1. Общая организация взаимодействия прикладных процессов через сокеты WinSock

Библиотека Sockets и поддерживаемые этими библиотеками специальные программные объекты типа Socket, называемые также сокетами, обеспечивают соединение двух процессов, работающих на одном и том же или на различных компьютерах (хостах) сети, через пару используемых этими процессами сокетов.

Сокет является специальным программным объектом, создаваемым при помощи вызова библиотечной функции socket(). Этот программный объект по своему назначению и использованию подобен дескриптору файла. Различие состоит в том, что вместо открытия файла должны быть выполнены операции установления соединения. Выполнение пересылки и приема данных также осуществляется с использованием специальных вызовов, хотя для этой цели могут применяться и традиционные функции чтения или записи данных в идентифицируемый сокетом «файл», т.е. – в компьютерную сеть.

Различают сокеты с установлением соединения (то есть сокеты для соединений типа «виртуальный канал») и без установления соединения, или дейтаграммные (datagram) сокеты. В первом случае информация, идентифицирующая взаимодействующие процессы сети указывается в параметрах системных вызовов, обеспечивающих установление соединения. Во втором случае эта информация указывается при каждом вызове передачи (приема или отправления) данных через дейтаграммный сокет [4,5].

Каждый из взаимодействующих сетевых процессов однозначно идентифицируется IP-адресом хоста, на котором он работает и номером порта, используемым процессом для обмена данными через сеть. Однако пользователям сетевых процессов привычнее и удобнее идентифицировать эти процессы сетевым доменным именем хоста и названием сетевого протокола (например, telnet, ftp, http и пр.), используемого этим процессом для взаимодействия со своими «партнерами». Для того чтобы обеспечить прикладному программисту

простоту преобразования задаваемых пользователями сетевых процессов данных, идентифицирующих такие процессы из формы, удобной пользователям, в форму, требуемую системными вызовами работы с сокетами, разработаны специальные функции, рассматриваемые в разделе 3 настоящего пособия.

Вкратце рассмотрим назначение системных вызовов библиотеки WinSock [3,6], подробнее рассматриваемых в последующих разделах настоящего пособия. Эти системные вызовы по их назначению удобно разбить на группы, в соответствии со следующими функциями: создание сокета, установление соединения между сокетами, передача данных через сокет и разрыв соединения между сокетами.

Создание сокета обеспечивается вызовом функции socket(). В результате выполнения этого системного вызова создается соответствующий программный объект, идентифицируемый значением специального типа – дескриптором сокета.

Установление соединений выполняется лишь для сокетов типа «виртуальный канал», используемых при реализации клиент серверных приложений. При этом установление соединения со стороны сервера и со стороны клиента выполняется по различному сценарию.

Установление соединения со стороны сервера выполняется путем последовательного выполнения трех функций: bind(), listen() и accept(). Системный вызов bind() (связать (с информацией об использующем сокет процессе)) используется для инициализации в сокете параметров самого серверного процесса, включающих IP-адрес локального хоста и номер порта, через который клиентские процессы могут обращаться к серверному процессу. Функция listen() (слушать) «прослушивает» сеть и принимает запросы, поступившие к серверному процессу через порт, заданный системным вызовом bind(). Принятые запросы помещаются в очередь, из которой они извлекаются для обработки системным вызовом accept() (принять (соединение)). В число параметров, получающих значения при выполнении этого системного вызова

относятся IP-адрес и номер порта клиентского процесса, с которым установлено соединение. Значения этих параметров предаются в пакете установления соединения, пересылаемом серверному процессу клиентским процессом.

Для дейтаграммных сокетов установление постоянного соединения между взаимодействующими через эти сокеты процессами не выполняется.

Следует отметить, что при ожидаемом разработчиком программы серверного процесса высоком темпе поступления запросов целесообразно предусмотреть их параллельную обработку. Организация многопотоковых серверных программ рассматривается в Части 2настоящего пособия.

Установление соединения с сервером со стороны клиента выполняется с помощью функции connect() (соединиться). Основными параметрами этого системного вызова являются IP-адрес хоста, на котором выполняется серверный процесс и номер порта этого серверного процесса, предназначенный для приема запросов клиентских процессов. Отметим, что в состав пересылаемого клиентом серверу пакета установления соединения включается также автоматически определяемый вызовом connect() IP-адрес локального хоста и динамически выделяемый (из числа не занятых) номер порта клиентского процесса. Динамическое выделение номера порта клиентскому процессу вполне допустимо, поскольку этот номер передается серверному процессу в пакете установления соединения. Отметим что, динамическое выделение номера порта клиентскому процессу обеспечивает возможность вызова на хосте нескольких экземпляров однотипных клиентских процессов, например, нескольких терминальных окон различных удаленных компьютеров. В то же время номер порта серверного процесса должен быть задан статически, для того чтобы клиентские процессы могли знать, какой порт сервера надо указывать при установлении соединения.

Пересылка данных между процессами, связанные через сокеты с установлением соединения, обычно выполняется при помощи системных вызовов send() (переслать) и recv() (от receive - принять). Однако для передачи данных

могут использоваться и традиционные функции файлового обмена write() и read() соответственно; при этом вместо дескриптора файла в этих системных вызовах задается дескриптор требуемого сокета.

Подчеркнем, что пересылка данных, выполняемая с использованием функций send() и write(), осуществляется с буферизацией, так что факт выполнения каждого системного вызова не обязательно влечет немедленную передачу серверу информации, занесенной в промежуточный буфер клиентского компьютера. Поэтому, если логика работы клиентского процесса такова, что после выполнения клиентом одной или нескольких операций передачи серверу клиент должен выполнить прием ответа от сервера, программа клиентского процесса должна выполнить специальные действия. Для того чтобы сервер фактически принял все переданные ему данные и выполнил подготовку и передачу требуемого клиенту ответа, клиент должен явно «вытолкнуть» в сеть все содержимое буфера. Это может быть сделано при помощи системного вызова flush() (очистить (буфер)). Аналогично, сервер при завершении обработки запроса клиента и выполнения последней операции send() или write() также должен выполнить очистку буфера с использованием того же системного вызова.

Пересылка данных через дейтаграммные сокеты выполняется при помощи функций sendto() (переслать для) и recvfrom() (принять от). Поскольку пересылка данных через дейтаграммные сокеты выполняется без предварительного установления соединения, в число параметров указанных системных вызовов кроме адреса и длины буфера, содержащего передаваемые через сеть данные, входят также параметры, задающие IP-адрес хоста и номер порта процесса, с которым взаимодействует данный экземпляр системного вызова. Кроме того, поскольку серверный процесс должен взаимодействовать с сетью через определенный порт, то в его программе перед выполнением системных вызовов по пересылке данных через сеть необходимо выполнить «связывание» с сокетом информации о номере порта с использованием системного вызова bind().

