# Передача сообщений по протоколу TCP/IP с использованием системной библиотеки Windows WinSock.

Автор: Томашевич Константин.

Содержание

[Передача сообщений по протоколу TCP/IP с использованием системной библиотеки Windows WinSock. 1](#_Toc9098743)

[Простейшее приложение с использованием WinSock на языке С. 2](#_Toc9098744)

[C++ обёртка над WinSock (Socket Helpers) 2](#_Toc9098745)

[Утилиты 2](#_Toc9098746)

[Инициализация WinSock 2](#_Toc9098747)

[Буффер посылаемого сообщения 3](#_Toc9098748)

[Буфер принимаемого сообщения 5](#_Toc9098749)

[Клиентский сокет 7](#_Toc9098750)

[Серверный сокет 11](#_Toc9098751)

[Результат 14](#_Toc9098752)

[Клиент-серверный HelloWorld 14](#_Toc9098753)

[Описание приложения 14](#_Toc9098754)

[Общие константы 14](#_Toc9098755)

[HelloWorldServer 14](#_Toc9098756)

[HelloWorldClient 15](#_Toc9098757)

[Результат 16](#_Toc9098758)

[Простой почтовый сервер и клиент для него 17](#_Toc9098759)

[Описание приложения 17](#_Toc9098760)

[Общие константы 17](#_Toc9098761)

[Представление почтового аккаунта на сервере 18](#_Toc9098762)

[Простой почтовый сервер 19](#_Toc9098763)

[Клиент для простого почтового сервера 27](#_Toc9098764)

[Результат 32](#_Toc9098765)

[Выводы 33](#_Toc9098766)

[Ссылки 33](#_Toc9098767)

## Простейшее приложение с использованием WinSock на языке С.

Для базового знакомства с технологией очень хорошо подходит официальный туториал от Microsoft [Getting Started With WinSock.](https://docs.microsoft.com/en-us/windows/desktop/winsock/getting-started-with-winsock) В ходе его выполнения обучающийся может создать простейшее сокетное TCP/IP приложение, работа которого детально описанна в оффициальном туториале. В дальнейшем будем считать, что студент ознакомился с этим туториалом (кроме секции Advanced Winsock Samples) и описанным в нём теорией.

## C++ обёртка над WinSock (Socket Helpers)

Как можно заметить при прохождении оффициального туториала, писать на чистом С достаточно неудобно, особенно для людей, не имевших достаточного опыта в функциональном программировании. Поэтому для упрощения выполнения данной лабораторной работы следует разработать обёртку над WinSock на языке С++.

### Утилиты

Для работы с ошибками WinSock будем использовать механизм шаблонного универсального исключения как простейший механизм создания пользовательских классов исключений в С++. В данной работе не будем рассматривать его подробно, студент может проста скачать необходимые файлы из [репозитория на GitHub](https://github.com/KonstantinTomashevich/gamedev-utils/tree/master/UniversalException).

### Инициализация WinSock

Инициализация и деинициализация WinSock достаточно проста, но для достижения полного покрытия необходимого функционала WinSock нам необходимо покрыть и её удобными простыми функцыями. Сначала создадим файл Init.hpp, который объявляет функции инициализации и деинициализации, а также класс для исключения инициализации:Top of Form

Bottom of Form

 Top of Form

#pragma once  
namespace Init  
{  
void LoadWindowsSocketLibrary ();  
void UnloadWindowsSocketLibrary ();  
  
namespace Exceptions  
{  
class UnableToLoadWSA;  
}  
}

Bottom of Form

Листинг 1. Init.hpp.

Теперь займёмся реализацией этих функций. От оффициального туториала Microsoft отличается тут только поведение в случае ошибки инициализации: програма создаёт исключение.

#include "Init.hpp"  
#pragma comment(lib, "Ws2\_32.lib")  
#ifndef WIN32\_LEAN\_AND\_MEAN  
#define WIN32\_LEAN\_AND\_MEAN  
#endif  
  
#include <windows.h>  
#include <winsock2.h>  
#include <ws2tcpip.h>  
#include <iphlpapi.h>  
#include <stdio.h>  
#include <Utils/UniversalException.hpp>  
  
WSAData wsaData;  
  
namespace Init  
{  
void LoadWindowsSocketLibrary ()  
{  
 int returnCode;  
 if ((returnCode = WSAStartup (MAKEWORD(2, 2), &wsaData)))  
 {  
 throw UniversalException <Exceptions::UnableToLoadWSA> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to start win sock library! Error: " +  
 std::to\_string (returnCode) + ".");  
 }  
}  
  
void UnloadWindowsSocketLibrary ()  
{  
 WSACleanup ();  
}  
}

Листинг 2. Init.cpp.

### Буффер посылаемого сообщения

Создание нового сообщения для WinSock как простого буфера на С довольно неудобно, поэтому будет довольно хорошим решением создать в обёртке специальные классы для работы как с буффером посылаемого сообщения, так и с буфером принимаемого сообщения.

Разберём основные функции, которые мы желаем иметь при работе с буфером посылаемого сообщения:

* Записать строку в конец буфера.
* Записать целое число в конец буфера.
* Записать дробное число в конец буфера.
* Записать последовательность байт в конец буфера.
* Получить текущий размер буфера.
* Получить текущий размер выделенной под буфер памяти.
* Если выделеннной памяти не хватает, буфер должен расширить её до необходимого размера.

Мы можем покрыть вышеуказанные функции следующим объявлением класса:

#pragma once  
#include <cstdlib>  
#include <string>  
  
class OutputMessageBuffer  
{  
public:  
 OutputMessageBuffer (size\_t initialSize);  
 virtual ~OutputMessageBuffer ();  
  
 const char \*GetConstCBuffer () const;  
 char \*GetCBuffer ();  
  
 size\_t GetCapacity () const;  
 size\_t GetSize () const;  
  
 void WriteInt (int value);  
 void WriteFloat (float value);  
 void WriteString (const std::string &string);  
 void WriteBuffer (const char \*buffer, size\_t size);  
  
protected:  
  
private:  
 void UpdateCapacity (size\_t neededAddition);  
  
 char \*cBuffer\_;  
 size\_t capacity\_;  
 size\_t size\_;  
};

Листинг 3. OutputMessageBuffer.hpp.

Дополнительно обсуждать реализацию данного класса не имеет особого смысла, так как это известные любому студенту, прошедшему курс Системного Программирования, операции с памятью.

#include <math.h>  
#include "OutputMessageBuffer.hpp"  
  
OutputMessageBuffer::OutputMessageBuffer (size\_t initialSize)  
 : capacity\_ (initialSize),  
 size\_ (0)  
{  
 cBuffer\_ = (char \*) malloc (capacity\_);  
}  
  
OutputMessageBuffer::~OutputMessageBuffer ()  
{  
 free (cBuffer\_);  
}  
  
const char \*OutputMessageBuffer::GetConstCBuffer () const  
{  
 return cBuffer\_;  
}  
  
char \*OutputMessageBuffer::GetCBuffer ()  
{  
 return cBuffer\_;  
}  
  
size\_t OutputMessageBuffer::GetCapacity () const  
{  
 return capacity\_;  
}  
  
size\_t OutputMessageBuffer::GetSize () const  
{  
 return size\_;  
}  
  
void OutputMessageBuffer::WriteInt (int value)  
{  
 WriteBuffer ((char \*) &value, sizeof (int));  
}  
  
void OutputMessageBuffer::WriteFloat (float value)  
{  
 WriteBuffer ((char \*) &value, sizeof (float));  
}  
  
void OutputMessageBuffer::WriteString (const std::string &string)  
{  
 WriteBuffer (string.c\_str (), string.size () + 1);  
}  
  
void OutputMessageBuffer::WriteBuffer (const char \*buffer, size\_t size)  
{  
 UpdateCapacity (size);  
 memcpy (cBuffer\_ + size\_, buffer, size);  
 size\_ += size;  
}  
  
void OutputMessageBuffer::UpdateCapacity (size\_t neededAddition)  
{  
 if (size\_ + neededAddition > capacity\_)  
 {  
 capacity\_ += neededAddition < capacity\_ / 5 ? capacity\_ / 5 : neededAddition;  
 cBuffer\_ = (char \*) realloc (cBuffer\_, capacity\_);  
 }  
}

Листинг 4. OutputMessageBuffer.cpp.

### Буфер принимаемого сообщения

Аналогичным образом разберём основные функции, которые должен выполнять буфер принимаемого сообщения:

* Поддерживать текущее положение чтения в буфере.
* Прочитать из текущего положения чтения строку.
* Прочитать из текущего положения чтения целое число.
* Прочитать из текущего положения чтения дробное число.
* Прочитать из текущего положения чтения строку и передвинуть положение чтения.
* Прочитать из текущего положения чтения целое число и передвинуть положение чтения.
* Прочитать из текущего положения чтения дробное число и передвинуть положение чтения.
* Прочитать из текущего положения чтения последовательность байт и передвинуть положение чтения.
* Установить положение чтения.

