класс со своими методами и логикой), менеджер подключенных сокетов (хранит активные подключенные к серверу сокеты), приемщик входящих сообщений, обработчик входящих сообщений, функция ответа на полученный запрос и вектор с потоками, на которых запускается данная функция к определенному сокету.

```
class ServerClass {
private:
	const std::string WAY_TO_DATA = "data.json";
	DataBaseClass Data;
	int Port;
	int SocketHandle;
	SocketMenedger ActiveSockets;
	std::queue<std::pair<int, char*>> HandleCommand;
	bool isExit = false;
	std::thread Handler;
	std::thread ExecuteStarter;
	std::thread Console;
	std::vector <std::pair<int, std::thread>> Executors;
	void Accept();
	void Close();
	void Start();

public:
	const static int CLIENT_CLASS_BUFFER_SIZE = 100;
	const static int EXECUTOR_COUNT = 10;
	const static int ACTIVE_SOCKETS_COUNT = 10;
	const static int FILE_BUFFER_SIZE = 10000;
	ServerClass(int argc, char* argv[]);
	ServerClass(int p);
	~ServerClass();

	int GetPort();
	int Send(int Socket, const char* b);
	int Send(int Socket, const char* b, int bytes);
	int Recieve(int recvSocket, char**);
	int Recieve(int recvSocket, char**, int b);
	int Recieve(char**);
	bool IsExit();
	void ConsoleStart();
	friend void HandlerFunction(ServerClass* Server);
	friend void ExecuteStarterFunction(ServerClass* Server);
	friend void ExecuteFunction(ServerClass* Server, int s, char* msg, int& res);
	friend void ConsoleFunction(ServerClass* Server, HANDLE pipe);
};
void HandlerFunction(ServerClass* Server);
void ExecuteStarterFunction(ServerClass* Server);
void ExecuteFunction(ServerClass* Server, int s, char* msg, int& res);
void ConsoleFunction(ServerClass* Server, HANDLE pipe);
```

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 конструкторы успешно отработали свои инструкции. Внутри метода Start() происходит следующее – к хранилищу базы данных привязывается имя файла, далее запускается вторая консоль, откуда пользователь может вводить команды (данная консоль служит для отключения сервера и поддерживает одну команду – «exit»), подключается библиотека для работы с сокетами windows, создается и связывается сокет, запускается приемщик сообщений (std::thread Handler) и модуль, который запускает потоки для обработки запросов (std::thread ExecuteStarter), вызывается метод Accept().

Метод Accept() – блокирует текущий поток и начинает 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.

Handler – (функция – HandlerFunction) поток, который принимает входящие запросы из подключенных сокетов посредством SocketMenedger, то есть Handler по очереди пытается каждые 100 миллисекунд принять какое-нибудь сообщение из всех подключенных сокетов. Если сообщение получено, оно отправляется в очередь сообщений HandleCommand, на последующую обработку ExecuteStarter.

ExecuteStarter – (функция – ExecuteStarterFunction) работает с очередью HandleCommand, логика данного модуля такова – если очередь не пуста,

забираем полученное сообщение, и пытаемся отправить его на обработку в какой-нибудь свободный поток. На этом потоке запускается функция – ExecuteFunction

void ExecuteFunction(ServerClass* Server, int sock, char* msg, int& res) – функция, которая непосредственно формирует запрос на полученное сообщение, в зависимости от того, что требует клиент, добавить в какую-то очередь сообщение, либо забрать какое-то сообщение, ExecuteFunction – либо отправляет сообщение, либо принимает.

Функции прием и отправки сообщения Send и Receive отправляют сообщение на заданный сокет (имеется две реализации для отправки целой строки, и для отправки заданного количества байт из буфера). Большие файлы передаются кусками, сначала сервер и клиент обмениваются информацией о типе сообщения, далее принимающей стороне отправляется размер файла и количество пакетов, на которое будет разделен файл, далее принимается это количество пакетов. Простая строка отправляется одним пакетом.

Для того, чтобы сервер мог понять какой сокет отключился от сервера, сервер перед обработкой сообщения посылает на сокет строку “Start”, далее, если отправка произошла успешно, начинается формирование ответа на запрос.

2.2 DataBaseClass.h

Хранилище базы данных – класс DataBaseClass, хранит в себе вектор очередей сообщений, каждой сообщение представляет из себя экземпляр MessageClass.

```
struct MessageClass {
	const static int FILE = 0;
	const static int STRING = 1;
	std::string Data;
	int Type;
	MessageClass() {}
	MessageClass(std::string);
	MessageClass(std::string, int t);
};

class DataBaseClass {
private:
	std::string File;
```

```
        std::map<std::string, std::queue<MessageClass>> Data;
public:
        DataBaseClass();
        DataBaseClass(const char*);
        ~DataBaseClass();
        std::queue<MessageClass>& GetQueueByName(const std::string&);
        void AddMessageInQueue(const std::string&, MessageClass);
        MessageClass GetFrontMessageInQueue(const std::string&);
        void PopMessageInQueue(const std::string&);
        bool isExistQueue(const std::string&);
        bool Upload(const char*);
        void Write(const char*);
        void NewBase(const char*);
        void AddQueue(const char*);
        bool RemoveQueue(const char*);
};
```

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.

Вся База данных рабоает с json файлами, для этого используется библиотека rapidjson.

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.

3. 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() – пустой конструктор, обнуляет сокеты.

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

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

ClientClass::std::string RecievePost(std::string id) – получить сообщение из очереди с идентификатором id.

ClientClass::void SendPost(std::string id, std::string msg) – отправить сообщение с в очередь с идентификатором id.Логика SendPost такова, что если в msg передано название файла, то оправляется файл, иначе – строка.

```
class ClientClass {
private:
	int SocketHandle = 0;
	int ServerSocket = 0;
	int Send(const char* b);
	int Send(const char* b, int bytes);
	char* Recieve();
	char* Recieve(int b);
public:
	const static int CLIENT_CLASS_ERROR_SOCKET = SOCKET_ERROR;
	const static int CLIENT_CLASS_BUFFER_SIZE = 100;
	const static int FILE_TYPE = 0;
	const static int STRING_TYPE = 1;
	const static int FILE_BUFFER_SIZE = 10000;
	const static int WAITING_TIME = 200;
	ClientClass();
	ClientClass(const char* str);
	~ClientClass();
	int GetSocket();
	int Connect(const char* str);
	int Disconnect();
	std::string RecievePost(std::string id);
	void SendPost(std::string id, std::string msg);
};
```

4. Плюсы и минусы

Главным плюсом данного проекта является то, что сервер работает на одном сокете, сам сервер состоит из модулей, которые при желании могут быть легко изменены без вреда для друг друга, за счет очереди входящих сообщений сервер быстро принимает запросы. К минусам стоит отнести возможность сбоя работы клиента при слишком долго ответе на запрос, также так как запросы обрабатываются на потоках, количество пользователей сильно ограничено, однако эту проблему можно решить, запуская обработку запрос на отдельном процессе.

5. 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.