Для разрыва соединения, установленного между клиентским и серверным процессом, используется функция shutdown(). Он позволят разорвать соединение как в одном из двух возможных направлений пересылки данных, так и для полного разрыва соединения в обоих направлениях. Полный разрыв соединения может быть также выполнен при помощи системного вызова close(), одновременно уничтожающем указанный его параметром сокет. Системный вызов close() используется также для уничтожения дейтаграммных сокетов после того, как все операции передачи данных через эти сокеты завершены.

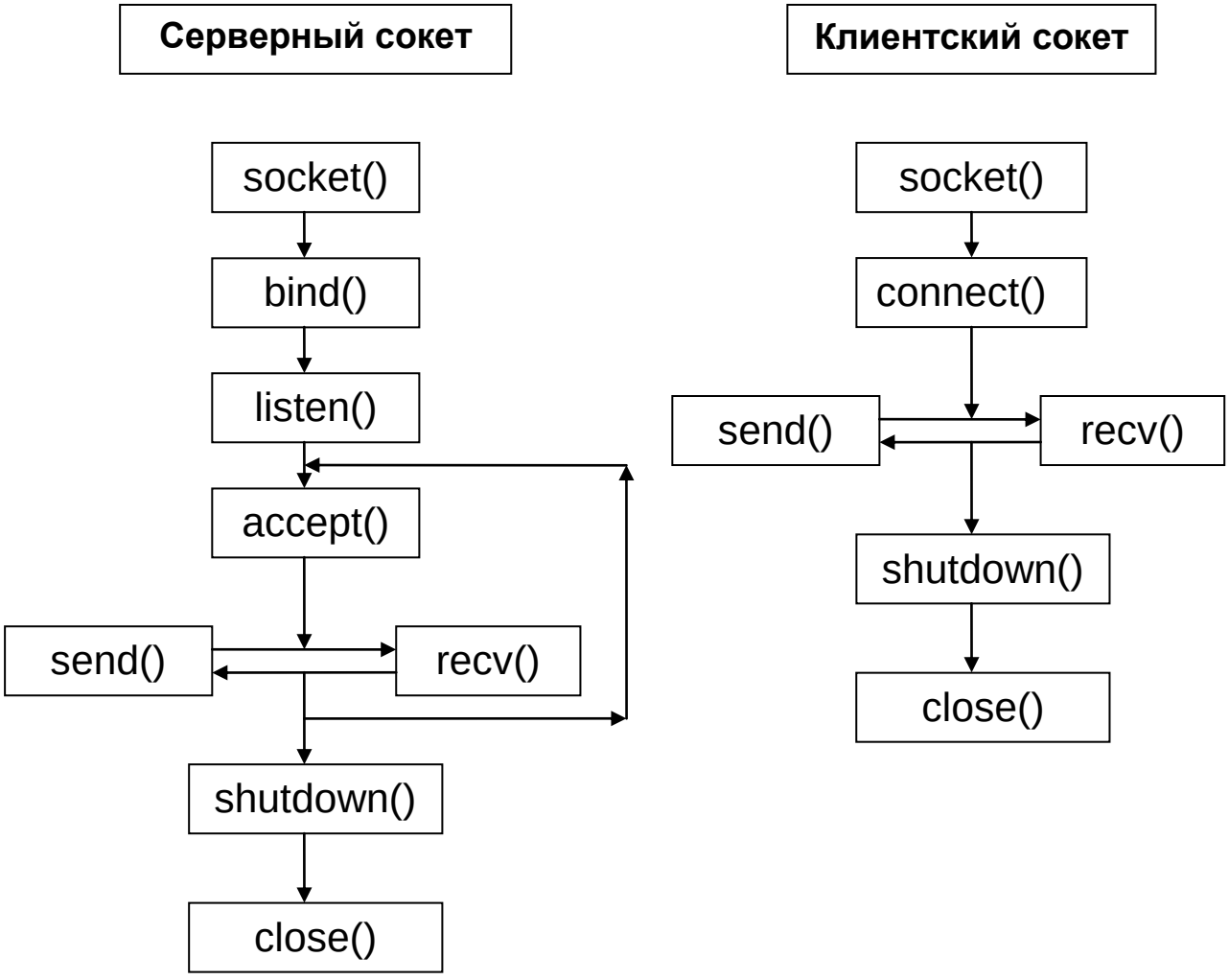

Рисунок 1. Последовательность выполнения функций при взаимодействии сервера и клиента через виртуальный канал

В завершение краткого рассмотрения системных вызовов для работы с сокетами приведем схему последовательности выполнения этих системных вызовов для взаимодействия клиентского и серверного процессов через соединение типа виртуальный канал, представленную на Рис. 1. и схему последовательности выполнения системных вызовов для дейтаграммных процессов, представленную на Рис. 2.