Мы можем покрыть вышеуказанные функции следующим объявлением класса:

#pragma once  
#include <cstdlib>  
#include <string>  
  
class InputMessageBuffer  
{  
public:  
 explicit InputMessageBuffer (size\_t maximumSize);  
 virtual ~InputMessageBuffer ();  
  
 size\_t GetPosition () const;  
 size\_t GetMaximumSize () const;  
 int IntFromPosition () const;  
 float FloatFromPosition () const;  
 std::string StringFromPosition () const;  
 const char \*GetConstCBuffer () const;  
  
 char \*GetCBuffer ();  
 void SetPosition (size\_t position);  
 int NextInt ();  
 float NextFloat ();  
 std::string NextString ();  
 void CopyNextBuffer (char \*to, size\_t size);  
  
 class Exceptions  
 {  
 public:  
 class EndReached;  
 };  
  
protected:  
  
private:  
 void CheckPosition () const;  
  
 char \*cBuffer\_;  
 size\_t position\_;  
 size\_t maximumSize\_;  
};

Листинг 5. InputMessageBuffer.hpp.

Обсуждать реализацию в данном случае также не имеет смысла по тем же причинам, что и в листинге реализации буфера принимаемого сообщения.

#include "InputMessageBuffer.hpp"  
#include <Utils/UniversalException.hpp>  
  
InputMessageBuffer::InputMessageBuffer (size\_t maximumSize)  
 : position\_ (0),  
 maximumSize\_ (maximumSize)  
{  
 cBuffer\_ = new char[maximumSize];  
}  
  
InputMessageBuffer::~InputMessageBuffer ()  
{  
 delete[] cBuffer\_;  
}  
  
size\_t InputMessageBuffer::GetPosition () const  
{  
 return position\_;  
}  
  
size\_t InputMessageBuffer::GetMaximumSize () const  
{  
 return maximumSize\_;  
}  
  
int InputMessageBuffer::IntFromPosition () const  
{  
 CheckPosition ();  
 return \*(int \*) (cBuffer\_ + position\_);  
}  
  
float InputMessageBuffer::FloatFromPosition () const  
{  
 CheckPosition ();  
 return \*(float \*) (cBuffer\_ + position\_);  
}  
  
std::string InputMessageBuffer::StringFromPosition () const  
{  
 CheckPosition ();  
 return std::string (cBuffer\_ + position\_);  
}  
  
const char \*InputMessageBuffer::GetConstCBuffer () const  
{  
 return cBuffer\_;  
}  
  
char \*InputMessageBuffer::GetCBuffer ()  
{  
 return cBuffer\_;  
}  
  
void InputMessageBuffer::SetPosition (size\_t position)  
{  
 position\_ = position;  
}  
  
int InputMessageBuffer::NextInt ()  
{  
 int value = IntFromPosition ();  
 SetPosition (GetPosition () + sizeof (int));  
 return value;  
}  
  
float InputMessageBuffer::NextFloat ()  
{  
 float value = FloatFromPosition ();  
 SetPosition (GetPosition () + sizeof (float));  
 return value;  
}  
  
std::string InputMessageBuffer::NextString ()  
{  
 std::string value = StringFromPosition ();  
 SetPosition (GetPosition () + value.size () + 1);  
 return value;  
}  
  
void InputMessageBuffer::CopyNextBuffer (char \*to, size\_t size)  
{  
 CheckPosition ();  
 const char \*source = GetConstCBuffer () + position\_;  
 memcpy (to, source, size);  
}  
  
void InputMessageBuffer::CheckPosition () const  
{  
 if (position\_ >= maximumSize\_)  
 {  
 throw UniversalException <Exceptions::EndReached> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to read from position " +  
 std::to\_string (position\_) + " as it higher than max " + std::to\_string (maximumSize\_) + ".");  
 }  
}

Листинг 6. InputMessageBuffer.cpp.

### Клиентский сокет

Тут и в дальнейшем будем называть клиентским сокетом такой сокет, который удовлетворяет одному из следующих определений:

1. Сокет создан на стороне клиента с целью подключения к серверному сокету и обмена с ним сообщениями.
2. Сокет получен сервером как результат выполнения функции accept и будем использоваться для обмена сообщениями с данным клиентом.

Нетрудно заметить, что данные два типа сокетов отличаются только методом создания, имея при этом одинаковую логику работы с посылаемыми и принимаемыми сообщения, что позволяет определить их как один класс с двумя разными конструкторами.

Рассмотрим основные функции клиентского сокета:

* Проверить, есть ли входящие сообщения, которые можна принять.
* Послать сообщение.
* Принять сообщение.
* Отключиться.

#pragma once  
#include <winsock2.h>  
#include <string>  
#include "InputMessageBuffer.hpp"  
#include "OutputMessageBuffer.hpp"  
  
class ClientSocket  
{  
public:  
 ClientSocket (const std::string &serverNode, const std::string &serverName);  
 explicit ClientSocket (SOCKET cSocket);  
 virtual ~ClientSocket ();  
  
 void Disconnect ();  
 bool AnyDataReceived () const;  
 int Receive (InputMessageBuffer &message);  
 int Send (const OutputMessageBuffer &message);  
  
 class Exceptions  
 {  
 public:  
 class UnableToParseAddress;  
 class UnableToCreateCSocket;  
 class UnableToConnect;  
 class UnableToDisconnect;  
 class ErrorDuringReceive;  
 class ErrorDuringSend;  
 };  
  
protected:  
  
private:  
 SOCKET cSocket\_;  
};

Листинг 7. ClientSocket.hpp.

Отметим, что первый конструктор создан для первого типа клиентских сокетов, а второй, соответсвенно, для второго типа.

Перед тем, как приступить к реализации, опишем включённые в ClientSocket.cpp файлы:

#include "ClientSocket.hpp"  
#pragma comment(lib, "Ws2\_32.lib")  
#ifndef WIN32\_LEAN\_AND\_MEAN  
#define WIN32\_LEAN\_AND\_MEAN  
#endif  
  
#include <windows.h>  
#include <winsock2.h>  
#include <ws2tcpip.h>  
#include <iphlpapi.h>  
#include <stdio.h>  
#include <Utils/UniversalException.hpp>  
  
typedef struct addrinfo addrinfo;

Листинг 8. Включения в ClientSocket.cpp.

Теперь приступим к реализации первого конструктора. Не сложно заметить, что в данном случае отличия от оффициального туториала незначительны. Абсалютно аналогичным образом мы получаем информацию про адрес из адресов узла и порта.

ClientSocket::ClientSocket (const std::string &serverNode, const std::string &serverName)  
{  
 int returnCode;  
 addrinfo \*resultAddress = nullptr;  
 addrinfo hints;  
  
 ZeroMemory (&hints, sizeof (hints));  
 hints.ai\_family = AF\_UNSPEC;  
 hints.ai\_socktype = SOCK\_STREAM;  
 hints.ai\_protocol = IPPROTO\_TCP;  
  
 returnCode = getaddrinfo (serverNode.c\_str (), serverName.c\_str (), &hints, &resultAddress);  
 if (returnCode)  
 {  
 throw UniversalException <Exceptions::UnableToParseAddress> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to parse address! Code: " +  
 std::to\_string (returnCode) + ".");  
 }

Листинг 9. Первый конструктор ClientSocket, начало.

Далее поочерёдно будем пробовать подключиться ко всем возможным адресам, указанным в адресной информации, полученной ранее. Подключение осуществляется ровно таким же образом, как и в оффициальном туториале. Также возвращаем исключение конструктора в случае невозможности подключения ни к одному возвращённому аддресу.

for (addrinfo \*ptr = resultAddress; ptr != nullptr; ptr = ptr->ai\_next)  
 {  
 cSocket\_ = socket (ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);  
 if (cSocket\_ == INVALID\_SOCKET)  
 {  
 freeaddrinfo (resultAddress);  
 throw UniversalException <Exceptions::UnableToCreateCSocket> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable create CSocket! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
  
 returnCode = connect (cSocket\_, ptr->ai\_addr, (int) ptr->ai\_addrlen);  
 if (returnCode == SOCKET\_ERROR)  
 {  
 closesocket (cSocket\_);  
 cSocket\_ = INVALID\_SOCKET;  
 continue;  
 }  
  
 break;  
 }  
  
 freeaddrinfo (resultAddress);  
 if (cSocket\_ == INVALID\_SOCKET)  
 {  
 throw UniversalException <Exceptions::UnableToConnect> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to establish connection!");  
 }  
}

Листинг 10. Первый конструктор ClientSocket, конец.

Реализация второго конструктора намного проще, так как мы подразумеваем, что мы сразу получили валидный сокет как аргумент в конструкторе. Также добавим в листинг деструктор, так как в нём мы просто закрываем объект сокета WinSock.

