**7.3. Практическая работа № 2. Обмен данными по TCP-соединению**

**7.3.1. Цель и задачи работы**

Основнойцелью практической работы является приобретение навыков разработки простейшего распределенного приложения архитектуры клиент-сервер, осуществляющего обмен данными в локальной сети через Windows Sockets TCP-соединение.

Результатом практической работы является разработанное распределенное приложение со схемой взаимодействия процессов, описанной в разделе 3.4 и изображенной на рисунке 3.4.2 пособия.

**7.3.2. Теоретические сведения**

Теоретические сведения необходимые для выполнения практической работы изложены в разделах 3.2-3.11, 3.14 пособия.

**7.3.3. Разработка серверной части распределенного приложения**

**Задание 1.** Ознакомьтесь со схемой сервера, изображенной на рисунке 3.4.2 пособия. Создайте с помощью Visual Studio консольное приложение **ServerT** (наименование проекта), которое будет использовано для построения серверной части приложения (сервера). Включите необходимые директивы компилятора (указанные в разделе 3.2 пособия) для подключения динамической библиотеки **WS2\_32.LIB**. Откомпилируйте приложение, убедитесь в отсутствии ошибок.

**Задание 2.** В рамках приложения **ServerT**, созданного в задании 1, разработайте о функцию **SetErrorMsgText,** предназначенную для обработки стандартных ошибок библиотеки **WS2\_32.LIB**. Предполагается, что функция **SetErrorMsgText**  будет использоваться в операторе **throw** для генерации исключения при возникновении ошибок в функциях интерфейса Winsock2. Для получения кода ошибки функций Winsock2 примените функцию **WSAGetLastError,** описание которой приводится в разделе 3.3 пособия. Там же приводится полный список кодов возврата функции **WSAGetLastError** и пример ее использования.

**Задание 3.** Доработайте приложение **ServerT**  таким образом, чтобы оно только инициализировало библиотеку **WS2\_32.LIB** и завешало работу с этой библиотекой. Для этого используйте функции **WSAStartup** и **WSACleanup**, описанные в разделах 3.5 и 3.6 пособия. Обработку ошибок осуществите с помощью конструкции **try-catch** и функции **SetErrorMsgText**, разработанной в задании 2. Используйте пример программы, приведенный в разделе 3.6 пособия. Убедитесь в работоспособности приложения.

**Задание 4.** Доработайте приложение **ServerT** таким образом**,** чтобы оно создавало и закрывало сокет, предназначенный для ориентированного на поток соединения. Для этого используйте функции **socket** и **closesocket**, описанные в разделе 3.8. Обратите внимание на параметр **type** функции **socket**, указывающийтип соединения. Воспользуйтесь примером из раздела 3.7 пособия. Убедитесь в работоспособности приложения.

**Задание 5.** Добавьте в приложение **ServerT** вызовфункций **bind** и **listen** для установки параметров сокета и перевода его в режим прослушивания. Функция **bind** описана в разделе 3.8, а функция **listen** в разделе 3.9 пособия. Используйте порт **2000**, в качестве параметра сокета. Установка параметров сокета осуществляется с помощью структуры **SOCKADDR\_IN**. Описание этой структуры приводится в разделе 3.8 пособия. Сверьте схему полученной программы со схемой сервера, изображенной на рисунке 3.4.2. Выполните приложение, убедитесь в его работоспособности.

**Задание 6.** Добавьте в приложение **ServerT** вызовфункции **accept**, описание которой приводится в разделе 3.10 пособия. Следует обратить внимание на: 1) успешным результатом работы функции **accept** является новый сокет; 2) первым параметром функции **accept** является уже созданный ранее сокет; 3) второй параметр функции **accept**  **–** указатель на структуру **SOCKADDR\_IN** (не надо ее путать с уже применяемой выше), предназначенную для приема параметров, подключившегося сокета со стороны клиента сокета. Запустите приложение в режиме отладки (Debug) и убедитесь, что после выполнения функции **accept**, программа переходит в режим ожидания (зависает). Завершите приложение. Сохраните программу **ServerT** для дальнейшего применения.

**7.3.4. Разработка клиентской части распределенного приложения**

**Задание 7.** Ознакомьтесь со схемой клиента, изображенной на рисунке 3.4.2 пособия.Создайте с помощью Visual Studio новое консольное приложение **ClientT** (наименование проекта), которое будет использовано для построения клиентской части приложения (клиента). Повторите все те же действия для этого приложения, которые были сделаны в заданиях 2-4. Убедитесь в работоспособности приложения **ClientT**.

**Задание 8.** Добавьте в приложение **ClientT,** вызов функции **connect**. Описание функции и примера ее использования приводится в разделе 3.10 пособия. Следует обратить внимание на следующее: 1) параметры сокета сервера устанавливаются в структуре **SOCKADDR\_IN**; 2) для номера порта необходимо установить значение **2000**  (такой же номер, что установлен при параметризации сокета сервера а задании 5); 3) для установки номера порта используются специальные функции, описание которых приводится в разделе 3.8. Используйте в качестве IP-адреса собственный адрес компьютера **127.0.0.1** (интерфейс внутренней петли) – это даст возможность отладки приложения на одном компьютере. Запустите приложение на выполнение. Убедитесь, что функция **connect**  завершилась с ошибкой и обработка ошибок осуществляется корректно. Найдите полученный код ошибки в таблице 3.3.1 пособия и проанализируйте его.

**7.3.5. Обмен данными между сервером и клиентом**

**Задание 9.** Запустите на выполнение приложение **ServerT** и убедитесь, что оно приостановилось на вызове функции **accept**.Запустите на выполнение на этом же компьютере (используется интерфейс внутренней петли) приложение **ClientT**.Убедитесь, что сервер **ServerT**,вышел из состояния ожидания, а клиент  **ClientT** завершилсябез ошибок.

**Задание 10.** Доработайте программу сервера **ServerT** таким образом, чтобы после подсоединения клиента на экран консоли **ServerT** выводился IP-адрес и порт, подсоединившегося клиента. Необходимые значения находятся в структуре **SOCKADDR\_IN**, которая заполняется функцией **accept**. Используйте функции **htons** и **inet\_ntoa**, описанные в разделе 3.8. Убедитесь в работоспособности распределенного приложения **ClientT-ServеrT**.

**Задание 11.** Добавьте в программу сервера **ServerT** вызов функции **recv,** а в программу клиента вызов функции **send.** Описание этих функций приводится в разделе 3.11 пособия. Перешлите текст ***Hello from Client*** от клиента серверу. Выведите полученный сервером текст на экран консоли. Обратите внимание на то, что команда **recv**  в программе сервера использует сокет, созданный функцией **accept,** ане созданный ранее с помощью функции **socket**.

**Задание 12.** Установитепрограмму **ServerT** на другой компьютер локальной сети, а в программу **ClientT** внесите необходимые изменения, позволяющие ей установить связь с сервером. Убедитесь в работоспособности распределенного в локальной сети приложения **ClientT-ServеrT**.

**Задание 13.** Внеситеизменения в программы  **ClientT** и **ServerT**, позволяющие 1000 раз передать сообщение типа ***Hello from Client xxx*** (***xxx*** *–* номер сообщения) от клиента серверу. Убедитесь в работоспособности в сети.

**Задание 14.**  Доработайте программы  **ClientT** и **ServerT** таким образом, чтобы программа **ServerT**  принимала последовательность сообщений вида ***Hello from Client xxx*** от программы **ClientT** и отправляла их без изменения обратно программе **ClientT**. Программа **ClientT,** получив вернувшееся сообщение, должна увеличить в нем счетчик ***xxx*** на единицу  и вновь направить в адрес **ServerT**.Количество передаваемых сообщений введите через консоль программы **ClientT**. Условием окончания работы для программы **ServerT**  является получение сообщения нулевой длины. Оцените время обмена 1000 сообщениями (с помощью функций **clock**) между **ClientT** и **ServerT** через локальную сеть. Запустите утилиту **netstat**  на компьютерах клиента и сервера, проанализируйте отчет и найдите информацию о приложении **ClientT- ServerT**.

**Задание 15.**  Доработайте программу  **ServerT** таким образом, чтобы отключения клиента она могла снова установить соединение с другим клиентом и продолжила свою работу.

ТЕОРИЯ:  
**3.1. Предисловие к главе**

Как уже отмечалось раньше, в основе интерфейса Windows Sockets лежит интерфейс сокетов BSD Unix и стандарт POSIX, определяющий взаимодействие прикладных программ с операционной системой.