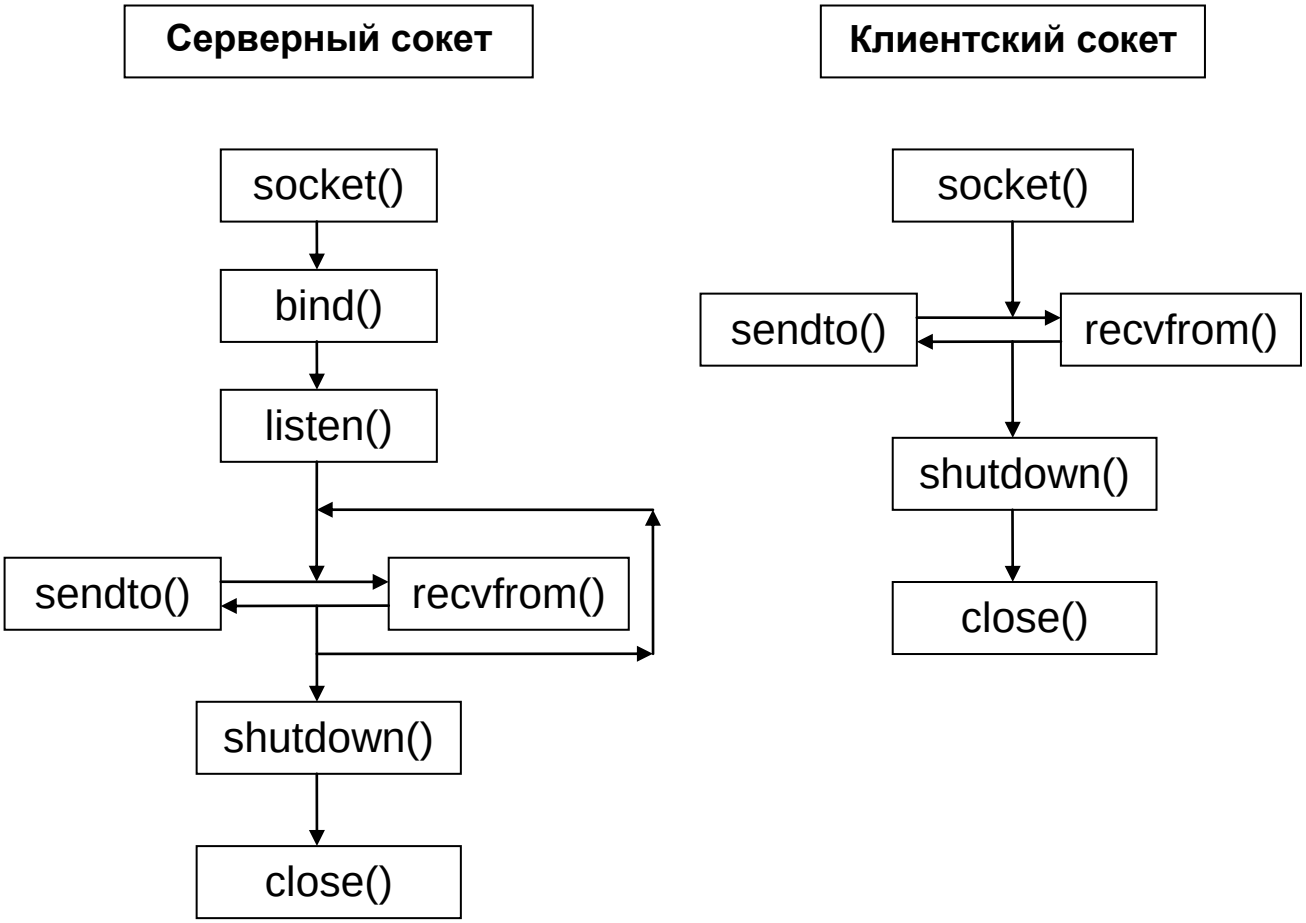

Рисунок 2. Последовательность выполнения функций в сетевой программе при взаимодействии через дейтаграммные сокеты

2. Оформление Windows Socket приложений

Рассмотрим особенности общего оформления программ сетевых приложений, создаваемых с использованием библиотеки Windows Sockets (WSA) в среде MS VisualStudio. Функции WSA описаны в заголовочном файле winsock2.h. Если создается оконное приложение, этот файл автоматически подключается из заголовочного файла windows.h, однако для консольных приложений его необходимо включать явным образом. Кроме того, следует подключить статическую библиотеку ws2_32.lib; это можно сделать одним из двух способов – либо при помощи директивы #pragma, либо добавить эту библиотеку в разделе дополнительных параметрах (Additional options) сборщика (Linker) программ (через настройки конфигурации проекта), как это показано на Рис. 3.

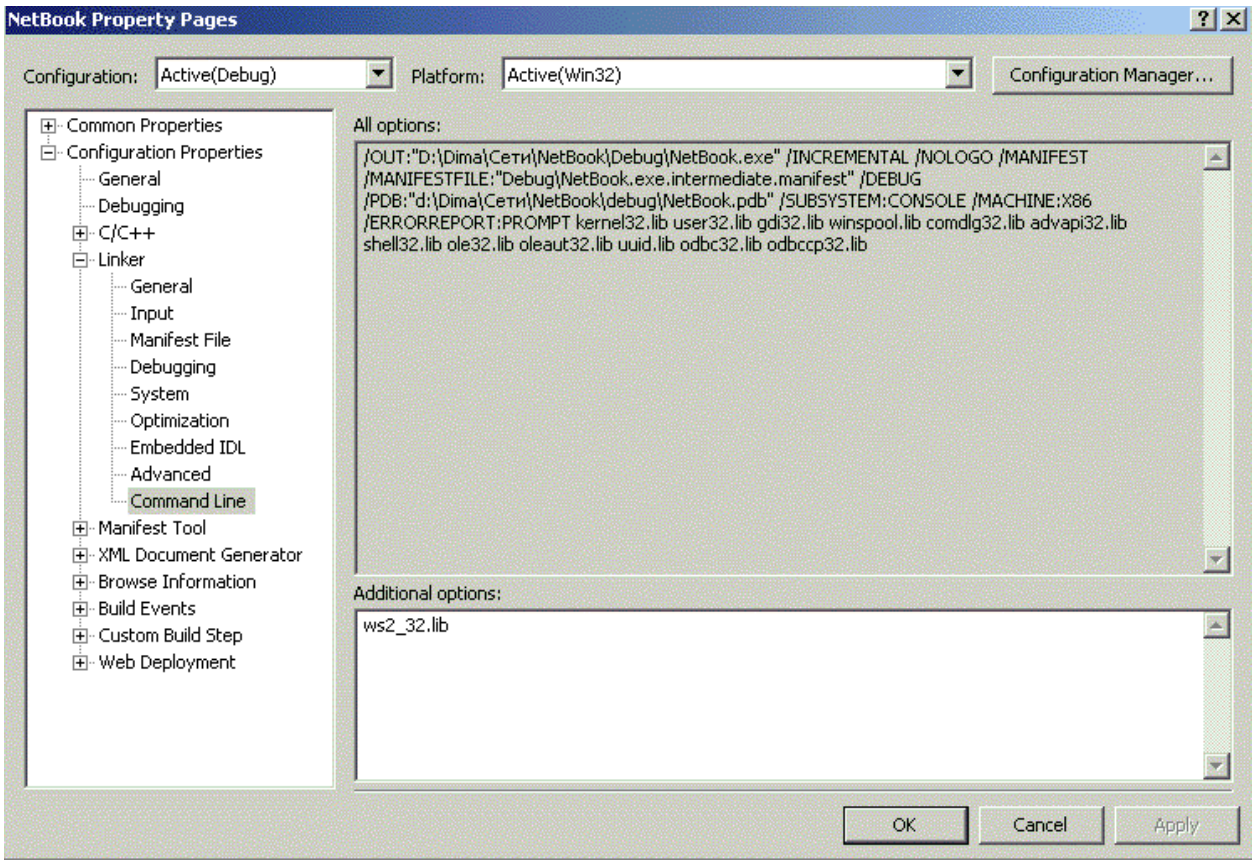

Рисунок 3. Подключение библиотеки ws2_32.lib

В программе приложения при использовании WinSock необходимо выполнить инициализацию библиотеки ws2_32.dll, входящей в состав операционной системы. Для этого используется пара функций WSAStartup() и WSACleanup(), прототипы вызовов которых приводятся ниже:

```
int WSAStartup( WORD wVersionRequested,
                LPWSADATA lpWSAData );
int WSACleanup(void);
```

Первый параметр функции WSAStartup() определяет запрашиваемую версию библиотеки ws2_32.dll (обычно используется версия 2.0), а второй параметр – специальное значение типа WSADATA, используемое для служебных целей. Функция WSAStartup(), таким образом, должна вызываться в приложении до первого вызова функций библиотеки Windows Sockets, а WSACleanup() – по окончанию работы. Ниже приведен пример использования этих функций:

```
#include "stdafx.h"
#include <winsock2.h>

int _tmain(int argc, _TCHAR* argv[])
{
//      ……
        WSADATA  ws;
        WSAStartup( MAKEWORD( 2,0 ), &ws );
//      Вызов функций библиотеки WSA
//      ……
        WSACleanup();
//      ……
}
```

Здесь макрос MAKEWORD(2,0) используется для построения номера используемой версии библиотеки.