ClientSocket::ClientSocket (SOCKET cSocket)  
 : cSocket\_ (cSocket)  
{  
  
}  
  
ClientSocket::~ClientSocket ()  
{  
 closesocket (cSocket\_);  
}

Листинг 11. Второй конструктор и деструктор ClientSocket.

Отключение сокета производится просто вызовом функции shutdown библиотеки WinSock и проверкой результата этой функции на ошибочность.

void ClientSocket::Disconnect ()  
{  
 int returnCode = shutdown (cSocket\_, SD\_BOTH);  
 if (returnCode == SOCKET\_ERROR)  
 {  
 throw UniversalException <Exceptions::UnableToDisconnect> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable disconnect socket! Code: " +  
 std::to\_string (returnCode) + ".");  
 }  
}

Листинг 12. Отключения сокета ClientSocket.

Проверка существования входных сообщений проводится в помощью функции select библиотеки WinSock. Функция принимает на вход текущий сокет как основной, список сокетов для проверки, в котором находится только текущий. На вход передаётся нулевой таймаут, так как мы должны получить результат на текущий момент, а не заблокировать поток в ожидании таймаута или получения данных.

bool ClientSocket::AnyDataReceived () const  
{  
 fd\_set readSet;  
 FD\_ZERO(&readSet);  
 FD\_SET(cSocket\_, &readSet);  
  
 timeval timeout;  
 timeout.tv\_sec = 0;  
 timeout.tv\_usec = 0;  
 return select (cSocket\_, &readSet, NULL, NULL, &timeout) > 0;  
}

Листинг 13. Проверка существования входных сообщений в ClientSocket.

Получение сообщения отличается от оффициального примера только использованием в качестве контейнера нашего класса для принимаемых сообщений и проверкой результата на ошибки с возможным выбросом исключения.

int ClientSocket::Receive (InputMessageBuffer &message)  
{  
 int resultCode = recv (cSocket\_, message.GetCBuffer (), message.GetMaximumSize (), 0);  
 if (resultCode == SOCKET\_ERROR)  
 {  
 throw UniversalException <Exceptions::ErrorDuringReceive> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Error during receive! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
  
 return resultCode;  
}

Листинг 14. Получение сообщения в ClientSocket.

Отправка сообщения также очень похожа на оффициальный пример.

int ClientSocket::Send (const OutputMessageBuffer &message)  
{  
 int resultCode = send (cSocket\_, message.GetConstCBuffer (), message.GetSize (), 0);  
 if (resultCode == SOCKET\_ERROR)  
 {  
 throw UniversalException <Exceptions::ErrorDuringSend> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Error during send! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
   
 return resultCode;  
}

Листинг 15. Отправка сообщения в ClientSocket.

### Серверный сокет

Тут и в дальнейшем будем называть серверным сокетом такой сокет, который привязывается к адресу в системе и ждёт подключений.

Рассмотрим основные функции серверного сокета:

* Прослушивание данного адреса на предмет новых подключений.
* Проверка данного адреса на предмет новых подключений.
* Отключение серверного сокета.

Данным функциям отвечает следующее объявление класса:

#pragma once  
#include <winsock2.h>  
#include <string>  
  
class ClientSocket;  
class ServerSocket  
{  
public:  
 explicit ServerSocket (const std::string &serverName);  
 virtual ~ServerSocket ();  
  
 void Disconnect ();  
 ClientSocket \*WaitForNextClient ();  
 ClientSocket \*WaitForNextClientNonBlocking ();  
  
 class Exceptions  
 {  
 public:  
 class UnableToParseAddress;  
 class UnableToCreateCSocket;  
 class UnableToBindCSocket;  
 class UnableToListen;  
 class UnableToAccept;  
 class UnableToDisconnect;  
 };  
  
protected:  
  
private:  
 SOCKET cSocket\_;  
};

Листинг 16. ServerSocket.hpp.

Перед тем, как приступить к реализации, опишем включённые в ServerSocket.cpp файлы:

#include "ServerSocket.hpp"  
#pragma comment(lib, "Ws2\_32.lib")  
#ifndef WIN32\_LEAN\_AND\_MEAN  
#define WIN32\_LEAN\_AND\_MEAN  
#endif  
  
#include <windows.h>  
#include <winsock2.h>  
#include <ws2tcpip.h>  
#include <iphlpapi.h>  
#include <stdio.h>  
  
#include <Utils/UniversalException.hpp>  
#include "ClientSocket.hpp"  
  
typedef struct addrinfo addrinfo;

Листинг 17.Включение файлов в ServerSocket.cpp.

Создание серверного сокета достаточно похоже на таковое в оффициальном примере. Сначала мы получаем информацию про адрес, который необходимо занять. После этого создаём объект сокета WinSock и привязываем её с помощью функции bind библиотеки WinSock.

ServerSocket::ServerSocket (const std::string &serverName)  
{  
 int returnCode;  
 addrinfo \*resultAddress = NULL;  
 addrinfo hints;  
  
 ZeroMemory(&hints, sizeof (hints));  
 hints.ai\_family = AF\_INET;  
 hints.ai\_socktype = SOCK\_STREAM;  
 hints.ai\_protocol = IPPROTO\_TCP;  
 hints.ai\_flags = AI\_PASSIVE;  
 returnCode = getaddrinfo (NULL, serverName.c\_str (), &hints, &resultAddress);  
  
 if (returnCode)  
 {  
 throw UniversalException <Exceptions::UnableToParseAddress> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to parse address! Code: " +  
 std::to\_string (returnCode) + ".");  
 }  
  
 cSocket\_ = socket (resultAddress->ai\_family, resultAddress->ai\_socktype, resultAddress->ai\_protocol);  
 if (cSocket\_ == INVALID\_SOCKET)  
 {  
 freeaddrinfo (resultAddress);  
 throw UniversalException <Exceptions::UnableToCreateCSocket> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable create CSocket! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
  
 returnCode = bind (cSocket\_, resultAddress->ai\_addr, (int) resultAddress->ai\_addrlen);  
 freeaddrinfo (resultAddress);  
  
 if (returnCode == SOCKET\_ERROR)  
 {  
 throw UniversalException <Exceptions::UnableToBindCSocket> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to bind CSocket! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
}

Листинг 18. Конструктор сокета ServerSocket.

В деструкторе мы также просто закрываем сокет.

ServerSocket::~ServerSocket ()  
{  
 closesocket (cSocket\_);  
}

Листинг 19. Деструктор сокета ServerSocket.

Отключение серверного сокета также аналогично отключению клиентского.

void ServerSocket::Disconnect ()  
{  
 int returnCode = shutdown (cSocket\_, SD\_BOTH);  
 if (returnCode == SOCKET\_ERROR)  
 {  
 throw UniversalException <Exceptions::UnableToDisconnect> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable disconnect socket! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
}

Листинг 20. Отключение сокета ServerSocket.

Прослушка адреса, к которому привязан серверный сокет, осуществляется похожим на оффициальный пример методом. Сначала мы проверяем валидность прослушивания адреса с помощью функции listen. После этого мы вызываем функцию accept, которая принимает первое подключение в очереди. Если такого подключения нет, accept блокирует поток выполнения в ожидании нового подключения. После подключения нового клиентского сокета мы создаём объект для этого сокета и возвращаем его как результат выполнения.

ClientSocket \*ServerSocket::WaitForNextClient ()  
{  
 if (listen (cSocket\_, SOMAXCONN) == SOCKET\_ERROR)  
 {  
 throw UniversalException <Exceptions::UnableToListen> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to listen to CSocket! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
  
 SOCKET clientSocket = accept (cSocket\_, NULL, NULL);  
 if (clientSocket == INVALID\_SOCKET)  
 {  
 throw UniversalException <Exceptions::UnableToAccept> (std::string (\_\_FILE\_\_) + ":" +  
 std::to\_string (\_\_LINE\_\_) + " Unable to accept new connection! Error: " +  
 std::to\_string (WSAGetLastError ()) + ".");  
 }  
  
 return new ClientSocket (clientSocket);  
}

Листинг 21. Прослушивание адреса в ServerSocket.

### Результат

В результате мы написали С++ обёртку для WinSock, которая позволит нам намного проще работать с сокетами Windows и даст возможность использовать механизм исключений. Полный код получившейся библиотеки можно найти на [GitHub](https://github.com/KonstantinTomashevich/WinSockProjects/tree/master/sources/SocketHelpers).

## Клиент-серверный HelloWorld

### Описание приложения

Обычно изучение любой технологии начинают с написания некого HelloWorld-приложения с использованием этой технологии. В случае с организацией клиент-серверной работы сокетов обычно начинают с такого приложения:

* На каком-либо заранее известном порте стартует HelloWorldServer.
* HelloWorldClient знает этот порт и при получении адреса хоста автоматически начинает подключение.
* HelloWorldServer принимает все подключения и посылает каждому подключившемуся клиенту сообщение «Hello world from server!», после чего обрывает соединение с клиентом. В нашем случае, будем отправлять ещё и число 123 в конце сообщения.
* Для упрощения обычно пропускают обработку сложных ошибок и критических случаев.