Интерфейс Windows Sockets акцентирован, прежде всего, на работу в сети TCP/IP, но обеспечивает обмен данными и по некоторым другим протоколам, например, ***IPX/SPX*** (стек протоколов операционной системы NetWare, компании Novell).

Ниже будет рассказано, как составлять сетевые приложения на языке программирования C++ с использованием интерфейса Windows Socket для протокола TCP/IP.

**3.2. Версии, структура и состав интерфейса Windows Sockets**

Существует две основные версии интерфейса: Windows Sockets 1.1 и Windows Sockets 2. В состав каждой версии входит динамическая библиотека, библиотека экспорта и заголовочный файл, необходимый для работы с библиотеками. Интерфейс версии 1.1 имеет две реализации: для 16-битовых и 32-битовых приложений.

Дальнейшее изложение интерфейса Windows Sockets ориентировано на версию 2, которую далее для краткости будем называть просто Winsock2. Полное описание функций Winsock2 содержится в документации, которая поставляется в составе SDK (Software Developer Kit) для программного интерфейса WIN32 или в MSDN.

Для использования интерфейса Winsock2 в исходный текст программы следует включить следующую последовательность директив компилятора C++.

//........................................................................

**#include "Winsock2.h"** // заголовок WS2\_32.dll

**#pragma comment(lib, "WS2\_32.lib")** // экспорт WS2\_32.dll

//........................................................................

Динамическая библиотека WS2\_32.DLL (которая содержит все функции Winsock2), входит в стандартную поставку Windows, а библиотека экспорта WS2\_32.LIB и заголовочный файл Winsock2.h в стандартную поставку Visual C++. С принципами построения и использования динамических библиотек (DLL) можно ознакомиться в [4, 12].

В таблице 3.2.1 приведен список функций интерфейса Windows. Список включает не все функции Winsock2 – здесь перечислены, только функции, которые будут применяться в дальнейших примерах. Кроме того, описания этих функций, не будет полным. Описания предназначены только для решения рассматриваемых в пособии задач. С полным описанием можно ознакомиться, например, на сайте [www.microsoft.com](http://www.microsoft.com).

Таблица 3.2.1

|  |  |
| --- | --- |
| **Наименование**  **функции** | **Назначение** |
| **accept** | Разрешить подключение к сокету |
| **bind** | Связать сокет с параметрами |
| **closesocket** | Закрыть существующий сокет |
| **connect** | Установить соединение с сокетом |
| **gethostbyaddr** | Получить имя хоста по его адресу |
| **gethostbyname** | Получить адрес хоста по его имени |
| **gethostname** | Получить имя хоста |
| **getsockopt** | Получить текущие опции сокета |
| **htonl** | Преобразовать u\_long в формат TCP/IP |
| **htons** | Преобразовать u\_short в формат TCP/IP |
| **inet\_addr** | Преобразовать символьное представление IPv4-адреса в формат TCP/IP |
| **inet\_ntoa** | Преобразовать сетевое представление IPv4-адреса в символьный формат |
| **ioctlsocket** | Установить режим ввода-вывода сокета |
| **listen** | Переключить сокет в режим прослушивания |
| **ntohl** | Преобразовать в u\_long из формата TCP/IP |
| **ntohs** | Преобразовать в u\_short из формата TCP/IP |
| **recv** | Принять данные по установленному каналу |
| **recvfrom** | Принять сообщение |
| **send** | Отправить данные по установленному каналу |
| **sendto** | Отправить сообщение |
| **setsockopt** | Установит опции сокета |
| **socket** | Создать сокет |
| **TransmitFile** | Переслать файл |
| **TransmitPackets** | Переслать область памяти |
| **WSACleanup** | Завершить использование библиотеки WS2\_32.DLL |
| **WSAGetLastError** | Получить диагностирующий код ошибки |
| **WSAStartup** | Инициализировать библиотеку WS2\_32.DLL |

**3.3. Коды возврата функций интерфейса Windows Sockets**

Все функции интерфейса Winsock2 могут завершаться успешно или с ошибкой. При описании каждой функции будет указано, каким образом можно проверить успешность ее завершения. В том случае, если функция завершает свою работу с ошибкой, формируется дополнительный диагностирующий код, позволяющий уточнить причину ошибки.

Диагностирующий код может быть получен с помощью функции WSAGetLastError. Функция WSAGetLastError вызывается, непосредственно сразу после функции Winsock2, завершившейся с ошибкой. Все диагностирующие коды представлены в таблице 3.3.1. Описание функции приводится на рисунке 3.3.1. На рисунке 3.3.2 приведен пример использования функции WSAGetLastError.

**// -- Получить диагностирующий код ошибки**

**// Назначение:** функция позволяет определить причину

**//** завершения функций Winsock2 с ошибкой

**int WSAGetLastError(void);** // прототип функции

**// Код возврата:** функция возвращает диагностический код

Рисунок 3.3.1. Функция WSAGetLastError

//...................................................................

**#include "Winsock2.h"**

**#pragma comment(lib, "WS2\_32.lib")**

//...................................................................

**string GetErrorMsgText(int code)** // cформировать текст ошибки

**{**

**string msgText;**

**switch (code)** // проверка кода возврата

**{**

**case WSAEINTR: msgText = "WSAEINTR"; break;**

**case WSAEACCES: msgText = "WSAEACCES"; break;**

//..........коды WSAGetLastError ..........................

**case WSASYSCALLFAILURE: msgText = "WSASYSCALLFAILURE";break;**

**default: msgText = "\*\*\*ERROR\*\*\*"; break;**

**};**

**return msgText;**

**};**

**string SetErrorMsgText(string msgText, int code)**

**{return msgText+GetErrorMsgText(code);};**

**int main(int argc, \_TCHAR\* argv[])**

**{**

//............................................................

**try**

**{**

//............................................................

**if ((sS = socket(AF\_INET, SOCK\_STREAM, NULL))== INVALID\_SOCKET)**

**throw SetErrorMsgText("socket:",WSAGetLastError());**

//.............................................................

**}**

**catch (string errorMsgText)**

**{ cout<< endl << "WSAGetLastError: " << errorMsgText;}**

//................................................................

**return 0;**

**}**

Рисунок 3.3.2. Пример использования функции WSAGetLastError

В приведенном примере для обработки ошибок используется функция SetErrorMsgText, которая в качестве параметра получает префикс формируемого сообщения об ошибке, код функции WSAGetLastError, а возвращает текст сообщения (используя функцию GetErrorMsgText).

Таблица 3.3.1

|  |  |
| --- | --- |
| **Коды возврата**  **функции GetLastError** | **Причина ошибки** |
| **WSAEINTR** | Работа функции прервана |
| **WSAEACCES** | Разрешение отвергнуто |
| **WSAEFAULT** | Ошибочный адрес |
| **WSAEINVAL** | Ошибка в аргументе |
| **WSAEMFILE** | Слишком много файлов открыто |
| **WSAEWOULDBLOCK** | Ресурс временно недоступен |
| **WSAEINPROGRESS** | Операция в процессе развития |
| **WSAEALREADY** | Операция уже выполняется |
| **WSAENOTSOCK** | Сокет задан неправильно |
| **WSAEDESTADDRREQ** | Требуется адрес расположения |
| **WSAEMSGSIZE** | Сообщение слишком длинное |
| **WSAEPROTOTYPE** | Неправильный тип протокола для сокета |
| **WSAENOPROTOOPT** | Ошибка в опции протокола |
| **WSAEPROTONOSUPPORT** | Протокол не поддерживается |
| **WSAESOCKTNOSUPPORT** | Тип сокета не поддерживается |
| **WSAEOPNOTSUPP** | Операция не поддерживается |
| **WSAEPFNOSUPPORT** | Тип протоколов не поддерживается |
| **WSAEAFNOSUPPORT** | Тип адресов не поддерживается протоколом |
| **WSAEADDRINUSE** | Адрес уже используется |
| **WSAEADDRNOTAVAIL** | Запрошенный адрес не может быть использован |
| **WSAENETDOWN** | Сеть отключена |
| **WSAENETUNREACH** | Сеть не достижима |
| **WSAENETRESET** | Сеть разорвала соединение |
| **WSAECONNABORTED** | Программный отказ связи |
| **WSAECONNRESET** | Связь восстановлена |
| **WSAENOBUFS** | Не хватает памяти для буферов |
| **WSAEISCONN** | Сокет уже подключен |
| **WSAENOTCONN** | Сокет не подключен |
| **WSAESHUTDOWN** | Нельзя выполнить send: сокет завершил работу |
| **WSAETIMEDOUT** | Закончился отведенный интервал времени |
| **WSAECONNREFUSED** | Соединение отклонено |
| **WSAEHOSTDOWN** | Хост в неработоспособном состоянии |
| **WSAEHOSTUNREACH** | Нет маршрута для хоста |
| **WSAEPROCLIM** | Слишком много процессов |