Функция WSAStartup() возвращает значение ноль в случае успешного выполнения.

Особо следует подчеркнуть необходимость отслеживания возникновения ошибок и обработки ошибочных ситуаций. Хорошей практикой при написании приложений является проверка наличия или отсутствия ошибки после каждого вызова библиотечной функции. Обычно наличие ошибки можно определить по возвращаемому значению функции. Например, функция WSAStartup() возвращает значение NO_ERROR при успешном завершении. Для того, чтобы узнать, какое значение возвращает функция при ошибке, необходимо обращаться к описанию данной функции в справочной службе MSDN. Кроме того, можно перейти к описанию функции или макроса в соответствующем заголовочном файле; для этого необходимо поместить курсор на имя функции или макроса в окне редактора с исходным кодом и в выпадающем меню, вызывамом по нажатию правой кнопки мыши, выбрать пункт «Go To Definition».

В случае выявления факта возникновения ошибки следующим шагом является получения числового кода ошибки и ее текстового описания. Для этого используются функции WSAGetLastError() и GetLastError(), которые возвращают код ошибки (для них в файле winerror.h определены макросы с мнемоническими именами), и функция FormatMessage(), предназначенная для получения текстового описания, соответствующего коду. Функции WSAGetLastError() и GetLastError(), являющиеся для WSA-приложений взаимозаменяемыми, следует вызывать немедленно после выхода из функции, завершившейся ошибочно, поскольку вызов следующей функции может сбрасывать код последней ошибки.

Переменную errno, используемую для получения кода ошибки в UNIX-приложениях [2], в среде ОС Windows использовать не следует.

Для получения текстового описания ошибки рекомендуется вызывать функцию FormatMessage() следующим образом:

```
wchar_t      ErrorMessage[1000];
FormatMessage(
        FORMAT_MESSAGE_FROM_SYSTEM, NULL,
        WSAGetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        ErrorMessage, sizeof( ErrorMessage ),
        NULL);
wprintf( L"\nОШИБКА: %s\n", ErrorMessage );
```

Здесь четвертый параметр – макрос MAKELANGID определяет язык, который будет использоваться для формирования текста с описанием ошибки. В данном примере используется языковая настройка по умолчанию. Можно попробовать явно указать требуемый язык, например, при помощи следующего фрагмента кода:

```
MAKELANGID( LANG_RUSSIAN, SUBLANG_DEFAULT)
```

Однако в большинстве случаев значение по умолчанию является единственно возможным значением языковой настройки. В случае невозможности сформировать сообщение с учетом запрошенного языка функция FormatMessage() возвращает ноль.

3. Методы работы с IP-адресами, доменными именами и портами

Рассмотрим методы работы с IP-адресами и доменными именами, структуры данных, используемые для их представления в API WinSock, и основные функции для манипулирования с ними.

Любой хост в TCP/IP-сетях имеет IP-адрес, который однозначно идентифицирует данный хост; у хоста обычно бывает один IP-адрес, но может быть и несколько (обычно – по количеству сетевых интерфейсов хоста).

IP-адрес представляет собой 32-разрядное (то есть 4-х байтное) значение. Во внешнем представлении значение IP-адреса обычно записывается побайтно в виде XXX.XXX.XXX.XXX, где символы XXX обозначают десятичное целое значение от 0 до 255 включительно, однако адреса со значениями 0 и 255 имеют специальный смысл. Такой вид IP-адресов используется в версии IPv4 протокола IP. Версия IPv6 протокола IP, хотя уже и начала свое победное шествие, но еще не распространена повсеместно и, в частности, не используется на компьютерах учебных классов ЮФУ. Поэтому, хотя работа с адресами IPv6 в WSA поддерживается, в данном пособии средства работы с такими адресами не рассматривается.

В программных интерфейсах IP-адреса обычно указываются в бинарном представлении. При этом полное значение IP-адреса может быть представлено длинным целым (для 32-разрядных приложений). В WSA для хранения IP-адреса предусмотрены структура in_addr и тип IN_ADDR; их определения, цитируемые из заголовочного файла winsock2.h, приведены ниже:

```
struct in_addr {
  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;
} in_addr;
typedef struct in_addr IN_ADDR;
```

Определение структуры in_addr, которое может показаться несколько громоздким, позволяет обращаться к каждому байту по отдельности (поля s_b1,s_b2,s_b3,s_b4 типа u_char – беззнаковое однобайтовое значение), или к адресу в целом (поле addr типа u_long – беззнаковое длинное).

Для преобразования IP-адресов из символьного представления в бинарное и обратно предусмотрены функции inet_addr() и inet_ntoa(); ниже приведены их спецификации:

```
unsigned long inet_addr (   const char* cp );
char* FAR inet_ntoa(   struct   in_addr in );
```

Функция inet_addr() получает указатель на начало строки, содержащей текстовое представление адреса, и возвращает адрес в виде длинного целого. Если строка содержит значение, которое не соответствует структуре IP-адреса, возвращается значение INADDR_NONE; отсутствие ошибки следует проверять при возврате из функции. Следует иметь в виду, что функция inet_addr() ни в коем случае не проверяет фактическое существование в сети хоста с этим адресом.

Функция inet_ntoa() выполняет обратное преобразование из бинарного, передаваемого в виде структуры, в текстовый вид, который возвращается в виде указателя на начало буфера.

Ниже приведен программный код, иллюстрирующий вызов этих двух функций.

```
setlocale( LC_ALL, "Russian_Russia.866" );
long       ip =  inet_addr( "10.0.0.31" );
if( ip == INADDR_NONE )
{
    wprintf( L"Неправильный формат IP-адреса\n"  );
     //      Обработка ошибки
}
struct     in_addr       ips;
ips.s_addr = ip;
wprintf( L"IP-адрес:\t%d.%d.%d.%d\n",
                    ips.S_un.S_un_b.s_b1, ips.S_un.S_un_b.s_b2,
                    ips.S_un.S_un_b.s_b3, ips.S_un.S_un_b.s_b4 );
```

Кроме IP-адреса, хосты обычно имеют мнемонические доменные имена, например, «www.microsoft.com». Такие имена, естественно, более удобны для пользователей прикладных программ, указывающих эти имена (через пользовательский интерфейс с прикладной программой) для доступа к поименованным серверам, однако при разработке программ сетевых приложений для идентификации хостов необходимо использовать именно IP-адреса.