### Общие константы

Для удобства работы объявим используемый порт в общем файле Defs.hpp:

#pragma once  
#define SERVER\_NAME "27016"

Листинг 22. HelloWorld: Defs.hpp.

### HelloWorldServer

Для начала перечислим все включения в Server.cpp:

#include <SocketHelpers/Init.hpp>  
#include <SocketHelpers/ServerSocket.hpp>  
#include <SocketHelpers/ClientSocket.hpp>  
#include <SocketHelpers/InputMessageBuffer.hpp>  
#include <SocketHelpers/OutputMessageBuffer.hpp>  
#include <Utils/CustomTerminate.hpp>  
#include "Defs.hpp"

Листинг 23. HelloWorld: включение в Server.cpp.

Теперь разберёмся, что у нас будем в main серверного приложения.

1. Для удобства заменим terminate на кастомный вариант, который перед остановкой приложения напечает вызвавшее остановку исключение, если оно относиться к типу AnyUniversalException.
2. Инициализировать WinSock теперь можно просто вызовом функции из библиотеки.
3. Серверный сокет создаётся просто вызовом конструктора, в котором мы уже всё необходимое реализовали.
4. Для избежания бесконечного цикла ограничим количество принятых клиентов тысячей.
5. В цикле ожидаем нового клиента и принимает подключение от него, тогда создаём новый тред для работы с этим клиентом.
6. После окончания цикла отключаем сервер и выгружаем WinSock.

int main ()  
{  
 std::set\_terminate (CustomTerminate);  
 Init::LoadWindowsSocketLibrary ();  
  
 ServerSocket \*server = new ServerSocket (SERVER\_NAME);  
 int maxConnections = 1000;  
 while (maxConnections--)  
 {  
 ClientSocket \*client = server->WaitForNextClient ();  
 CreateThread (NULL, 0, ClientThread, (LPVOID) client, 0, NULL);  
 }  
  
 server->Disconnect ();  
 delete server;  
 Init::UnloadWindowsSocketLibrary ();  
 return 0;  
}

Листинг 24. HelloWorld: main в Server.cpp.

Теперь рассмотрим функцию обработки подключения клиента:

1. Так как на вход функция принимает void\*, нам необходимо назад привести параметр к типу ClientSocket\*.
2. Отправлять сообщение будем в изолированном блоке кода, чтобы буфер отправки автоматически очистился после выхода из блока.
3. С помощью созданного буфера отправки мы можем легко создать отправляемое сообщение. Отметим, что буфер намеренно создаётся меньшего размера для проверки правильности работы реаллокации буфера.
4. Отправляем сообщение и ожидаем 1с, чтобы сообщение было точно отправлено до отключения клиента.
5. Отключаем клиента и оканчиваем выполнение клиентского потока.

DWORD WINAPI ClientThread (LPVOID param)  
{  
 auto \*client = static\_cast<ClientSocket \*> (param);  
 {  
 OutputMessageBuffer message (10);  
 message.WriteString ("HelloWorld from server!");  
 message.WriteInt (123);  
 client->Send (message);  
 Sleep (1000);  
 }  
  
 client->Disconnect ();  
 delete client;  
 return 0;  
}

Листинг 25. HelloWorld: ClientThread в Server.cpp.

### HelloWorldClient

Рассмотрим код клиента:

1. Для тех же целей устанавливаем свой terminate.
2. Инициализируем библиотеку WinSock и запрашивает адрес хоста у клиента.
3. Создаём и подключаем клиентский совет к введённому адресу.
4. Ожидаем 2с, таким образом проверим, что клиент всё ещё сможет прочитать сообщение, хоть сервер его и уже отключил.
5. Читаем сообщение в принимающий буфер.
6. Достаём и выводим значения из принимающего буфера.
7. Отключаем сокет и выгружаем библиотеку.

#include <SocketHelpers/Init.hpp>  
#include <SocketHelpers/ServerSocket.hpp>  
#include <SocketHelpers/ClientSocket.hpp>  
#include <SocketHelpers/InputMessageBuffer.hpp>  
#include <SocketHelpers/OutputMessageBuffer.hpp>  
#include <Utils/CustomTerminate.hpp>  
#include "Defs.hpp"  
  
int main ()  
{  
 std::set\_terminate (CustomTerminate);  
 Init::LoadWindowsSocketLibrary ();  
  
 std::string server;  
 std::cout << "Input server host: ";  
 std::cin >> server;  
  
 ClientSocket \*socket = new ClientSocket (server, SERVER\_NAME);  
 Sleep (2000);  
  
 {  
 InputMessageBuffer message (512);  
 socket->Receive (message);  
 std::string string = message.NextString ();  
 int number = message.NextInt ();  
 printf ("%s %d", string.c\_str (), number);  
 }  
  
 socket->Disconnect ();  
 delete socket;  
 Init::UnloadWindowsSocketLibrary ();  
 return 0;  
}

Листинг 26. HelloWorld: Client.cpp.

### Результат

Таким образом мы написали простейшее клиент-серверное приложение с использованием сокетов WinSock и нашей С++ обёртки. Полный код можно найти на [GitHub](https://github.com/KonstantinTomashevich/WinSockProjects/tree/master/sources/HelloWorld).

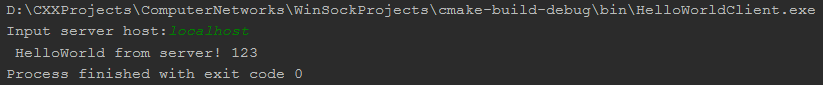


Рисунок 1. Результат исполнения клиента при активном сервере.

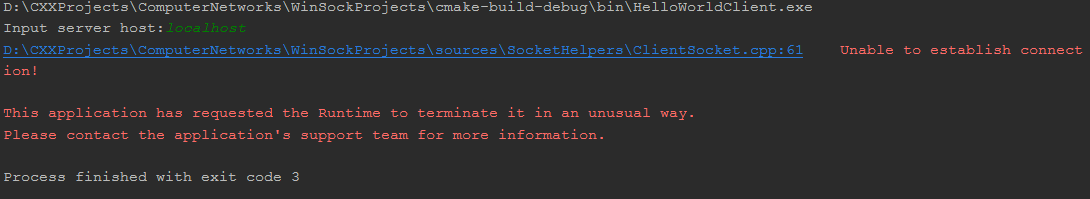


Рисунок 2. Результат исполнения клиента при отсутствии подключения к серверу.

## Простой почтовый сервер и клиент для него

### Описание приложения

Наша задача – написать простой почтовый сервер, работающий подобно POP3 почтовым серверам. Опишем требования к серверу:

* Для работы с почтой клиент должен идентифицировать себя с уже существующим аккаунтом или создать новый.
* Аккаунты сохраняются всё время выполнения сервера.
* Письма хранятся в аккаунте как очередь. Запрос первого письма удаляет его из очереди.
* Письмом будем называть тройку строк (отправитель, тема, сообщение).
* Клиент может запросить количество писем в аккаунте.
* Клиент может запросить первое письмо в аккаунте.
* Клиент может написать письмо на любой другой аккаунт. Если такой аккаунт отсутствует, клиент должен получить уведомление об ошибке.
* Сервер должен обслуживать неограниченное количество клиентов одновременно.
* Должна быть возможность выключить сервер консольной коммандой.
* Приложение должно корректно обрабатывать все ошибки.

Требования к клиенту:

* Клиент должен иметь возможность идентифицировать себя с уже существующим аккаунтом или создать новый.
* Клиент может запросить количество писем в аккаунте.
* Клиент может запросить первое письмо в аккаунте.
* Клиент может написать письмо на любой другой аккаунт. Если такой аккаунт отсутствует, клиент должен получить уведомление об ошибке.
* Клиент может отключиться от сервера с помощью консольной команды.
* Приложение должно корректно обрабатывать все ошибки.

### Общие константы

При рассмотрении требований можно заметить, что существует достаточно большое количество типов сообщений между клиентом и сервером. При этом на запрос одного типа клиент может получить сообщение сервера одного из нескольких типов. В этом случае будет логично использовать типовую подпись сообщений: в начале сообщение приписывается число, биективно связанное с типом сообщения. Для упрощения будем считать, что идентифицируя себя с несуществующим аккаунтом клиент создаёт новый.

Опишем типы сообщений от сервера к клиенту:

* Для действия необходима привязка к аккаунту.
* Возвращается количество писем в аккаунте.
* Возвращается первое письмо в аккаунте.
* Не найден аккаунт, на который должно было быть послано письмо.
* В привязанном аккаунте нет писем.
* Невозможно прислать первое письмо в аккаунте из-за внутренней ошибки.
* Письмо успешно отправлено.
* Невозможно отправить письмо из-за внутренней ошибки.