Таблица 3.3.1 (продолжение)

|  |  |
| --- | --- |
| **Коды возврата**  **функции GetLastError** | **Причина ошибки** |
| **WSASYSNOTREADY** | Сеть не доступна |
| **WSAVERNOTSUPPORTED** | Данная версия недоступна |
| **WSANOTINITIALISED** | Не выполнена инициализация WS2\_32.DLL |
| **WSAEDISCON** | Выполняется отключение |
| **WSATYPE\_NOT\_FOUND** | Класс не найден |
| **WSAHOST\_NOT\_FOUND** | Хост не найден |
| **WSATRY\_AGAIN** | Неавторизированный хост не найден |
| **WSANO\_RECOVERY** | Неопределенная ошибка |
| **WSANO\_DATA** | Нет записи запрошенного типа |
| **WSA\_INVALID\_HANDLE** | Указанный дескриптор события с ошибкой |
| **WSA\_INVALID\_PARAMETER** | Один или более параметров с ошибкой |
| **WSA\_IO\_INCOMPLETE** | Объект ввода-вывода не в сигнальном состоянии |
| **WSA\_IO\_PENDING** | Операция завершится позже |
| **WSA\_NOT\_ENOUGH\_MEMORY** | Не достаточно памяти |
| **WSA\_OPERATION\_ABORTED** | Операция отвергнута |
| **WSAINVALIDPROCTABLE** | Ошибочный сервис |
| **WSAINVALIDPROVIDER** | Ошибка в версии сервиса |
| **WSAPROVIDERFAILEDINIT** | Невозможно инициализировать сервис |
| **WSASYSCALLFAILURE** | Аварийное завершение системного вызова |

Комментарии с точками в приведенном примере и дальше будут использоваться для обозначения того, что тексты программ не являются законченными и предназначены только для демонстрации использования функций.

**3.4. Схемы взаимодействия процессов в распределенном приложении**

Существование двух различных протоколов на транспортном уровне TCP/IP, определяет две схемы взаимодействия процессов распределенного приложения: схема, ориентированная на сообщения, и схема, ориентированная на поток.

Принципиальное различие этих схем, заключается в следующем.

В первом между сокетами курсируют UDP-пакеты, и поэтому вся работа, связанная с обеспечением надежности и установкой правильной последовательности передаваемых пакетов возлагается на само приложение. В общем случае, получатель узнает адрес отравителя вместе с пакетом данных.

Во втором случае между сокетами устанавливается TCP-соединение и весь обмен данных осуществляется в рамках этого соединения. Передача по каналу является надежной и данные поступают в порядке их отправления.

В распределенных приложениях архитектуры клиент-сервер, клиенту и серверу отводится разная роль: инициатором обмена является клиент, а сервер ждет запросы клиента и обслуживает их. Таким образом, предполагается, что к моменту выдачи запроса клиентом, сервер должен быть уже активным, а клиент должен “знать” параметры сокета сервера. На рисунках 3.4.1 и 3.4.2. изображены схемы взаимодействия клиента и сервера, для первого и второго случаев.



Рисунок 3.4.1. Схема взаимодействия процессов без установки соединения

На рисунке 3.4.1 схематично изображены две программы, реализующие два процесса распределенного приложения. Рассматриваемое приложение имеет архитектуру клиент-сервер (на рисунке сделаны соответствующие обозначения). Обе программы разбиты на пять блоков, а стрелками обозначается движение информации по сети TCP/IP.

Первые блоки обеих программ одинаковые и предназначены для инициализации библиотеки WS2\_32.DLL.

Второй блок программы сервера создает сокет (функция soсket) и устанавливает параметры этого сокета. Следует обратить внимание на параметр SOCK\_DGRAM функции socket, указывающий на тип сокета (в данном случае – сокет, ориентированный на сообщения). Для установки параметров сокета, используется функция bind. При этом говорят, что сокет ***связывают*** с параметрами. Для хранения параметров сокета в Winsock2 предусмотрена специальная структура SOCKADDR\_IN (она тоже присутствует на рисунке). Перед выполнением функции bind, которая использует эту структуру в качестве параметра, необходимо ее заполнить данными. Пока скажем только, что в SOCKADDR\_IN хранится IP-адрес и номер порта сервера.

В третьем блоке программы сервера выполняется функция recvfrom, которая переводит программу сервера в состояние ожидания, до поступления сообщения от программы клиента (функция sendto). Функция recvfrom тоже использует структуру SOCKADDR\_IN – в нее автоматически помещаются параметры сокета клиента, после приема от него сообщения. Данные поступают в буфер, который обеспечивает принимающая сторона (на рисунке символьный массив bfrom). Следует отметить, что в качестве параметра функции recvfrom используется связанный сокет – именно через него осуществляется передача данных.

Четвертый блок программы сервера предназначен для пересылки данных клиенту. Пересылка данных осуществляется с помощью функции sendto. В качестве параметров sendto использует структуру SOCKADDR\_IN с параметрами сокета принимающей стороны (в данном случае клиента) и заполненный буфер с данными.

Пятые блоки программ сервера и клиента одинаковые и предназначены для закрытия сокета и завершения работы с библиотекой WS2\_32.DLL.

Всем блокам программы клиента, кроме второго, есть аналог в программе сервера. Второй блок, в сравнении с сервером, не использует команду bind. Здесь проявляется основное отличие между сервером и клиентом. Если сервер, должен использовать однозначно определенные параметры (IP-адрес и номер порта), то для клиента это не обязательно – ему Windows выделяет эфемерный порт. Т.к. инициатором связи является клиент, то он должен точно “знать” параметры сокета сервера, а свои параметры клиент получит от Windows и сообщит их вместе с переданным пакетом серверу.

Взаимодействие программ клиента и сервера в случае установки соединения схематично изображено на рисунке 3.4.2. Как и в предыдущем случае обе программы разбиты на блоки. Сплошными направленными линиями обозначается движение данных по сети TCP/IP, прерывистой – синхронизация (ожидание) процессов.

Первые блоки обеих программ идентичны и предназначены для инициализации библиотеки WS2\_32.DLL.

Второй блок сервера имеет то же предназначение, что и в предыдущем случае. Единственным отличием является значение SOCK\_STREEM параметра функции socket, указывающий, что сокет будет использоваться для соединения (сокет ориентированный на поток).

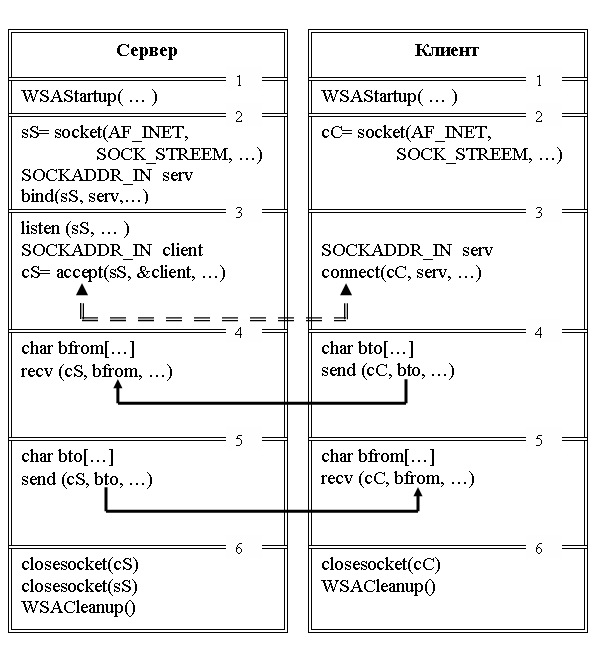
****

Рисунок 3.4.2. Схема взаимодействия процессов с установкой соединения

В третьем блоке программы сервера выполняются две функции Winsock2: listen и accept. Функция listen переводит сокет, ориентированный на поток, в состояния прослушивания (открывает доступ к сокету) и задает некоторые параметры очереди соединений. Функция accept переводит процесс сервера в состояние ожидания, до момента пока программа клиента не выполнит функцию connect (подключится к сокету). Если на стороне клиента корректно выполнена функция connect, то функция accept возвращает новый сокет (с эфемерным портом), который предназначен для обмена данными с подключившимся клиентом. Кроме того, автоматически заполняется структура SOCKADDR\_IN параметрами сокета клиента.