Поддержка соответствия между IP-адресами и доменными именами выполняется доменной службой имен (DNS) [4]. Для каждого хоста существует один или несколько IP-адресов, официальное (или – каноническое) доменное имя, и список синонимов. Официальное имя – это либо полностью квалифицированное доменное имя, если используется система DNS, либо локальное имя, присвоенное компьютеру при инсталляции операционной системы. Отметим, что имена хостов типа «www», «ftp» и другие имена, совпадающие с названиями прикладных сетевых служб, обычно являются как раз синонимами, тогда как официальные

имена в большинстве случаев являются малоинформативными с пользовательской точки зрения.

Для представления IP-адресов, имени и синонимов хоста используется структура hostent и тип HOSTENT:

```
typedef struct hostent {
        char FAR* h_name;
        char FAR  FAR** h_aliases;
        short h_addrtype;
        short h_length;
        char FAR  FAR** h_addr_list;
} hostent
```

Здесь поле h_name содержит указатель на строку с каноническим именем хоста, h_aliases – список синонимов, h_addrtype – тип используемого адреса, h_length - длина значения адреса в байтах, и h_addr_list – список IP-адресов в бинарном представлении.

Напомним, что под списком в языке C понимается массив указателей, каждый элемент которого является адресом, а в качестве последнего элемента находится NULL, что является признаком конца массива. Опуская модификатор FAR, описание поля h_aliases можно было бы записать так:

```
char* h_aliases[ ];
```

Для перебора всех элементов таких списков можно использовать следующий цикл:

```
hostent* host;
…….
for( char** pc = host->h_aliases; *pc != NULL; pc++)
        printf( *pc );
```

Напомним, что pc++ - это операция инкрементации указателя, вычисление которой состоит в перемещении указателя на следующий элемент в списке.

Может вызвать некоторое удивление определение списка IP-адресов как char FAR FAR** h_addr_list; более ожидаемым, возможно, было бы видеть определение примерно такого вида: in_addr ** h_addr_list. Объясняется такое определение тем, что во время разработки библиотеки Berkeley Sockets (вторая половина 70-х годов) для языка C было характерно определять указатели на произвольную область памяти (т.е. нетипизированные указатели, согласно современной терминологии) именно как char*, что закрепилось в библиотеках. При компиляции для C++, однако, необходимо выполнять явное приведение типа фактического указателя к указателю на тип, использованный в прототипе функции (обычно это char*).

Имя локального хоста можно получить при помощи функции gethostname():

```
int gethostname( char* name,   int namelen  );
```

Эта функция записывает в передаваемый буфер указанной длины имя хоста и возвращает значение 0 в случае успеха.

Для получения полной информации о хосте следует использовать следующие две функции:

```
struct hostent* FAR gethostbyname(  const char* name );
struct hostent * FAR gethostbyaddr(const char* addr, int len, int type);
```

Эти функции выполняют обращение к DNS-серверу, если таковой доступен (в этом случае возвращается официальное имя, полный набор зарегистрированных IP-адресов и синонимов) или к NetBIOS для получения локального имени. Функция gethostbyname() получает имя хоста в виде строки, gethostbyaddr() – IP-адрес в бинарном виде, его длину и тип сети (следует

использовать значение-макрос AF_INET). Обе функции возвращают указатель на структуру hostent; следует иметь в виду, что повторный вызов функции стирает предыдущее значение структуры, поэтому значения рекомендуется копировать в локальные переменные. В случае ошибки эти функции возвращают нулевой указатель, это обычно связано с тем, что данный хост не существует. Следует иметь в виду, что наличие у DNS-сервера информации о хосте не свидетельствует о том, что этот хост в настоящее время действительно функционирует в сети.

Вызов функции gethostbyname() возвращает указатель на структуру с данными о локальном хосте.

Рассмотрим пример использования приведенных функций.

```
char  HostName[100];
if( gethostname( HostName, sizeof( HostName ) ) != SOCKET_ERROR )
{
//      Обработка ошибки
}
wprintf( L"Название данного хоста: %S\n",  HostName );
hostent*  host = gethostbyname( NULL );
if( host != NULL )
{
        wprintf( L"Официальное имя хоста: %S\n", host->h_name );
        for( char** pc=host->h_aliases; *pc ; pc++ )
                wprintf( L"Синоним: %S\n",  *pc );
}
long addr = inet_addr( "10.0.0.26" );
host = gethostbyaddr( (char*)&addr, 4, AF_INET  );
if( host != NULL )
{
        wprintf( L"IP-адрес: %S\n", inet_ntoa(  *(in_addr*)(host->h_addr_list[0])) );
}
```

Следует подробнее остановиться следующем на выражении и его значении:

```
*(in_addr*)( host->h_addr_list[0] )
```

Разбор этого выражения следует начинать справа, с подвыражения host->h_addr_list[0]. Напомним, что это указатель типа char*, указывающий на 32-разрядное значение адреса. Однако этот указатель непосредственно разыменовывать не следует по двум причинам. Во-первых, выражение *host->h_addr_list[0], с формальной точки зрения, имеет тип char вместо ожидаемого in_addr, согласно прототипу функции. Эта ошибка будет обнаружена компилятором C++, который выполняет проверку соответствия типов указателей. При использовании компилятора C эта ошибка компиляции не возникает, однако произошло бы неявное преобразование значения выражения *host->h_addr_list[0], имеющего тип char, к типу long, в результате чего реально был скопирован только первый байт значения IP-адреса. Поэтому необходимо сначала выполнить явное приведение указателя char* к in_addr*, а затем выполнить его разыменование.

Для студентов типичны аналогичные ошибки следующего вида:

```
cout << *host->h_addr_list[0];
long ip = *host->h_addr_list[0];
```

Напомним, последня строка эквивалентна

```
long ip = (long)*host->h_addr_list[0];
```

Корректными будут следующие выражения:

```
cout << *(long*)host->h_addr_list[0];
long ip = *(long*)host->h_addr_list[0];
```

В дополнение к IP-адресу для идентификации сетевых прикладных процессов в TCP/IP сетях используется номер порта – 16-разрядное

положительное целое число, которое необходимо для идентификации конкретной сетевой службы; в современной терминологии пара <IP-адрес, номер порта> называется конечной точкой сетевого соединения. Многие номера портов, в особенности в начале диапазона, закреплены за стандартными (или – общеизвестными) сетевыми службами (well-known services); эти службы имеют имена, обычно совпадающие с названиями соответствующих сетевых протоколов прикладного уровня.

Для представления сведений о службах используется структура servent:

```
typedef struct servent {
        char FAR* s_name;
        char FAR  FAR** s_aliases;
        short s_port;
        char FAR* s_proto;
} servent;
```

Поля этой структуры содержат, соответственно, название службы, список синонимов, номер порта и название протокола, используемого данной службой.