Опишем типы сообщений от клиента к серверу:

* Клиент идентифицирует себя с аккаунтом.
* Клиент посылает письмо.
* Клиент запрашивает количество писем в аккаунте.
* Клиент запрашивает первое письмо в аккаунте.

Также установим максимальный размер сообщения как 512 байт. В итоге получаем такой файл общих констант:

#pragma once  
#define PORT\_STRING "29377"  
#define MAX\_MESSAGE\_SIZE 512  
  
#define STC\_AUTH\_REQUIRED 1  
#define STC\_UNREAD\_COUNT 2  
#define STC\_POP\_MESSAGE 3  
#define STC\_TARGET\_NOT\_FOUND 4  
#define STC\_NO\_MESSAGES 5  
#define STC\_UNABLE\_TO\_POP 6  
#define STC\_SUCCESSFULLY\_PUSHED 7  
#define STC\_UNABLE\_TO\_PUSH 8  
  
#define CTS\_AUTH -1  
#define CTS\_PUSH\_MESSAGE -2  
#define CTS\_REQUEST\_UNREAD -3  
#define CTS\_REQUEST\_POP -4

Листинг 27. SimpleMail: Defs.hpp.

### Представление почтового аккаунта на сервере

Вынесем создание класса почтового ящика в отдельный пункт. Разберём необходимую функциональность почтового ящика:

* Сконструировать пустой ящик из логина и пароля.
* Осуществлять доступ для чтения логина и пароля созданного ящика.
* Вставить письмо в ящик.
* Достать письмо из ящика.
* Получить количество писем в ящике.

Отметим, что для избежания излишнего копирования удобнее будет разбить операцию получения письма из ящика на доступ для чтения первого письма в одном методе и простое удаление первого письма в другом методе. Также опишем структуру сообщения в данном ящике согласно требованиям выше. В итоге получим такое объявление класса:

#pragma once  
#include <string>  
#include <queue>  
  
class Mail  
{  
public:  
 typedef struct  
 {  
 std::string fromWhom;  
 std::string theme;  
 std::string text;  
 } Message;  
  
 Mail (const std::string &login, const std::string &password);  
 virtual ~Mail () = default;  
  
 const std::string &GetLogin () const;  
 const std::string &GetPassword () const;  
 int GetMessagesCount () const;  
  
 const Message &TopMessage () const;  
 void PopMessage ();  
 void PushMessage (const Message &message);  
  
protected:  
  
private:  
 std::string login\_;  
 std::string password\_;  
 std::queue <Message> messages\_;  
};

Листинг 28. SimpleMailServer: Mail.hpp.

Реализация данных методов не осуществляет проблемы при условии прохождения курса программирования, поэтому не будем её комментировать.

#include "Mail.hpp"  
  
Mail::Mail (const std::string &login, const std::string &password)  
 : login\_ (login),  
 password\_ (password),  
 messages\_ ()  
{  
  
}  
  
const std::string &Mail::GetLogin () const  
{  
 return login\_;  
}  
  
const std::string &Mail::GetPassword () const  
{  
 return password\_;  
}  
  
int Mail::GetMessagesCount () const  
{  
 return messages\_.size ();  
}  
  
const Mail::Message &Mail::TopMessage () const  
{  
 return messages\_.front ();  
}  
  
void Mail::PopMessage ()  
{  
 messages\_.pop ();  
}  
  
void Mail::PushMessage (const Mail::Message &message)  
{  
 messages\_.push (message);  
}

Листинг 29. SimpleMailServer: Mail.cpp.

### Простой почтовый сервер

Для начала перечислим включения в исполняемый файл сервера:

#include <SocketHelpers/Init.hpp>  
#include <SocketHelpers/ServerSocket.hpp>  
#include <SocketHelpers/ClientSocket.hpp>  
#include <SocketHelpers/InputMessageBuffer.hpp>  
#include <SocketHelpers/OutputMessageBuffer.hpp>  
  
#include <unordered\_map>  
#include <Utils/CustomTerminate.hpp>  
#include <SimpleMail/Defs.hpp>  
#include "Mail.hpp"

Листинг 30. SimpleMailServer: включения в Main.cpp.

Для упрощения написания воспользуемся механизмом глобальных переменных. Это приемлемо, так как мы не подразумеваем существования нескольких серверов в одном процессе, то есть аналогично созданию класса-синглетона.

bool \_working = true;  
int \_clientThreadsActive = 0;  
HANDLE \_registryMutex;  
HANDLE \_cerrMutex;  
std::unordered\_map <std::string, Mail \*> \_mails;

Листинг 31. SimpleMailServer: глобальные переменные в Main.cpp.

Опишем предназначение каждой переменной:

* \_working – флаг продолжения работы сервера. Как только он станет равен false, все клиентские потоки перейдут в режим завершения подключения и информирования главного потока о своём завершении.
* \_clientThreadsActive – количество активных на данный момент клиентских потоков. Как только \_working станет равно false, главный поток перейдёт в режим ожидания обнуления этой переменной, чтобы освободить все ресурсы.
* \_regestryMutex – мьютекс, обеспечивающий безопасность работы с регистром аккаунтов \_mails из разных потоков.
* \_cerrMutex – мьютекс, обеспечиваюзий безопасность работы с системный потоком вывода ошибок из разных потоков.
* \_mails – регистр аккаунтов, оформленный как хэш-карта, что позволительно для небольшого проекта-примера.

Дальше опишем декларации потоков в процессе сервера:

DWORD WINAPI ClientThread (LPVOID param);  
DWORD WINAPI AskShutdownThread (LPVOID param);

Листинг 32. SimpleMailServer: декларации функций потоков в Main.cpp.

Опишем их предназначение:

* ClientThread – как и в предыдущем примере занимается обработкой подключения клиента.
* AskShutdownThread – так как чтение из консоли является блокирующей операцией, нам необходимо содержать один поток, который ожидает нажатия клавишы завершения сервера.

Опишем декларацию остальных функций в коде сервера:

bool ParseClientMessage (ClientSocket \*client, Mail \*&mail);  
void PrintUnknownMessageToCerr (int code);  
Mail \*TryAuth (InputMessageBuffer &message);  
void RequestAuth (ClientSocket \*client);  
void SendUnread (ClientSocket \*client, Mail \*mail);  
void PopMessage (ClientSocket \*client, Mail \*mail);  
void ProceedWithMessage (ClientSocket \*client, InputMessageBuffer &inMessage, Mail \*mail);

Листинг 33. SimpleMailServer: декларация вспомогательных функций в Main.cpp.

Опишем их предназначение:

* ParseClientMessage – пробует прочесть тип сообщения клиента и отправить его на обработку в правильную вспомогательную функцию.
* PrintUnknownMessageToCerr – печает ошибку неизвестного кода сообщения от клиента.
* TryAuth – осуществляет обработку запроса на идентификацию клиента.
* RequestAuth – посылает клиенту сообщение о том, что ему необходимо индентифицировать себя с аккаунтом.
* SendUnread – осуществляет обработку запроса на количество писем в ящике.
* PopMessage – осуществляет обработку запроса на получения письма из ящика.
* ProceedWithMessage – осуществляет обработку запроса на посылку письма клиентом.

Теперь разберёмся с реализацией функции main:

int main ()  
{  
 bool successful = true;  
 \_registryMutex = CreateMutex (NULL, FALSE, NULL);  
 \_cerrMutex = CreateMutex (NULL, FALSE, NULL);  
  
 std::set\_terminate (CustomTerminate);  
 Init::LoadWindowsSocketLibrary ();  
  
 ServerSocket \*server = nullptr;  
 try  
 {  
 server = new ServerSocket (PORT\_STRING);  
 }  
 catch (AnyUniversalException &exception)  
 {  
 WaitForSingleObject (\_cerrMutex, INFINITE);  
 std::cerr << exception.GetException () << std::endl;  
 ReleaseMutex (\_cerrMutex);  
 successful = false;  
 }  
  
 if (successful)  
 {  
 CreateThread (NULL, 0, AskShutdownThread, NULL, 0, NULL);  
 while (\_working)  
 {  
 ClientSocket \*client = nullptr;  
 try  
 {  
 client = server->WaitForNextClientNonBlocking ();  
 }  
 catch (AnyUniversalException &exception)  
 {  
 WaitForSingleObject (\_cerrMutex, INFINITE);  
 std::cerr << exception.GetException () << std::endl;  
 ReleaseMutex (\_cerrMutex);  
 successful = false;  
 break;  
 }  
  
 if (client)  
 {  
 CreateThread (NULL, 0, ClientThread, (LPVOID) client, 0, NULL);  
 }  
 }  
  
 while (\_clientThreadsActive);  
 delete server;  
 }  
  
 Init::UnloadWindowsSocketLibrary ();  
 for (auto &loginMailPair : \_mails)  
 {  
 delete loginMailPair.second;  
 }  
  
 CloseHandle (\_registryMutex);  
 CloseHandle (\_cerrMutex);  
 return successful ? 0 : 1;  
}

Листинг 34. SimpleMailServer: main в Main.cpp.