Четвертый и пятый блоки программы сервера предназначены для обмена данными по созданному соединению. Следует обратить внимание, что, во-первых, используются функции send и recv, а во-вторых, в качестве параметра эти функции используют сокет, созданный командой accept.

В программе клиента осталось пояснить, только работу третьего блока. В этом блоке выполняется функция connect, предназначенная для установки соединения с сокетом сервера. Функция в качестве параметров имеет, созданный в предыдущем блоке, дескриптор сокета (ориентированного на поток) и структуру SOCKADDR\_IN с параметрами сокета сервера.

**3.5. Инициализация библиотеки Windows Sockets**

Для инициализации библиотеки WS2\_32.DLL предназначена функция WSAStartup. Описание функции приводится на рисунке 3.5.1.

// -- **инициализировать библиотеку WS2\_32.DLL**

// **Назначение**: функция позволяет инициализировать

// динамическую библиотеку, проверить номер

// версии, получить сведения о конкретной

// реализации библиотеки. Функция должна быть

// выполнена до использования любой функции

// Windows Sockets

//

**int WSAStartup(**

**WORD ver,** //[in] версия Windows Sockets

**lpWSAData wsd** //[out] указатель на WSADATA

**);**

// **Код возврата:** в случае успешного завершения функция

// возвращает нулевое значение, в случае ошибки

// возвращается не нулевое значение

// **Примечания:** - параметр **ver** представляет собой два байта,

// содержащих номер версии Windows Sockets,

// причем. старший байт содержит

// младший номер версии, а младший байт-

// старший номер версии;

// - обычно параметр **ver** задается с помощью макро // M**AKEWORD**;

// - шаблон структуры **WSADATA** содержится в

// Winsock2.h

Рисунок 3.5.1. Функция WSAStartup

Как уже отмечалось раньше, функция WSAStartup должна быть выполнена до использования любых функция Winsock2. Пример, использования функции будет приведен ниже.

**3.6. Завершение работы с библиотекой Windows Sockets**

Для завершения работы с библиотекой WS2\_32.DLL используется функция WSACleanup. Описание функции приводится на рисунке 3.6.1.

// -- **завершить работу с библиотекой WS2\_32.DLL**

// **Назначение**: функция завершает работу с динамической

// библиотекой **WS2\_32.DLL**, делает недоступным

// выполнение функций библиотеки, освобождает

// ресурсы.

//

**int WSACleanup(void);**

// **Код возврата:** в случае успешного завершения функция

// возвращает нулевое значение, в случае ошибки

// возвращается **SOCKET\_ERROR**

Рисунок 3.6.1. Функция WSACleanup

На рисунке 3.6.2 приводится пример использования функций WSAStatrup и WSACleanup.

//...........................................................

**#include "Winsock2.h"**

**#pragma comment(lib, "WS2\_32.lib")**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**WSADATA wsaData;**

**try**

**{**

**if (WSAStartup(MAKEWORD(2,0), &wsaData) != 0)**

**throw SetErrorMsgText("Startup:",WSAGetLastError());**

//...........................................................

**if (WSACleanup()== SOCKET\_ERROR)**

**throw SetErrorMsgText("Cleanup:",WSAGetLastError());**

**}**

**catch (string errorMsgText)**

**{ cout<< endl << errorMsgText;}**

**return 0;**

**}**

Рисунок 3.6.2. Пример использования функций WSAStartup и WSACleanup

**3.7. Создание сокета и закрытие сокета**

Для создания сокета используется функция socket. Описание функции приводится на рисунке 3.7.1.

// -- **создать сокет**

// **Назначение**: функция позволяет создать сокет (точнее

// дескриптор сокета) и задать его характеристики

//

**SOCKET socket(**

**int af,** //[in] формат адреса

**int type,** //[in] тип сокета

**int prot** //[in] протокол

**);**

// **Код возврата:** в случае успешного завершения функция

// возвращает дескриптор сокета, в другом

// случае возвращается **INVALID\_SOCKET**

// **Примечания:** - параметр **af** для стека TCP/IP принимает

// значение **AF\_INET**;

// - параметр **type** может принимать два значения:

// **SOCK\_DGRAM** – сокет, ориентированный на

// сообщения(UDP); **SOCK\_STREEM** – сокет

// ориентированный на поток;

// старший номер версии;

// - параметр **prot** определяет протокол

// транспортного уровня: для TCP/IP можно

// указать NULL

Рисунок 3.7.1. Функция socket

После завершения работы с сокетом, обычно, его закрывают (освобождают ресурс). Для закрытия сокета применяется функция closesocket. Описание этой функции приводится на рисунке 3.7.2.

// -- **закрыть существующий сокет**

// **Назначение**: переводит сокет в неработоспособное состояние и

// освобождает все ресурсы связанные с ним

//

**SOCKET closesocket(**

**SOCKET s,** //[in] дескриптор сокета

**);**

// **Код возврата:** в случае успешного завершения функция

// возвращает нуль, в другом случае

// возвращается **SOCKET\_ERROR**

Рисунок 3.7.2. Функция closesocket

На рисунке 3.7.3. приводится пример программы использующей функции socket и closesocket.

//................................................................

**#include "Winsock2.h"**

**#pragma comment(lib, "WS2\_32.lib")**

**int \_tmain(int argc, \_TCHAR\* argv[])**

**{**

**SOCKET sS;** // дескриптор сокета

**WSADATA wsaData;**

**try**

**{**

**if (WSAStartup(MAKEWORD(2,0), &wsaData) != 0)**

**throw SetErrorMsgText("Startup:",WSAGetLastError());**

**if ((sS = socket(AF\_INET, SOCK\_STREAM, NULL))== INVALID\_SOCKET)**

**throw SetErrorMsgText("socket:",WSAGetLastError());**

//...........................................................

**if (closesocket(sS)== SOCKET\_ERROR)**

**throw SetErrorMsgText("closesocket:",WSAGetLastError());**

**if (WSACleanup()== SOCKET\_ERROR)**

**throw SetErrorMsgText("Cleanup:",WSAGetLastError());**

**}**

**catch (string errorMsgText)**

**{ cout << endl << errorMsgText;}**

**return 0;**

**}**

Рисунок 3.7.3. Пример использования функций socket и closesocket

**3.8. Установка параметров сокета**

Для установки параметров существующего сокета используется функция bind. Описание функции приводится на рисунке 3.8.1.

// -- **связать сокет с параметрами**

// **Назначение**: функция связывает существующий сокет с

// с параметрами, находящимися в структуре

// SOCKADDR\_IN

//

**int bind(**

**SOCKET s,** //[in] сокет

**cost struct sockaddr\_in\* a,** //[in]указатель на **SOCKADDR\_IN**

**int la** //[in] длина **SOCKADDR\_IN** в байтах

**)**

// **Код возврата:** в случае успешного завершения функция

// возвращает нуль, в случае ошибки

// возвращается **SOCKET\_ERROR**

Рисунок 3.8.1. Функция bind

Функция связывает дескриптор сокета и структуру SOCKADDR\_IN, которая предназначена для хранения параметров сокета. IШаблон структуры SOCKADDR\_IN содержится в файле Winsock2.h. Описание структуры SOCKADDR\_IN и используемых вместе с ней констант приводится на рисунке 3.8.2. Особое внимание следует обратить внимания на строки, отмеченные тремя знаками плюс. В дальнейшем отмеченные поля и константы будут использоваться в текстах программ. IP-адрес и номер порта в структуре SOCKADDR\_IN хранятся в специальном сетевом формате. Этот формат отличается, от формата компьютеров с архитектурой Intel. В составе Winsock2 имеются функции, позволяющие преобразовывать форматы данных.