Получить описание стандартной службы по номеру используемого порта или по названию этой службы можно при помощи функций getservbyport() и getservbyname() соответственно:

```
servent* FAR getservbyport( int port, const char* proto );
servent* FAR getservbyname( const char* name,  const char* proto );
```

Второй параметр функций – название протокола; вместо названия можно передать значение 0, если не имеет значения, какой конкретный протокол используется. Функции в случае успеха возвращают указатель на структуру servent, и NULL в случае ошибки.

При передаче значения номера порта следует использовать функцию htons(), преобразующую этот номер специальным образом.

Рассмотрим два примера использования этих функций.

```
servent *serv = getservbyname( "ftp", 0 );
if( serv )
        wprintf( L"Служба:\t%S\t%d\t%S\n",
                serv->s_name, htons(serv->s_port),
                serv->s_proto );

for( int i = 0; i < 200; i++ )
{
        servent *serv = getservbyport( htons( i ), "UDP" );
        if( serv )
                wprintf( L"Служба:\t%S\t%d\t%S\n",
                        serv->s_name, htons( serv->s_port ),
                        serv->s_proto );
}
```

Первый пример иллюстрирует вызов getservbyname() и вывод сведений о службе «ftp»; обратите внимание на доступ к полю s_port. Во втором примере выводятся сведения обо всех стандартных службах в диапазоне до 200, использующих протокол UDP. Отметим, что одна и так же служба может использовать различные протоколы, и списки общеизвестных служб на различных хостах могут несколько отличаться.

4. Создание и использование сокетов

Как отмечалось в разделе 1 настоящего пособия, сокетом называется специальный программный объект, который предназначен для организации двустороннего обмена данными с другими хостами в сети. Сокет как структуру данных языков C и C++ можно рассматривать как файл специального вида, с которым можно работать при помощи функций read(), write(),close() и др., однако рекомендуется использовать специфичные для сокетов функции.

Для создания сокета используется функция socket():

```
SOCKET socket( int af,  int type,  int protocol );
```

Обратим внимание, что тип SOCKET в WSA не является синонимом типа int, как в Berkerley Sockets.

Функция socket() при успешном завершении возвращает созданный сокет, и в случае ошибки – значение INVALID_SOCKET.

Три параметра этой функции определяют, соответственно, тип используемых в сети адресов, тип сокета и тип используемого протокола. В качестве значения первого параметра можно использовать, в том числе, именованные константы (значения-макросы):

AF_UNIX – это значение указывает, что сокеты используются для межпроцессного взаимодействия на локальном хосте;

AF_INET – это значение используется при создании сокетов сетевых приложений;

AF_INET6 – значение указывается для сетевых приложений, использующих IPv6-адреса.

Следует отметить, что WSA позволяет использовать широкий спектр протоколов и соответственно различных видов адресации, и не только семейства

TCP/IP-сетей. Полный список возможных значений для поддерживаемых протоколов можно найти в файле winsock2.h, см. также [3,5].

К поддерживаемым типам сокета относятся:

SOCK_STREAM - сокеты на основе виртуальных каналов,
SOCK_DGRAM - сокеты на основе дейтаграммных соединений,
SOCK_RAW - т.н. Row –сокеты (“сырые” сокеты).

Наконец, третий параметр определяет тип протокола; протоколу TCP соответствует макрос IPPROTO_TCP протоколу UDP - IPPROTO_UDP; можно использовать значение 0. В этом случае для сокетов, используемых для соединений типа «виртуальный канал», автоматически выбирается протокол TCP; для дейтаграммных сокетов – протокол UDP.

Список именованных констант (макросов), соответствующих типам протоколов, можно найти в заголовочном файле winsock2.h. Кроме того, можно использовать пару функций getprotobyname() и getprotobynumber():

```
struct protoent FAR * getprotobynumber( int number );
struct protoent FAR * getprotobyname( char* name );
```

Эти функции позволяют по имени или номеру протокола получать сведения о службе, возвращаемые в виде указателя на структуру protoent.

```
LPPROTOENT     proto = getprotobyname("TCP" );
SOCKET     sock = socket( AF_INET, SOCK_STREAM, proto->p_proto );
```

Следует отметить, что не все номера доступных протоколов перечислены среди макроопределений файла winsock2.h, и наоборот, не все эти перечисленные номера протоколов можно использовать при создании сокетов. Возможность создания сокета для некоторого протокола определяется операционной системой

и ее сетевыми настройками на конкретном хосте. Если необходимо создать сокет для работы с каким-либо специфическим сетевым протоколом, рекомендуется использовать функцию getprotobyname().

Неиспользуемые сокеты следует закрывать, для это предусмотрена функция closesocket():

```
int closesocket(   SOCKET s );
```

Перейдем к рассмотрению архитектуры сетевых распределенных приложений на основе WSA [3,5]. Программа, которая инициируется взаимодействие, называется клиентом, а программа, ожидающая обращение– сервером.

Клиент и сервер имеют несколько различную структуру; зависящую, в том числе, от того, используется ли TCP- или UDP-соединение (в других терминах, соединение типа «виртуальный канал» или дейтаграммное соединение). Сначала рассмотрим работу с TCP-соединением.

Процедуры инициализации WSA-приложения и создания сокета и на клиенте, и на сервере одинаковы.

Клиентская часть приложения после создания сокета должна вызвать функцию connect(), которая устанавливает соединение клиента с сервером и начинает TCP-сеанс.

```
int connect( SOCKET s,  const struct sockaddr* name, int namelen  );
```

Параметрами этой функции являются: правильным образом созданный сокет, указатель на значение структуры sockaddr, содержащий адресную информацию и сервере, и длина этого значения.

При использовании приложений в сетях TCP/IP на основе Ipv4-адресов вместо структуры sockaddr (которая является «прототипной») следует использовать структуру sockaddr_in, определение которой приводится ниже:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

Поле sin_family определяет тип сокета (например, значение AF_INET), sin_port – номер порта, который использует серверная программа, sin_addr – IP-адрес хоста с серверной частью приложения, и, наконец, массив sin_zero содержательной информации не содержит (рекомендуется заполнить его нолями).

Функция connect(), вызываемая в клиентской программе, возвращает значение 0 при успешном ее выполнении. Это означает, что по указанному адресу действительно находится некоторое активное серверное приложение, «слушающее» указанный порт, и что с этим приложением можно обмениваться данными. При неуспехе возвращается значение SOCKET_ERROR.