Разберём отличия этой реализации от реализации в примере HelloWorldServer:

* Производится инициализация необходимых мьютексов.
* Производится отлов исключений из конструктора серверного сокета. В случае исключения в этой части кода выход из программы происходит нормально.
* Создаётся поток для запроса выхода.
* В цикле используется неблокирующая функция ожидания подключения, чтобы обеспечить насколько возможно быстрый выход в случае запроса выхода.
* Исключения ожидания также отлавливаются и подаются в системный поток печати ошибок. Исполнение программы при этом НЕ прерывается.
* После получения запроса на выход главный поток ожидает завершения всех клиентских потоков.
* К освобождению ресурсов добавилось освобождение мьютексов и созданных почтовых ящиков.

Рассмотрим код потока запроса выхода. Он не нуждается в дополнительных комментариях:

DWORD WINAPI AskShutdownThread (LPVOID param)  
{  
 std::cout << "Press q to shutdown server...";  
 while (getc (stdin) != 'q');  
 \_working = false;  
 return 0;  
}

Рисунок 35. SimpleMailServer: поток запроса выхода в Main.cpp.

Теперь рассмотрим изменения в клиентском потоке.

DWORD WINAPI ClientThread (LPVOID param)  
{  
 ++\_clientThreadsActive;  
 auto \*client = static\_cast<ClientSocket \*> (param);  
  
 {  
 OutputMessageBuffer message (MAX\_MESSAGE\_SIZE);  
 message.WriteInt (STC\_AUTH\_REQUIRED);  
 client->Send (message);  
 }  
  
 Mail \*mail = nullptr;  
 try  
 {  
 while (\_working)  
 {  
 if (client->AnyDataReceived () && !ParseClientMessage (client, mail))  
 {  
 break;  
 }  
 }  
 }  
 catch (AnyUniversalException &exception)  
 {  
 WaitForSingleObject (\_cerrMutex, INFINITE);  
 std::cerr << exception.GetException () << std::endl;  
 ReleaseMutex (\_cerrMutex);  
 }  
  
 try  
 {  
 client->Disconnect ();  
 }  
 catch (UniversalException <ClientSocket::Exceptions::UnableToDisconnect> &exception)  
 {  
 /\* No action, client already disconnected \*/  
 }  
  
 delete client;  
 --\_clientThreadsActive;  
 return 0;  
}

Листинг 36. SimpleMailServer: клиентский поток в Main.cpp.

* На входе в поток осуществляется инкементация счётчика потоков. Это атомарная операция в нотации компиляторов GNU GCC (для компиляции примеров используется GNU MinGW).
* Далее сервер уведомляет клиента о необходимости идентификации.
* Переменная mail используется как указатель на текущий связанный почтовый ящик.
* В цикле получения сообщений используются неблокирующие методы, чтобы осуществить наиболее быстрое выключения по запросу.
* В случае отключения с стороны клиента ParseClientMessage вернёт false (далее по исходному коду будет видно, что false возвращается только в этом случае).
* Вся циклическая обработка сообщений от клиента производится в блоке ловли исклчений. Если возникло исключение при работе с клиентом мы будем считать подключение оборванным.
* Отключение клиента также осуществляется в блоке ловли исключений. Заметим, что получение исключения на данным шагу просто значит отключение по инициативе клиента.
* В конце счётчик потоков декрементируется, это также атомарная операция.

Теперь перейдём к рассмотрению ParseClientMessage.

bool ParseClientMessage (ClientSocket \*client, Mail \*&mail)  
{  
 InputMessageBuffer inMessage (MAX\_MESSAGE\_SIZE);  
 if (client->Receive (inMessage) == 0)  
 {  
 return false;  
 }  
  
 int code = inMessage.NextInt ();  
 switch (code)  
 {  
 case CTS\_AUTH:  
  
 mail = TryAuth (inMessage);  
 if (mail)  
 { SendUnread (client, mail); }  
 else  
 { RequestAuth (client); }  
 break;  
  
 case CTS\_PUSH\_MESSAGE:  
  
 if (mail)  
 { ProceedWithMessage (client, inMessage, mail); }  
 else  
 { RequestAuth (client); }  
 break;  
  
 case CTS\_REQUEST\_POP:  
  
 if (mail)  
 { PopMessage (client, mail); }  
 else  
 { RequestAuth (client); }  
 break;  
  
 case CTS\_REQUEST\_UNREAD:  
  
 if (mail)  
 { SendUnread (client, mail); }  
 else  
 { RequestAuth (client); }  
 break;  
  
 default:  
  
 PrintUnknownMessageToCerr (code);  
 }  
  
 return true;  
}

Листинг 37. SimpleMailServer: ParseClientMessage в Main.cpp.

1. Сначала читаем полученное сообщение в буфер. Функция Receive возвращает число прочитанных байт, которое равно 0 тогда и только тогда, когда подключение оборвано.
2. Далее мы читаем код сообщения из буфера.
3. Через switch выбирается дальнейший обработчик полученного сообщения и производится его обработка.

Далее приводим код функции PrintUnknownMessageToCerr, он достаточно прост и не нуждается в комментариях.

void PrintUnknownMessageToCerr (int code)  
{  
 WaitForSingleObject (\_cerrMutex, INFINITE);  
 std::cerr << "Unknown message " << code << " received from client thread " <<  
 GetCurrentThreadId () << "." << std::endl;  
 ReleaseMutex (\_cerrMutex);  
}

Листинг 38. SimpleMailServer: PrintUnknownMessageToCerr в Main.cpp

Теперь рассмотрим обработчик запроса на авторизацию:

Mail \*TryAuth (InputMessageBuffer &message)  
{  
 std::string login = message.NextString ();  
 std::string password = message.NextString ();  
 Mail \*mail = nullptr;  
  
 WaitForSingleObject (\_registryMutex, INFINITE);  
 auto iterator = \_mails.find (login);  
  
 if (iterator == \_mails.end ())  
 {  
 mail = new Mail (login, password);  
 \_mails.insert (std::make\_pair (login, mail));  
 }  
 else if (iterator->second->GetPassword () == password)  
 {  
 mail = iterator->second;  
 }  
  
 ReleaseMutex (\_registryMutex);  
 return mail;  
}

Листинг 39. SimpleMailServer: TryAuth в Main.cpp

1. Сначала мы читаем логин и пароль из сообщения (в данном примере для упрощения не используется шифрование).
2. Дальнейшие операции с регистром аккаунтов осуществляются в критической секции.
3. Если аккаунт с таким логином не существует, мы создаём новый с таким логином и паролем, иначе проверяем пароль и в случае его совпадения даём доступ к ящику.
4. Функция возвращает новый связанный ящик для подключения, он может быть ошибочным, то есть nullptr.

Реализация функции RequestAuth не требует дополнительных комментариев:

void RequestAuth (ClientSocket \*client)  
{  
 OutputMessageBuffer message (MAX\_MESSAGE\_SIZE);  
 message.WriteInt (STC\_AUTH\_REQUIRED);  
 client->Send (message);  
}

Листинг 40. SimpleMailServer: RequestAuth в Main.cpp.

То же и реализацией SendUnread:

void SendUnread (ClientSocket \*client, Mail \*mail)  
{  
 OutputMessageBuffer message (MAX\_MESSAGE\_SIZE);  
 message.WriteInt (STC\_UNREAD\_COUNT);  
 message.WriteInt (mail->GetMessagesCount ());  
 client->Send (message);  
}

Листинг 41. SimpleMailServer: SendUnread в Main.cpp.

Разберёмся с реализацией функции PopMessage:

void PopMessage (ClientSocket \*client, Mail \*mail)  
{  
 OutputMessageBuffer message (MAX\_MESSAGE\_SIZE);  
 if (mail->GetMessagesCount () > 0)  
 {  
 const Mail::Message &mailMessage = mail->TopMessage ();  
  
 if (mailMessage.fromWhom.size () + mailMessage.text.size () +  
 mailMessage.theme.size () + 3 + sizeof (int) < MAX\_MESSAGE\_SIZE)  
 {  
 message.WriteInt (STC\_POP\_MESSAGE);  
 message.WriteString (mailMessage.fromWhom);  
 message.WriteString (mailMessage.theme);  
 message.WriteString (mailMessage.text);  
 }  
 else  
 {  
 message.WriteInt (STC\_UNABLE\_TO\_POP);  
 }  
  
 mail->PopMessage ();  
 }  
 else  
 {  
 message.WriteInt (STC\_NO\_MESSAGES);  
 }  
  
 client->Send (message);  
}

Листинг 42. SimpleMailServer: PopMessage в Main.cpp.