**#define INADDR\_ANY (u\_long)0x00000000** //любой адрес **+++**

**#define INADDR\_LOOPBACK 0x7f000001** // внутренняя петля **+++**

**#define INADDR\_BROADCAST (u\_long)0xffffffff** // широковещание **+++**

**#define INADDR\_NONE 0xffffffff** // нет адреса

**#define ADDR\_ANY INADDR\_ANY** // любой адрес

**struct in\_addr**

**{** // IP-адрес

**union {**

**struct { u\_char s\_b1,s\_b2,s\_b3,s\_b4; } S\_un\_b;**

**struct { u\_short s\_w1,s\_w2; } S\_un\_w;**

**u\_long S\_addr;**

**} S\_un;**

**#define s\_addr S\_un.S\_addr** // 32-битный IP-адрес **+++**

**#define s\_host S\_un.S\_un\_b.s\_b2**

**#define s\_net S\_un.S\_un\_b.s\_b1**

**#define s\_imp S\_un.S\_un\_w.s\_w2**

**#define s\_impno S\_un.S\_un\_b.s\_b4**

**#define s\_lh S\_un.S\_un\_b.s\_b3**

**}**

**struct sockaddr\_in {**

**short sin\_family;** //тип сетевого адреса **+++**

**u\_short sin\_port;** // номер порта +++

**struct in\_addr sin\_addr;** // IP-адрес **+++**

**char sin\_zero[8];** // резерв

**};**

**typedef struct sockaddr\_in SOCKADDR\_IN;** // **+++**

**typedef struct sockaddr\_in \*PSOCKADDR\_IN;**

**typedef struct sockaddr\_in FAR \*LPSOCKADDR\_IN;**

Рисунок 3.8.2. Структура SOCKADDR\_IN

Для преобразования номера порта в формат TCP/IP следует использовать функцию htons. Описание этой функции приведено на рисунке 3.8.3. Функция ntohs является обратной функцией, предназначена для преобразования двух байтов в формате TCP/IP в формат u\_short.

// -- **преобразовать u\_short в формат TCP/IP**

// **Назначение**: функция преобразовывает два байта данных

// формата u\_short (unsigned short) в два

// два байта, сетевого формата

//

**u\_short htons (**

**u\_short hp** //[in] 16 битов данных

**);**

//

// **Код возврата:** 16 битов в формате TCP/IP

//

Рисунок 3.8.3 Функция htons

Полезной является функция inet\_addr, предназначенная для преобразования символьного представления IPv4-адреса в формат TCP/IP. Описание функции приведено на рисунке 3.8.4. Функция inet\_ntoa предназначена для обратного преобразования из сетевого представления в символьный формат.

// -- **преобразовать символьное представление IPv4-адреса в формат TCP/IP**

// **Назначение**: функция преобразует общепринятое символьное

// представление IPv4-адреса (n.n.n.n) в

// четырехбайтовый IP-адрес в формате TCP/IP

//

**unsigned long** **inet\_addr(**

**const char\* stra** //[in] строка символов, закачивающаяся 0x00

**);**

//

// **Код возврата:** в случае успешного завершения функция

// IP-адрес в формате TCP/IP, иначе

// возвращается **INADDR\_NONE**

Рисунок 3.8.4 Функция inet\_addr

На рисунке 3.8.5 приведен фрагмент программы сервера. Функция bind связывает сокет с параметрами, заданными в структуре SOCKADDR\_IN. Структура содержит три значения (параметры сокета): тип используемого адреса (константа AF\_INET используется для обозначения семейства IP-адресов); номер порта (устанавливается значение 2000 с помощью функции htons) и адрес интерфейса . Последний параметр определяет собственный IP-адрес сервера. При этом предполагается, что хост, в общем случае, может иметь несколько IP-интерфейсов. Если требуется использовать определенный IP-интерфейс хоста, то необходимо его здесь указать. Если выбор IP-адреса не является важным или IP-интерфейс один на хосте, то следует указать значение INADDR\_ANY (как это сделано в примере). Программа клиента для пересылки сообщений (обратите внимание, что при создании сокета использовался параметр со значением SOCKET\_DGRAM), должна их отправлять именно этому сокету (т.е. указывать его IP-адрес и его номер порта).

//................................................................

**SOCKET sS;** // серверный сокет

**if ((sS = socket(AF\_INET, SOCK\_DGRAM, NULL))== INVALID\_SOCKET)**

**throw SetErrorMsgText("socket:",WSAGetLastError());**

**SOCKADDR\_IN serv;** // параметры сокета sS

**serv.sin\_family = AF\_INET;** // используется IP-адресация

**serv.sin\_port = htons(2000);** // порт 2000

**serv.sin\_addr.s\_addr = INADDR\_ANY;** // любой собственный IP-адрес

**if (bind(sS,(LPSOCKADDR)&serv, sizeof(serv))== SOCKET\_ERROR)**

**throw SetErrorMsgText("bind:",WSAGetLastError());**

//................................................................

Рисунок 3.8.5. Пример использования функции bind

**3.9. Переключение сокета в режим прослушивания**

После создания сокета и выполнения функции bind сокет остается недоступным для подсоединения клиента. Чтобы сделать доступным уже связанный сокет, необходимо его переключить, в так называемый, прослушивающий режим. Переключение осуществляется с помощью функции listen (рисунок 3.9.1).

// **-- переключить сокет в режим прослушивания**

// **Назначение:** функция делает сокет доступным для подключений

//и устанавливаетмаксимальную длину очереди

//подключений

**int listen(**

**SOCKET s,** //[in] дескриптор связанного сокета

**int mcq,** //[in] максимальная длина очереди

**);**

// **Код возврата:** при успешном завершении функция возвращает

// нуль, иначе возвращается значение

// **SOCKET\_ERROR**

// **Примечания**: для установки значения параметра **mcq** можно

// использовать константу **SOMAXCONN**,позволяющую

// установить максимально возможное значение

Рисунок 3.9.1. Функции listen

После выполнения функции listen клиентские программы могут осуществить подключение к сокету (выполнить функцию connect). Кроме того, функция listen устанавливает максимальную длину очереди подключений. Если количество одновременно подключающихся клиентов превысит установленное максимальное значение, то последние попытки подключения потерпят неудачу (и функция connect на стороне клиента сформирует соответствующий код ошибки). Следует отметить, что функция listen применяется только для сокетов ориентированных на поток.

**3.10. Создание канала связи**

Канал связи (или соединение) создается между двумя сокетами, ориентированными на поток. На стороне сервера это должен быть связанный (функция bind) и переключенный в режим прослушивания (функция listen) сокет. На стороне клиента должен быть создан дескриптор ориентированного на поток сокета (функция socket).

Канал связи создается в результате взаимодействия функций accept (на стороне сервера) и connect (на стороне клиента). Алгоритм взаимодействия этих функций зависит от установленного режима ввода-вывода для участвующих в создании канала сокетов.

Winsock2 поддерживает два режима ввода вывода: blocked и nonblocked. Установить или изменить режим можно с помощью функции ioctlsocket. Дальнейшее изложение предполагает, что для сокетов установлен режим blocked (действующий по умолчанию), режим nonblocked будет рассматриваться отдельно.

// **-- разрешить подключение к сокету**

// **Назначение:** функция используется для создания канала на

//стороне сервера и создает сокет для обмена

//данными по этому каналу

**SOKET accept(**

**SOCKET s,** // [in] дескриптор связанного сокета

**struct sockaddr\_in\* a,**//[out] указатель на **SOCKADDR\_IN**

**int\* la** //[out] указатель на длину **SOCKADDR\_IN**

**);**

// **Код возврата:** при успешном завершении функция возвращает

// дескриптор нового сокета, предназначенного

// для обмена данными по этому каналу, иначе

// возвращается значение **INVALID\_SOCKET**

// **Примечания**: в случае успешного выполнения функции,

// указатель **a** содержит адрес структуры

// **SOCKADDR\_IN** с параметрами сокета,

// осуществившего подключение (connect)сокета,

// a указатель **la** содержит адрес 4-х байт с

// длинной (в байтах) структуры **SOCKADDR\_IN**

Рисунок 3.10.1 Функция accept

Функция accept (описание на рисунке 3.10.1) приостанавливает выполнение программы сервера до момента срабатывания в программе клиента функции connect (описание на рисунке 3.10.3). В результате работы функции accept создается новый сокет, предназначенный для обмена данными с клиентом. На рисунке 3.10.2 представлен пример использования функции accept.

//........................................................................

**try**

**{**

//...**WSAStartup(...),sS = socket(...,SOCKET\_STREAM,...),bind(sS,...)**

**if (listen(sS,SOMAXCONN)== SOCKET\_ERROR)**

**throw SetErrorMsgText("listen:",WSAGetLastError());**

**SOCKET cS;**  // сокет для обмена данными с клиентом

**SOCKADDR\_IN clnt;**  // параметры сокета клиента

**memset(&clnt,0,sizeof(clnt));** // обнулить память

**int lclnt = sizeof(clnt);** // размер SOCKADDR\_IN

**if ((cS = accept(sS,(sockaddr\*)&clnt, &lclnt)) == INVALID\_SOCKET)**

**throw SetErrorMsgText("accept:",WSAGetLastError());**

//.................................................................

**}**

**catch (string errorMsgText)**

**{cout << endl << errorMsgText;}**

//........................................................................