Ниже приведен пример инициализации структуры sockaddr_in и вызова функции connect():

```
const char   IPAddr[] = "127.0.0.1";
const long   NPort = 12001;
struct sockaddr_in addr;
memset( &addr, 0, sizeof( addr ) );
addr.sin_family = AF_INET;
addr.sin_port = htons( NPort );
addr.sin_addr.s_addr = inet_addr( IPAddr );
if( connect( sock, (sockaddr*) &addr, sizeof( addr ) ) != 0 )
{
        DWORD error = GetLastError();
```

```
        wchar_t Mess[1000];
        swprintf( Mess, L"Ошибка Connect() %S:%d",
                inet_ntoa( addr.sin_addr ), htons(NPort) );
                WriteErrorMessage( Mess, error );
}
```

Для инициализации серверной части необходимо после создания сокета последовательно вызвать функции bind() и listen():

```
int bind(  SOCKET s,  const struct sockaddr* name,   int namelen );
int listen(   SOCKET s,   int backlog );
```

Функция bind() используется для привязки вызвавшего ее приложения к локальному адресу и номеру порта серверного процесса, информация о которых предварительно помещается в передаваемую структуру sockaddr. Как и в случае с функцией connect(), следует использовать структуру sockaddr_in, при инициализации которой следует присвоить значения для типа адресации, номера порта, который будет прикреплен данный сокет, и IP-адреса. В качестве IP-адреса следует указать один из IP-адресов хоста, на котором размещен серверная часть приложения, или макроопределение INADDR_ANY.

Функция bind() должна быть вызвана на сервере для TCP-соединений; на клиенте эту функцию можно не вызывать (хотя это возможно), поскольку connect() автоматически выполняет неявную привязку клиентской части к некоторому свободному порту. Вызов bind() на клиенте необходим, если планируется использовать какой-либо конкретный номер порта. Если запрошенный порт занят, bind() возвращает значение SOCKET_ERROR, свидетельствующее об ошибке.

Если необходимо узнать, к какому номеру порта был привязано клиентское приложение после неявной привязки, можно воспользоваться функцией getsockname():

```
int getsockname( SOCKET s, struct sockaddr* name,   int* namelen );
struct sockaddr_inaddr_self;
memset( &addr_self, 0, sizeof( addr_self ) );
int len = sizeof( addr_self );
if( getsockname( sock, (sockaddr*)&addr_self, &len ) == 0 )
        wprintf( L"Параметры соедининия с %S:%d\n",
                inet_ntoa( addr_self.sin_addr ), htons( addr_self.sin_port) );
```

Следует обратить внимание на то, что третьим параметром является указатель на переменную, содержащую адрес структуры, а не значение указателя, как в функциях connect() и bind().

Функция listen() переводит привязанный сокет серверной части в состояние, когда приложение начинает принимать входящие соединения, при этом устанавливается длина очереди для принятых запросов на установление соединения (второй параметр). В качестве второго параметр рекомендуется использовать макрос SOMAXCONN. После того как очередь будет заполнена входящими соединениями, последующие соединения будут отвергаться сервером, вызывающее возникновение ошибки при возврате из функции connect().

Для обработки очередного входящего сообщения, извлекаемого из очереди, используется функция accept():

```
SOCKET accept( SOCKET s, struct sockaddr* addr,  int* addrlen );
```

Эта функция является блокирующей, что означает, что вызвавшее ее приложение будет заблокировано, если в очереди нет входящих соединений. Возврат из функции accept() произойдет, когда в очереди к приложению будет помещено входящее соединение. Это происходит после того, как какой-либо клиент выполнит функцию connect(), адресованный данному серверному приложению. При возврате из функции в структуру addr помещается информация

о клиенте, а в параметр addrlen – длина структуры; вместо этих параметров функции можно передать NULL-указатели, в этом случае, естественно, сведения о клиенте не возвращаются.

Возвращаемое значение функции accept() – дескриптор сокета, которую следует использоваться при взаимодействии с клиентом. В случае ошибки возвращается значение INVALID_SOCKET.

Следует иметь в виду, что при однопотоковой архитектуре сервер сможет взаимодействовать только с одним клиентом одновременно, и только по окончанию работы и завершению соединения возможен переход к обработке очередного соединения из очереди. Для последовательной обработки входящих соединений рекомендуется использовать следующую архитектуру серверного приложения:

```
SOCKET     sock = socket( AF_INET, SOCK_STREAM,  IPPROTO_TCP );
bind( sock, (sockaddr*)&baddr, sizeof( baddr ));
listen( sock, SOMAXCONN );
for(  ; ; )
{
        SOCKET     newsock = accept( sock,  (sockaddr*)&aaddr, &len );
                if( newsock != INVALID_SOCKET )
                {
                //       Взаимоедействие с клиентом

                        shutdown( newsock, 0 );
                        closesocket( newsock );
                }
}
shutdown( sock, 0 );
closesocket( sock );
```

После того как на клиенте и на сервере произошел возврат из пары connect() – accept(), установлено соединение, по которому можно передавать данные. Как говорилось выше, TCP-соединение допускает двустороннее чтение-запись через сокет, который можно рассматривать как файл. Для чтения и записи можно использовать функции read() и write(), а так же специализированные функции WIN32API ReadFile() и WriteFile(), однако рекомендуется использовать предусмотренные функции send() и recv():

```
int recv(  SOCKET s,  char* buf,  int len,   int flags );
int send(   SOCKET s,   const char* buf,   int len,   int flags );
```

Функция send(), по аналогии с функциями записи в файл write(), посылает данные из буфера (в количестве значения параметра len) в сокет (то есть – в сеть), а recv() – получает из сети указанное количество байт и помещает их в буфер. Последние параметры этих функций используются как настройки пересылки; можно использовать значение 0. Функция recv() является так же блокирующей.

Функции возвращают количество реально переданных или полученных байтов; эти величины удобно использовать для проверки получателем того, что все данные через сокет получены.

Ниже приведен фрагмент кода, иллюстрирующий использование функций send() и recv() на примере пересылки строки. Следует обратить внимание на то, что с принципиальной точки зрения не важно, какая именно сторона приложения – клиент или сервер – обращается к другой стороне с операцией чтения или записи в сокет, однако необходимо разрабатывать согласованные программы сторон, чтобы операции записи одной стороны соответствовала операция чтения другой, и наоборот.

```
//  Сторона, посылающая данные

    const  int              buflen = 100;
    char   Str[buflen];
    memset( Str, 0, buflen );
    gets( Str );
    int rc = send( sock, Str, strlen( Str ), 0 );


//  Сторона, принимающая данные

    memset( Str, 0, buflen );
    int  rc = recv( newsock, Str, buflen, 0 );
    if( SOCKET_ERROR == rc )
    {
    //      Обработка ошибок
    }
```

Отметим, что при отсутствии аварийной ситуации функция send() обычно не генерирует ошибку (однако и доставка данных получающей стороной не гарантируется), а вот возникновение ошибки при вызове recv() в нормальных условиях обычно означает прекращение сеанса. Рекомендуется для нормального прекращения сеанса использовать функцию shutdown(sock,0) (см. следующий раздел) на обеих сторонах приложения.