1. Сначала создаётся буфер для ответа.
2. Если количество сообщений нулевое, в буфер ответа просто записывается тип сообщения ошибки отсутствия писем.
3. Если письма есть, то необходимо проверить вмещается ли первое письмо в максимальный размер сообщения, иначе мы просто запишем в буфер тип сообщения ошибки обработки письма на сервере.
4. Если письмо удовлетворяет условиям размера, записываем тип сообщения как сообщение-письмо и записываем строки отправителя, темы и сообщения.
5. В конце буфер посылается клиенту вне зависимости от того, что в него записано.

Теперь рассмотрим ProceedWithMessage:

void ProceedWithMessage (ClientSocket \*client, InputMessageBuffer &inMessage, Mail \*mail)  
{  
 OutputMessageBuffer message (MAX\_MESSAGE\_SIZE);  
 std::string toWhom = inMessage.NextString ();  
 Mail::Message mailMessage;  
  
 mailMessage.fromWhom = mail->GetLogin ();  
 mailMessage.theme = inMessage.NextString ();  
 mailMessage.text = inMessage.NextString ();  
  
 if (mailMessage.fromWhom.size () + mailMessage.theme.size () +  
 mailMessage.text.size () + 3 + sizeof (int) < MAX\_MESSAGE\_SIZE)  
 {  
 Mail \*target = nullptr;  
 WaitForSingleObject (\_registryMutex, INFINITE);  
 auto iterator = \_mails.find (toWhom);  
  
 if (iterator != \_mails.end ())  
 {  
 target = iterator->second;  
 }  
  
 ReleaseMutex (\_registryMutex);  
 if (target != nullptr)  
 {  
 target->PushMessage (mailMessage);  
 message.WriteInt (STC\_SUCCESSFULLY\_PUSHED);  
 }  
 else  
 {  
 message.WriteInt (STC\_TARGET\_NOT\_FOUND);  
 }  
 }  
 else  
 {  
 message.WriteInt (STC\_UNABLE\_TO\_PUSH);  
 }  
  
 client->Send (message);  
}

Листинг 43. SimpleMailServer: ProceedWithMessage в Main.cpp.

1. Сначала создаётся буфер для ответа.
2. Считывается получатель письма.
3. Далее создаётся объект письма, в который записывается отправитель и считываются тема и сообщение.
4. Проверяется размер полученного письма, если он больше размера максимального сообщения, в буфер записывается тип сообщения ошибки обработки на сервере и обработка прекращается.
5. Далее в критической секции осуществляется поиск почтового ящега получателя.
6. Если невозможно найти почтовый ящик получателя (не зарегестрирован получатель с таким именем), в буфер записывается тип сообщения ошибка отсутсвия получателя.
7. В случае соблюдения всех вышеописаных условий, сообщение добавляется в ящик получателя и в буфер записывается тип сообщения успешной отправки.
8. В конце буфер посылается клиенту вне зависимости от того, что в него записано.

На этом реализация простого почтового сервера закончена. Полный код можно найти на [GitHub](https://github.com/KonstantinTomashevich/WinSockProjects/tree/master/sources/SimpleMail/Server).

### Клиент для простого почтового сервера

Для начала перечислим включения и функции в исполняемом файле клиента:

#include <SocketHelpers/Init.hpp>  
#include <SocketHelpers/ClientSocket.hpp>  
#include <SocketHelpers/InputMessageBuffer.hpp>  
#include <SocketHelpers/OutputMessageBuffer.hpp>  
#include <Utils/CustomTerminate.hpp>  
#include <SimpleMail/Defs.hpp>  
  
std::string ServerMessageCodeAsString (int code);  
void ProcessInitialMessage (ClientSocket \*socket);  
void MakeAuthRequest (ClientSocket \*socket);  
void MakeUnreadRequest (ClientSocket \*socket);  
void MakePushRequest (ClientSocket \*socket);  
void MakePopRequest (ClientSocket \*socket);

Листинг 44. SimpleMailClient: включения и объявления в Main.cpp.

Рассмотрим назначение функций:

* ServerMessageCodeAsString – конвертирует тип полученного сообщения в читаемую строку. Для удобства отладки мы всегда будем выводить на экран такую строку типа.
* ProcessInitialMessage – ожидает и выводит первичное сообщение, посылаемое сервером.
* MakeAuthRequest – запрашивает идентификационные данные с консоли и отправляет запрос на сервер. Также проверяет ответ от сервера.
* MakeUnreadRequest – отправляет запрос про количество писем в ящике и проверяет ответ от сервера.
* MakePushRequest – запрашивает новое письмо и отправляет его на сервер, проверяет ответ от сервера.
* MakePopRequest – запрашивает первое письмо в ящике и проверяет ответ от сервера.

Теперь разберём функцию main клиента:

int main ()  
{  
 std::set\_terminate (CustomTerminate);  
 Init::LoadWindowsSocketLibrary ();  
  
 std::string hostAddress;  
 std::cout << "Input host address: ";  
 std::cin >> hostAddress;  
  
 ClientSocket \*socket = nullptr;  
 try  
 {  
 socket = new ClientSocket (hostAddress, PORT\_STRING);  
 ProcessInitialMessage (socket);  
  
 std::string command;  
 while (command != "quit")  
 {  
 std::cout << "Input command: ";  
 std::cin >> command;  
  
 if (command == "auth")  
 { MakeAuthRequest (socket); }  
 else if (command == "unread")  
 { MakeUnreadRequest (socket); }  
 else if (command == "push")  
 { MakePushRequest (socket); }  
 else if (command == "pop")  
 { MakePopRequest (socket); }  
 }  
  
 socket->Disconnect ();  
 delete socket;  
 }  
 catch (AnyUniversalException &exception)  
 {  
 delete socket;  
 std::cerr << exception.GetException () << std::endl;  
 }  
  
 Init::UnloadWindowsSocketLibrary ();  
 return 0;  
}

Листинг 45. SimpleMailClient: main в Main.cpp.

При сравнении с версией из примера HelloWorldClient мы видим следующие отличия:

* Вся работа с сокетом обёрнута в блок ловли исключений. Если при работе с сокетом произошла ошибка, мы считаем соединение оборваным.
* Циклически запрашиваются с консоли и обрабатываются команды, пока не будет введена команда «quit».

Ниже приведёт код функции ServerMessageCodeAsString, он чрезвычайно прост и не нуждается в комментарии.

std::string ServerMessageCodeAsString (int code)  
{  
 switch (code)  
 {  
 case STC\_AUTH\_REQUIRED:  
  
 return "Auth required for this action!";  
  
 case STC\_UNREAD\_COUNT:  
  
 return "Unread messages count sent as additional data with this message.";  
  
 case STC\_POP\_MESSAGE :  
  
 return "Popped sent as additional data with this message.";  
  
 case STC\_TARGET\_NOT\_FOUND:  
  
 return "Message target not found!";  
  
 case STC\_NO\_MESSAGES:  
  
 return "Your mail is empty!";  
  
 case STC\_UNABLE\_TO\_POP:  
  
 return "Popped message is too big, unable to send it!";  
  
 case STC\_SUCCESSFULLY\_PUSHED:  
  
 return "Successfully sent!";  
  
 case STC\_UNABLE\_TO\_PUSH:  
  
 return "Unable to sent!";  
  
 default:  
  
 return "Unknown message!";  
 }  
}

Листинг 46. SimpleMailClient: ServerMessageCodeAsString в Main.cpp.

Также и с ProcessInitialMessage, мы просто получаем сообщение и выводим информацию про его код.

void ProcessInitialMessage (ClientSocket \*socket)  
{  
 InputMessageBuffer inMessage (MAX\_MESSAGE\_SIZE);  
 socket->Receive (inMessage);  
 std::cout << "Server: " << ServerMessageCodeAsString (inMessage.NextInt ()) << std::endl;  
}

Листинг 47. SimpleMailClient: ProcessInitialMessage в Main.cpp.

Теперь разберём функцию MakeAuthRequest:

void MakeAuthRequest (ClientSocket \*socket)  
{  
 std::string login;  
 std::string password;  
  
 std::cout << "Login: ";  
 std::cin >> login;  
  
 std::cout << "Password: ";  
 std::cin >> password;  
  
 OutputMessageBuffer outMessage (MAX\_MESSAGE\_SIZE);  
 outMessage.WriteInt (CTS\_AUTH);  
 outMessage.WriteString (login);  
 outMessage.WriteString (password);  
 socket->Send (outMessage);  
  
 InputMessageBuffer inMessage (MAX\_MESSAGE\_SIZE);  
 socket->Receive (inMessage);  
  
 int code = inMessage.NextInt ();  
 std::cout << "Server: " << ServerMessageCodeAsString (code) << std::endl;  
  
 if (code == STC\_UNREAD\_COUNT)  
 {  
 std::cout << "Additional data: " << inMessage.NextInt () << std::endl;  
 }  
}

Листинг 48. SimpleMailClient: MakeAuthRequest в Main.cpp.