Рисунок 3.10.2 Фрагмент программы сервера с функцией accept

// **-- установить соединение с сокетом**

// **Назначение:** функция используется клиентом для создания

// канала с определенным сокетом сервера

**int connect (**

**SOCKET s,** // [in] дескриптор связанного сокета

**struct sockaddr\_in\* a,** // [in] указатель на **SOCKADDR\_IN**

**int la** // [in] длина **SOCKADDR\_IN** в байтах

**);**

// **Код возврата:** при успешном завершении функция возвращает

// нуль, иначе возвращается значение

// **SOCKET\_ERROR**

// **Примечания**: - параметр **a** является указателем на структуру

// **SOCKADDR\_IN;** структура должна быть

// инициализирована параметрами серверного

// сокета (тип адреса, IP-адрес, порт);

// - параметр **la**, должен содержать длину

// (в байтах) структуры **SOCKADDR\_IN**

Рисунок 3.10.3 Функция connect

На стороне клиента создание канала осуществляется с помощью функции connect. Для того, чтобы выполнить функцию connect, достаточно просто предварительно создать сокет (функция socket), ориентированный на поток. Функция connect указывает модулю TCP сокет клиента, который будет использоваться для соединения с сокетом сервера (его параметры указываются через параметры connect). При этом предполагается, что серверный сокет создан (функции socket и bind) и для него уже выполнена функция listen. На рисунке 3.10.4 приведен фрагмент текста программы клиента, в котором используется функция connect.

//................................................................

**try**

**{**

//....**WSAStartup(...)**...........................................

**SOCKET cC;** // серверный сокет

**if ((cC = socket(AF\_INET, SOCK\_STREAM, NULL))== INVALID\_SOCKET)**

**throw SetErrorMsgText("socket:",WSAGetLastError());**

**SOCKADDR\_IN serv;** // параметры сокета сервера

**serv.sin\_family = AF\_INET;** // используется IP-адресация

**serv.sin\_port = htons(2000);** // TCP-порт 2000

**serv.sin\_addr.s\_addr = inet\_addr("80.1.1.7");** // адрес сервера

**if ((connect(cC,(sockaddr\*)&serv, sizeof(serv))) == SOCKET\_ERROR)**

**throw SetErrorMsgText("connect:",WSAGetLastError());**

//..............................................................

**}**

**catch (string errorMsgText)**

**{ cout << endl << errorMsgText;}**

//..............................................................

Рисунок 3.10.4. Фрагмент программы клиента с функцией connect

**3.11. Обмен данными по каналу связи**

Обмен данными по каналу связи осуществляется между двумя сокетами и возможен сразу после того, как этот канал создан (выполнена функция accept на стороне сервера и функция connect на стороне клиента). Для пересылки данных по каналу Winsock2 предоставляет функции send и recv (рисунки 3.11.1 и 3.11.2). Функция send пересылает по каналу, указанного сокета, определенное количество байт данных. Функция recv принимает по каналу, указанного сокета, определенное количество байт данных. Для работы обеих функций в программе необходимо выделить память (буфер) для приема или оправления данных. Размер буфера для приема данных и для отправления данных указывается в параметрах функций. Реальное количество пересланных или принятых байт данных возвращается функциями send и recv в виде кода возврата.

Следует иметь в виду, что не всегда количество переданных или оправленных байт совпадает с размерами выходного или входного буферов. Более того, разрешается выполнять пересылку с нулевым количеством байт. Обычно такую пересылку используют для обозначения конца передачи. Для принимающей стороны операционная система буферизирует принимаемые данные. Если при очередном приеме данных размеры буфера будут исчерпаны, отправляющей стороне будет выдано соответствующий код ошибки.

// **-- отправить данные по установленному каналу**

// **Назначение:** функция пересылает заданное количество

// байт данных по каналу определенного сокета

**int send (**

**SOCKET s,** // [in] дескриптор сокета (канал для передачи)

**const char\* buf,** // [in] указатель буфер данных

**int lbuf,** // [in] количество байт данных в буфере

**int flags** // [in] индикатор особого режима маршрутизации

**);**

// **Код возврата:** при успешном завершении функция возвращает

// количество переданных байт данных, иначе

// возвращается **SOCKET\_ERROR**

// **Примечания**: для параметра **flags** следует установить

// значение **NULL**

Рисунок 3.11.1 Функция send

// **-- принять данные по установленному каналу**

// **Назначение:** функция принимает заданное количество

// байт данных по каналу определенного сокета

**int recv (**

**SOCKET s,** // [in] дескриптор сокета (канал для приема)

**const char\* buf,** // [in] указатель буфер данных

**int lbuf,** // [in] количество байт данных в буфере

**int flags** // [in] индикатор

**);**

// **Код возврата:** при успешном завершении функция возвращает

// количество принятых байт данных, иначе

// возвращается **SOCKET\_ERROR**

// **Примечания**: параметр **flags** определяет режим обработки

// буфера: **NULL** - входной буфер очищается

// после считывания данных (рекомендуется),

// **MSG\_PEEK** – входной буфер не очищается

Рисунок 3.11.2 Функция recv

Работа функций send и recv является синхронной, т.е. до тех пор, пока не будет выполнена пересылка или прием данных выполнение программы приостанавливается. Поэтому, если одной из сторон распределенного будет выдана функция recv для которой нет данных для приема и при этом соединение не разорвано, то это приведет к зависанию программы на некоторое время (называемоe time-out) и завершению функции recv c соответствующим кодом ошибки.

//........................................................................

**try**

**{**

//...**WSAStartup(...),sS = socket(...,SOCKET\_STREAM,...),bind(sS,...)**

//...**listen(sS,...),**  **cS = accept(sS,...)...........................**

**char ibuf[50],** //буфер ввода

**obuf[50]= "sever: принято ";** //буфер вывода

**int libuf = 0,** //количество принятых байт

**lobuf = 0;** //количество отправленных байь

**if ((libuf = recv(cS,ibuf,sizeof(ibuf),NULL)) == SOCKET\_ERROR)**

**throw SetErrorMsgText("recv:",WSAGetLastError());**

**\_itoa(lobuf, obuf+sizeof("sever: принято ")-1,10);**

**if ((lobuf = send(cS,obuf,strlen(obuf)+1,NULL)) == SOCKET\_ERROR)**

**throw SetErrorMsgText("send:",WSAGetLastError());**

//.....................................................................

**}**

**catch (string errorMsgText)**

**{cout << endl << errorMsgText;}**

//........................................................................

Рисунок 3.11.3. Пример использования функций send и recv

На рисунке 3.11.3 приведен пример использования функций send и recv в программе сервера. После создания канала сервер выдал функцию recv ожидающую данные от подсоединившегося клиента. После получения данных от клиента, сервер формирует выходной буфер и отправляет его содержимое в адрес клиента с помощью функции send.

**3.12. Обмен данными без соединения**

Если для передачи данных на транспортном уровне используется протокол UDP, то говорят, об обмене данными без соединения или об обмене данными с помощью сообщений. Для отправки и приема сообщений в Winsock2 используются функции sendto и recvfrom (рисунки 3.12.1 и 3.12.2). При этом предполагается, что сообщения будут курсировать между сокетами ориентированными на сообщения.

Особенностью использования этих функций заключается в том, что протоколом UDP не гарантируется доставка и правильная последовательность приема отправленных сообщений. Весь контроль надежности доставки и правильной последовательностью поступления сообщений возлагается на разработчика приложения. В связи с этим, обмен данными с помощью сообщений используется, в основном, для широковещательных сообщений или для пересылки коротких сообщений, последовательность получения которых не имеет значения.

// **-- отправить сообщение**

// **Назначение:** функция предназначена для отправки сообщения

// без установления соединения

//

**int sendto(**

**SOCKET s,** // [in] дескриптор сокета

**const char\* buf,** // [in] буфер для пересылаемых данных

**int len,** // [in] размер буфера **buf**

**int flags,** // [in] индикатор режима маршрутизации

**const struct sockaddr\* to,** // [in] указатель на **SOCKADDR\_IN**

**int tolen** // [in] длина структуры to

**);**

// **Код возврата:** при успешном завершении функция возвращает

// количество пересланных байт данных, иначе

// возвращается **SOCKET\_ERROR**

// **Примечания:** - функция может применяться только для

// сокетов, ориентированных на сообщения

// (**SOCK\_DGRAM**))

// - параметр **to** указывает на структуру

// **SOCKADDR\_IN** с параметрами сокета получателя;

// - для параметра **flags** рекомендуется установить

// значение **NULL**

Рисунок 3.12.1. Функция sendto

Также как и функции send и recv, функции sendto и recvfrom работают в синхронном режиме, т.е. вызвав, например, функцию recvfrom вызывающая программа не получит управления до момента завершения приема данных.