5. Использование дейтаграммных соединений

Теперь рассмотрим использование дейтаграммных соединений. Структура клиента и сервера в этом случае похожа на структуру приложений, использующих соединения типа виртуальный канал, но имеет ряд отличий. Эти отличия состоят в следующем. Прежде всего, при создании сокета необходимо использовать параметры SOCK_DGRAM и IPPROTO_UDP, соответствующие инициализации сокета для передачи дейтаграммного сообщения. Далее, поскольку дейтаграммный канал является односторонним одноразовым, предварительного установления сеанса не требуется, соответственно функции connect() на клиенте и accept() и listen() на сервере использовать не нужно. При этом вызов bind() на сервере необходим.

Для фактической передачи и приема сообщения используются функции sendto() и recvfrom():

```
int sendto( SOCKET s,  const char* buf,   int len,   int flags,
        const struct sockaddr* to,   int tolen  );
int recvfrom(  SOCKET s,  char* buf,   int len,   int flags,
        struct sockaddr* from,   int* fromlen  );
```

Первые четыре параметра этих функций идентичны параметрам функций send() и recv() (это сокет, адрес буфера с передаваемой и получаемыми данными, длина буфера и флаг передачи), пятый и шестой параметры функций соответствуют параметрам функций connect() и accept(). Таким образом, sendto() в определенном смысле комбинирует функциональность пары функций connect() и send(), а recvfrom() – пары accept()) и recv().

Отметим, что sendto() и recvfrom() можно использовать и для TCP-соединений как аналог send() и recv(); в этом случае пятые и шестые параметры игнорируются.

Функция sendto() возвращает количество реально отправленных байтов, а recvfrom() – реально полученных; эти значения могут отличаться от значения параметра len. Для дейтаграмных соединений, как и для виртуального канала, существует ограничение на размер передаваемого одной операцией sendto() блока данных. Так же следует иметь в виду, что для дейтаграмных соединений доставка данных не гарантируется; однако для локальных сетей потеря пакетов не характерна.

Размер максимально допустимого блока данных, передаваемых за одну операцию чтения-записи в сокет можно установить при помощи вызова функции getsockopt():

```
int getsockopt( SOCKET s, int level, int optname, char* optval, int* optlen  );
```

Параметры level и optname определяют категорию настройки сокета, а optval и optlen - указатели на переменные, в которые будут записаны значение параметра и его длина. Существует так же функция setsockopt() с аналогичным синтаксисом, с помощью которой можно изменить некоторые настройки. Подробное описание существующих настроек выходит за пределы настоящего пособия (см. например, [6]).

Ниже приведены (фрагментарно) примеры использования дейтаграммного соединения для серверной и клиентской частей приложения

```
//      DGServer.cpp

SOCKET sock = socket( AF_INET,SOCK_DGRAM,IPPROTO_UDP);
if( sock == INVALID_SOCKET )
{
//              Обработка ошибки
}
struct sockaddr_in baddr;
```

```
memset( &baddr, 0, sizeof( baddr ) );
baddr.sin_family = AF_INET;
baddr.sin_port = htons( NPort );
baddr.sin_addr.s_addr = INADDR_ANY;
if( bind( sock, ( sockaddr* )&baddr,  sizeof( baddr ) ) !=0  )
{
//              Обработка ошибки
}
wprintf( L"Сетевая служба активна на  %S:%d\n",
        inet_ntoa( baddr.sin_addr ),     htons( baddr.sin_port ) );
struct sockaddr_inaddr;
memset( &addr, 0, sizeof( addr ) );
int  len = sizeof( addr );
const   int maxsize = 10000;
char    Buf[maxsize];
memset( Buf, 0, sizeof( Buf ) );
int rec = recvfrom( sock, Buf, sizeof( Buf ), 0,  (sockaddr*) &addr, &len );
if( rec == SOCKET_ERROR  )
{
//              Обработка ошибки
}
wprintf( L"Получено %d байтов от хоста  %S:%d\n",  rec,
                  inet_ntoa( addr.sin_addr ), htons( addr.sin_port ) );
shutdown( sock, 0 );
closesocket( sock );
```

Функция shutdown() используется для отключения сокета от чтения и/или записи данных, в зависимости от значения второго параметра:

int shutdown(SOCKET s, int how);

Поскольку сокет после вызова этой функции повторно использовать не следует, ее явное использование обычно не целесообразно.

```
//      DGClient.cpp

SOCKET sock = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );
if( sock == INVALID_SOCKET )
{
//              Обработка ошибки
}
struct sockaddr_in addr;
memset( &addr, 0, sizeof( addr ) );
addr.sin_family = AF_INET;
addr.sin_port = htons( NPort );
addr.sin_addr.s_addr = inet_addr( IPAddr );
const   int maxsize = 1000;
char    Buf[maxsize];
memset( Buf, 'A', sizeof( Buf ) );
int option;
int optsize = sizeof( option );
if( getsockopt( sock, SOL_SOCKET, SO_MAX_MSG_SIZE,
        (char*)&option, &optsize ) != SOCKET_ERROR )
        wprintf( L"Максимальный размер блока данных %d\n",
                option );
int sent = sendto( sock, Buf, sizeof( Buf ), 0,
        (sockaddr*) &addr, sizeof( addr ) );
wprintf( L"Отправлено %d б. на хост %S:%d\n", sent,
                inet_ntoa( addr.sin_addr ), htons( addr.sin_port ) );
closesocket( sock );
```

В приведенном примере данные пересылаются в одну сторону, - от клиента к серверу. Если сервер должен вернуть клиенту какие-либо данные в качестве ответа, для этой цели следует использовать IP-адрес и номер порта клиента, которые заносятся в пятый параметр функции recvfrom() (в примере – структура addr). Следует иметь в виду, что, даже если на клиенте не использовалась функция bind(), при вызове sendto() или connect() (при использовании виртуального соединения) системой автоматически выделяется свободный порт, и этот порт остается закрепленным за процессом до закрытия сокета.

Литература

1. Чан, Т. Системное программирование на C++ для UNIX [Текст] : / Т. Чан. - Киев: BHV, 1997, - 589 с.
2. Робачевский, А. Операционная система UNIX [Текст] : / А. Робачевский. - СПб.: БХВ-Петербург, 2002.
3. Харт, Джонсон. Системное программирование в среде Windows [Текст], 3-е издание : / Джонсон Харт. - М.- Эком. 2005.
4. Олифер, В., Н.А. Компьютерные сети. Принципы, технологии, протоколы [Текст] : / В. Г. Олифер, Н.А. Олифер. - Санкт-Петербург: Питер, 2001г. - 765 с.
5. Стивенс, Ричард. Протоколы TCP/IP. Практическое руководство [Текст]: / Ричард Стивенс. – М. – изд. BHV. 2003 г. – 672 с.
6. Джонс, Э. Программирование в сетях Microsoft Windows [Текст] : / Э.Джонс, Д. Оланд. : - СПб.- изд. Питер. 2001 г. - 608 с.