1. С консоли считываются логин и пароль для идентификации.
2. Создаётся буфер посылаемого сообщения с типом запроса авторизации и строками логина и пароля. После чего сообщение отправляется на сервер.
3. Создаётся буфер принимаемого сообщения и клиент входит в состояние ожидания ответа от сервера.
4. Выводится на печать информация про код присланого сервером сообщения.
5. Если присланное сообщение – количество писем в ящике (стандартное поведение сервера при успешном входе) – выводим это число на печать.

Рассмотрим MakeUnreadRequest:

void MakeUnreadRequest (ClientSocket \*socket)  
{  
 OutputMessageBuffer outMessage (MAX\_MESSAGE\_SIZE);  
 outMessage.WriteInt (CTS\_REQUEST\_UNREAD);  
 socket->Send (outMessage);  
  
 InputMessageBuffer inMessage (MAX\_MESSAGE\_SIZE);  
 socket->Receive (inMessage);  
  
 int code = inMessage.NextInt ();  
 std::cout << "Server: " << ServerMessageCodeAsString (code) << std::endl;  
  
 if (code == STC\_UNREAD\_COUNT)  
 {  
 std::cout << "Additional data: " << inMessage.NextInt () << std::endl;  
 }  
}

Листинг 49. SimpleMailClient: MakeUnreadRequest в Main.cpp.

* Создаётся буфер посылаемого сообщения с кодом запроса на количество писем в ящике. Сообщение отправляется на сервер.
* Создаётся буфер принимаемого сообщения и клиент входит в состояние ожидания ответа от сервера.
* Выводится на печать информация про код присланого сервером сообщения.
* Если присланное сообщение – количество писем в ящике, то выводим это число на печать.

Перейдём к MakePushRequest:

void MakePushRequest (ClientSocket \*socket)  
{  
 std::string toWhom;  
 std::string theme;  
 std::string text;  
  
 char symbol;  
 std::cout << "To whom: ";  
 std::cin >> toWhom;  
  
 std::cin.get ();  
 std::cout << "Theme: ";  
 while ((symbol = std::cin.get ()) != '\n')  
 { theme += symbol; }  
  
 std::cout << "Text: ";  
 while ((symbol = std::cin.get ()) != '\n')  
 { text += symbol; }  
  
 OutputMessageBuffer outMessage (MAX\_MESSAGE\_SIZE);  
 outMessage.WriteInt (CTS\_PUSH\_MESSAGE);  
 outMessage.WriteString (toWhom);  
 outMessage.WriteString (theme);  
 outMessage.WriteString (text);  
 socket->Send (outMessage);  
  
 InputMessageBuffer inMessage (MAX\_MESSAGE\_SIZE);  
 socket->Receive (inMessage);  
  
 int code = inMessage.NextInt ();  
 std::cout << "Server: " << ServerMessageCodeAsString (code) << std::endl;  
}

Листинг 50. SimpleMailClient: MakePushRequest в Main.cpp.

Структура этой функции подобна структуре предыдущих двух, поэтому опустим пошаговое описание. Следует обратить внимание, что тема и текст сообщения считываются с помощью cin.get для того, чтобы считать весь ввод до перевода на новую строку.

И в конце рассмотрим MakePopRequest:

void MakePopRequest (ClientSocket \*socket)  
{  
 OutputMessageBuffer outMessage (MAX\_MESSAGE\_SIZE);  
 outMessage.WriteInt (CTS\_REQUEST\_POP);  
 socket->Send (outMessage);  
  
 InputMessageBuffer inMessage (MAX\_MESSAGE\_SIZE);  
 socket->Receive (inMessage);  
  
 int code = inMessage.NextInt ();  
 std::cout << "Server: " << ServerMessageCodeAsString (code) << std::endl;  
  
 if (code == STC\_POP\_MESSAGE)  
 {  
 std::cout << "From whom: " << inMessage.NextString () << std::endl;  
 std::cout << "Theme: " << inMessage.NextString () << std::endl;  
 std::cout << "Text: " << inMessage.NextString () << std::endl;  
 }  
}

Листинг 50. SimpleMailClient: MakePopRequest в Main.cpp.

Структура этой функции также похожа и не нуждается в излишнем комментировании. В ней мы отправляем запрос на сервер и ожидаем ответа. Если ответ имеет тип успешно доставленного первого письма, то выводим это письмо на печать.

### Результат

Таким образом был написан простой почтой сервер и клиент для работы с ним с использованием библиотеки WinSock и нашей С++ обёртки к ней. Полный код можно найти на [GitHub.](https://github.com/KonstantinTomashevich/WinSockProjects/tree/master/sources/SimpleMail/)

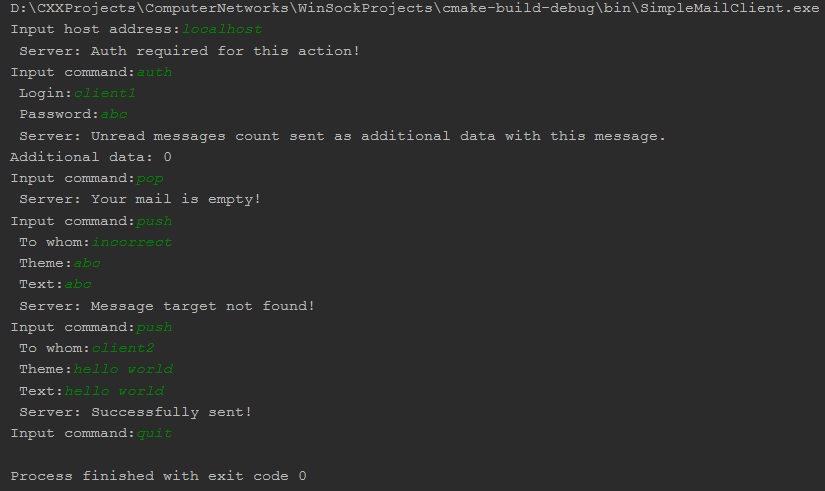


Рисунок 3. Работа клиентского приложения, идентификовавшегося как client1.

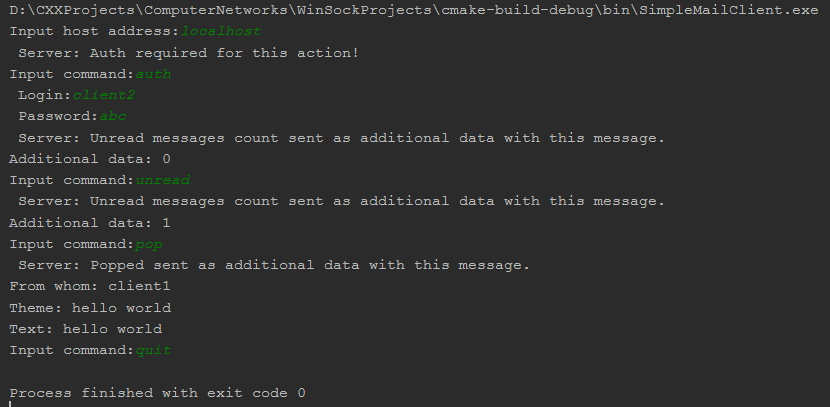


Рисунок 4. Работа клиентского приложения, идентификовавшегося как client2.

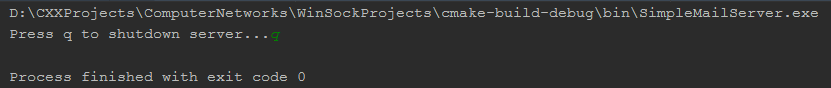


Рисунок 5. Заметим, что на сервере не происходило ошибок в ходе выполнения предыдущих двух клиентов и выключение было успешно произведено по нажатию клавишы «q».

## Выводы

1. В ходе выполнения данной работы была разработана обёртка на языке С++ над библиотекой WinSock, позволившая использовать механизм исключений и скрыть сложности работы с WinSock под довольно лаконичным интерфейсом.
2. Для базового тестирования обёртки была разработана простейшая клиент-серверная HelloWorld программа.
3. Для более глубоко примера работы с сокетами, а также для проверки удобства и стабильности работы нашей обёртки, были написаны простой почтовый сервер и клиент для него.

## Ссылки

1. [Репозиторий на GitHub](https://github.com/KonstantinTomashevich/WinSockProjects/) со всеми проектами из этого туториала. Сборка организована через CMake-скрипты.
2. [Getting Started with Winsock](https://docs.microsoft.com/en-us/windows/desktop/winsock/getting-started-with-winsock) – оффициальный туториал по библиотеке WinSock от Microsoft.