Следует также обратить внимание, что обе функции используют в качестве параметров структуру SOCKADDR\_IN. В случае выполнения функции send, структура должна содержать параметры сокета получателя. У функция recv структура SOCKADDR\_IN, наоборот, предназначена для получения параметров сокета отправителя.

Часто протокол UDP (и соответственно функции sendto и recvfrom) используется для пересылки сообщений предназначенных для рассылки одного сообщения всем компьютерам сети (широковещательные сообщения). Для этого в параметрах сокета отправляющей стороны используются специальные широковещательные и групповые IP-адреса. Использование и групповых адресов функцией sendto по умолчанию запрещено. Разрешение

// **-- принять сообщение**

// **Назначение:** функция предназначена для получения сообщения

// без установления соединения

**int recvfrom(**

**SOCKET s,** // [in] дескриптор сокета

**char\* buf,** // [out] буфер для получаемых данных

**int len,** // [in] размер буфера **buf**

**int flags,** // [in] индикатор режима маршрутизации

**struct sockaddr\* to,** // [out] указатель на **SOCKADDR\_IN**

**int\* tolen** //[out] указатель на размер to

**);**

// **Код возврата:** при успешном завершении функция возвращает

// количество принятых байт данных, иначе

// возвращается **SOCKET\_ERROR**

// **Примечания:** - функция может применяться только для

// сокетов, ориентированных на сообщения

// (**SOCK\_DGRAM**));

// - параметр **to** указывает на структуру

// **SOCKADDR\_IN** с параметрами сокета отправителя;

// - параметр **tolen** содержит адрес четырех байт, с

// размером структуры **SOCADDR\_IN**

// - для параметра **flags** рекомендуется установить

// значение **NULL**

Рисунок 3.12.2 Функция recvfrom

//........................................................................

**try**

**{** //...**WSAStartup(...),sS = socket(...,SOCKET\_DGRAM,...)**

**SOCKADDR\_IN serv**; // параметры сокета sS

**serv.sin\_family = AF\_INET;** // используется IP-адресация

**serv.sin\_port = htons(2000);** // порт 2000

**serv.sin\_addr.s\_addr = INADDR\_ANY;** // адрес сервера

**if(bind(sS,(LPSOCKADDR)&serv, sizeof(serv))== SOCKET\_ERROR)**

**throw SetErrorMsgText("bind:", WSAGetLastError());**

**SOCKADDR\_IN clnt;** // параметры сокета клиента

**memset(&clnt,0,sizeof(clnt));** // обнулить память

**int lc = sizeof(clnt);**

**char ibuf[50];** //буфер ввода

**int lb = 0;** //количество принятых байт

**if (lb = recvfrom(sS, ibuf, sizeof(ibuf), NULL,**

**(sockaddr\*)&clnt, &lc)) == SOCKET\_ERROR)**

**throw SetErrorMsgText("recv:",WSAGetLastError());**

//................................................................

**}**

**catch (string errorMsgText)**

**{cout << endl << errorMsgText;}**

//........................................................................

Рисунок 3.12.3 Пример использования функции recvfrom в программе сервера

на использование широковещательных устанавливается функцией setsockopt.

На рисунках 3.12.3 и 3.12.4 приведены примеры применения функций recvfrom и sendto, которые используются в программах сервера и клиента соответственно.

//........................................................................

**try**

**{**//...**WSAStartup(...),сС = socket(...,SOCKET\_DGRAM,...)**

**SOCKADDR\_IN serv;** // параметры сокета сервера

**serv.sin\_family = AF\_INET;** // используется ip-адресация

**serv.sin\_port = htons(2000);** // порт 2000

**serv.sin\_addr.s\_addr = inet\_addr("127.0.0.1");** // адрес сервера

**char obuf[50]= "client: I here";** //буфер вывода

**int lobuf = 0;** //количество отправленных

**if ((lobuf = sendto(cC,obuf,strlen(obuf)+1,NULL,**

**(sockaddr\*)&serv, sizeof(serv))) == SOCKET\_ERROR)**

**throw SetErrorMsgText("recv:",WSAGetLastError());**

//............................................................

**}**

**catch (string errorMsgText)**

**{cout << endl << errorMsgText;}**

//........................................................................

Рисунок 3.12.3 Пример использования функции sendto в программе клиента

**3.13. Пересылка файлов и областей памяти**

Интерфейс Winsock2 может быть использован для пересылки файлов и непрерывной области памяти компьютера. Пересылка файлов существляется с помощью функции TransmitFile (рисунок 3.13.1), а пересылку области памяти можно осуществить с помощью функции TransmitPackets (описание этой функции здесь не рассматривается). Следует отметить, что эти функции не поддерживаются интерфейсом сокетов BSD, но активно используются операционной системой Windows для кэширования данных.

Использование функции TransmitFile предполагает существование соединения и наличие доступного и открытого файла данных. Пересылка осуществляется блоками, размер которых указывается в параметрах функции. Прием данных на другой стороне осуществляется с помощью функции recv. Прием данных осуществляется до тех пор пока не разорвется соединение или не поступит пустой блок данных (функция recv вернет нулевое значение).

На рисунке 3.13.2 приведен фрагмент программы использующей функцию TransmitFile. Следует обратить внимание, что пересылаемый файл должен быть открытым с помощью стандартной функции \_open. Кроме того, в качестве параметра функция TransmitFile использует системный

// **-- переслать файл**

// **Назначение:** функция предназначена для пересылки файла

// по установленному соединению

//

**BOOL TransmitFile(**

**SOCKET s,** // [in] дескриптор сокета

**HANDLE hf,** // [in] HANDLE файла

**DWORD nw,** // [in[ общее количество пересылаемых байтов

**DWORD ns,** // [in] размер буфера пересылки

**LPOVERLAPPED po,** // [in] указатель на структуру **OVERLAPPED**

**LPTRANSMIT\_FILE\_BUFFERS pb,** //[in] указатель **TRANSMIT\_FILE\_BUFFERS**

**DWORD flag** // [in] индикатор режим сокета

**)**;

// **Код возврата:** в случае успешного завершения возвращается

// **TRUE**, иначе функция возвращает значение **FALSE**

// **Примечания:** - значение параметра **nw** может иметь значение

// **NULL**, в этом случае пересылается весь файл;

// - значение параметра **ns** может иметь значение

// **NULL**, в этом случае размер буфера пересылки

// устанавливается по умолчанию;

// - структура **OVERLAPPED** используется для

// управления вводом/выводом; если параметр **po**

// имеет значение **NULL**, то файл пересылается с

// текущей позиции файла;

// - структура **LPTRANSMIT\_FILE\_BUFFERS**

// используется для пересылки информации

// перед и после пересылаемых данных файла;

// если значение параметра **pb** установлено в

// **NULL**, то пересылаются только данные файла;

// - параметр **flag** может принимать различныe:

// **TF\_DISCONNECT** - указывает на необходимость

// разрыва соединения после завершения функции;

// **TF\_REUSE\_SOCKET** - предполагает дальнейшее

// использование сокета.

Рисунок 3.13.1. Функция TransmitFile

//.................................................................

**int f;**

**long fh;**

**if ((f =\_open("TransmitFile.txt",O\_RDONLY))> 0)** // файл открылся ?

**{**

**fh = \_get\_osfhandle(f);** // получить os-handel файла

**if** **(TransmitFile(sS,(HANDLE)fh,0,0,NULL,NULL,TF\_DISCONNECT)**

**== FALSE)**

**throw SetErrorMsgText("transmit:",WSAGetLastError())**;

**}**

//..................................................................

Рисунок 3.13.2. Пример использования функции TransmitFile

дескриптор файла, который в примере получается с помощью другой библиотечной функции \_get\_osfhandle. Описание стандартных функций библиотеки можно найти в справочниках по языку С++, например в [13, 14].

**3.14. Применение интерфейса внутренней петли**

При отладке распределенных приложений удобно использовать интерфейс внутренней петли. Применение интерфейса внутренней петли позволяет моделировать обмен данными между процессами распределенного приложения на одном компьютере.

Как уже отмечалось выше, для интерфейса внутренней петли зарезервирован IP-адрес 127.0.0.1. В формате TCP/IP этот адрес можно задать с помощью определенной в Winsock2.h константы INADDR\_LOOPBACK. Все приведенные выше примеры, демонстрирующие обмен данными в сети TCP/IP, можно выполнить на одном компьютере с использованием этого адреса.

**3.15. Использование широковещательных IP-адресов**

До сих пор при разработке распределенного приложения предполагались известными сетевой адрес компьютера, на котором находится программа-сервер и номер порта, прослушиваемый этой программой. В реальности распределенное приложение не должно быть привязано к конкретным параметрам сокетов, т.к. это делает ограниченным его применение.

Для обеспечения независимости приложения от параметров сокета сервера (сетевой адрес и номера порта), как правило, номер порта делают одним из параметров инициализации сервера и хранят в специальных конфигурационных файлах, которые считывается сервером при загрузке (реже номер порта передается в виде параметра в командной строке). Так, например, большинство серверов баз данных в качестве одного из параметров инициализации используют номер порта, а при конфигурации (или инсталляции) клиентских приложений указывается сетевой адрес и порт сервера.

В некоторых случаях удобно возложить поиск сетевого адреса сервера на само клиентское приложение (при условии, что номер порта сервера известен). В этих случаях используются широковещательные сетевые адреса, позволяющие адресовать сообщение о поиске сервера всем компьютерам сети. Предполагается, что сервер (или несколько серверов) должен находиться в состоянии ожидания (прослушивания) на доступном в сети компьютере. При получении сообщения от клиента, сервер определяет параметры сокета клиента и передает клиенту необходимые данные для установки канала связи. В общем случае в сети может находиться несколько серверов, которые откликнутся на запрос клиента. В этом случае алгоритм работы клиента должен предполагать процедуру обработки откликов и выбора подходящего сервера. Сразу следует оговориться, что реально данный метод можно применять только внутри сегмента локальной сети, т.к. широковещательные пакеты, как правило, не пропускаются маршрутизаторами и шлюзами .

Использование широковещательных адресов возможно только в протоколе UDP. Поэтомупри создании дескрипторов сокетов (в программах клиентов и серверов) при вызове функции socket значение параметраtype должно быть SOCK\_DGRAM**,** а для обмена данными этом случае используются функции sendto и recvfrom**.**

// **-- установить опции сокета**

// **Назначение:** функция предназначена для установки режимов

// использования сокета

**int setsockopt (**

**SOCKET s,** // [in] дескриптор сокета

**int level,** // [in] уровень действия режима

**int optname,** // [in] режим сокета для установки

**const char\* optval,** // [in] значение режима сокета

**int fromlen** // [in] длина буфера **optval**

**);**

// **Код возврата:** в случае успешного завершения возвращается

// нуль, иначе функция возвращает значение

// **SOCKET\_ERROR**

// **Примечания:** - поддерживаются два значения параметра **level**:

// **SOL\_SOCKET** и **IPPROTO\_TCP;**

// - для уровня **SOL\_SOCKET** параметр **optval** может

// принимать более десяти различных значений;

// например, **SO\_BROADCAST** - для разрешения

// использования широковещательного адреса**;**

**//** - для уровня **IPPROTO\_TCP** поддерживается одно

// значение параметра **level**:**TCP\_NODELAY**,

// которое позволяет устанавливать или отменять

// использование алгоритма Нейгла (см. TCP/IP);

// - значение **fromlen** всегда **sizeof(int)**;

// - если необходимо установить указанный

// параметр**(optname)** в состояние Enabled**,**

// то в поле **optval** долнжо быть не нулевое

//значение (например, **0x00000001**), еслиже

// параметр устанавливается в состояние

// Disabled, то поле **optval** должно содержать

// **0x00000000**

Рисунок 3.15.1. Функция setsockopt

Стандартный широковещательный адрес в формате TCP/IP задается с помощью константы INADDR\_BROADCAST, которая определена в Winsock2.h. По умолчанию использование стандартного широковещательного адреса не допускается и для его применения необходимо установить специальный режим использования сокета SO\_BROADCASTс помощью функции setsockopt (рисунок 3.15.1). Проверить установленные для сокета режимы можно с помощью функции getsockopt (описание здесь не приводится).

//...........................................................

**SOCKET cC;**

**if ((cC = socket(AF\_INET, SOCK\_DGRAM, NULL))== INVALID\_SOCKET)**

**throw SetErrorMsgText("socket:",WSAGetLastError());**

**int optval = 1;**

**if (setsockopt(cC,SOL\_SOCKET,SO\_BROADCAST,**

**(char\*)&optval,sizeof(int)) == SOCKET\_ERROR)**

**throw SetErrorMsgText("opt:",WSAGetLastError());**

**SOCKADDR\_IN all;** // параметры сокета sS

**all.sin\_family = AF\_INET**; // используется IP-адресация

**all.sin\_port = htons(2000);** //порт 2000

**all.sin\_addr.s\_addr = INADDR\_BROADCAST;** // всем

**char buf[] = "answer anyone!";**

**if ((sendlen = sendto(cC, sendbuf, sizeof(buf), NULL,**

**(sockaddr\*)&all, sizeof(all)))== SOCKET\_ERROR)**

**throw SetErrorMsgText("sendto:",WSAGetLastError());**

//...........................................................

Рисунок 3.15.1. Пример применения setsockopе

На рисунке 3.15.1 приводится фрагмент программы, использующей стандартный широковещательный адрес. Функция setsockopt используется в этом примере для установки опции сокета SO\_BROADCAST, позволяющей использовать адрес INADDR\_BROADCAST.

**3.16. Применение символических имен компьютеров**

В предыдущем разделе разбирался механизм поиска серверного компьютера с помощью использования широковещательных адресов. При наличии специальной службы в сети способной разрешить адрес компьютера по его символическому имени (например, DNS или некоторые протоколы, работающие поверх TCP/IP) проблему можно решить с помощью функции gethostbyname (рисунок 3.16.1). При этом предполагается, что известно символическое имя компьютера, на котором находится программа сервера.

Такое решение достаточно часто применяется разработчиками распределенных систем. Связав набор программ-серверов с определенными стандартными именами компьютеров, распределенное приложение становится не зависимым от адресации в сети. Естественно при этом необходимо позаботиться, чтобы существовала служба, разрешающая адреса компьютеров по имени. Установка таких служб, как правило, возлагается на системного администратора сети.

Помимо функции gethostbyname в составе Winsock2имеется функция gethostbyaddr (рисунок 3.16.2)**,**  назначение которой противоположно**:** получение символического имени компьютера по сетевому адресу. Обе функции используют структуру hosten (рисунок 3.16.3), содержащуюся в Winsock2.h.

// **-- получить адрес хоста по его имени**

// **Назначение:** функция для получения информации о хосте по

// его символическому имени

**hostent\* gethostbyname**

**(**

**const char\* name,** // [in] символическое имя хоста

**);**

// **Код возврата:** в случае успешного завершения функция

// возвращает указатель на структуру **hosten**,

// иначе значение **NULL**

// **Примечание:** допускается в качестве символического имени,

// указать символическое обозначение адреса

// хоста в виде n.n.n.n

Рисунок 3.16.1. Функция gethostbyname

//-- **получить имя хоста по его адресу**

// **Назначение:** функция для получения информации о хосте по

// его символическому имени

**hostent\* gethostbyaddr**

**(**

**const char\* addr,** // [in] адрес в сетевом формате

**int la,** // [in] длина адреса в байтах

**int ta** // [in] тип адреса: для TCP/IP **AF\_INET**

**);**

// **Код возврата:** в случае успешного завершения функция

// возвращает указатель на структуру **hosten**,

// иначе возвращается значение **NULL**

Рисунок 3.16.2 Функция gethostbyaddr

**typedef struct hostent {** // структура hostent

**char FAR\* h\_name;** // имя хоста

**char FAR FAR\*\* h\_aliases;** // список алиасов

**short h\_addrtype**; // тип адресации

**short h\_length;** // длина адреса

**char FAR FAR\*\* h\_addr\_list;** // список адресов

**} hostent;**

Рисунок 3.16.3 Структура hostent

Следует отметить, что символическое имя ***localhost***  является зарезервированным именем и предназначено для обозначения собственного имени компьютера. Если с помощью функции gethostbyname получить адрес компьютера с именем localhost, то в будет собственный получен IP-адрес компьютера или адрес INADDR\_LOOPBACK.

Кроме того, для получения действительного собственного имени компьютера (NetBIOS-имени или DNS-имени) можно использовать функцию gethostname (рисунок 3.16.4).

// **-- получить имя хоста**

// **Назначение:** функция для получения собственного имени хоста

**int gethostname**

**(**

**char\* name ,** // [out] имя хоста

**int ln** // [in] длина буфера **name**

**);**

// **Код возврата:** в случае успешного завершения функция

// возвращает нуль, иначе возвращается значение

// **SOCKET\_ERROR**

Рисунок 3.16.4 Функция gethostname