Лабораторная работа № 5-6

**Приложение для организации обмена данными c FTP/HTTP сервером. Сокеты без блокировки.**

*Цель работы:* **1.** Изучить принципы работы с HTTP/FTP сервером с использованием Winsock2. **2.** Изучить механизм работы с не блокирующими сокетами.

*Продолжительность работы* – 4 ч.

**Принцип работы с FTP/HTTP сервером.**

Для примера, рассмотрим следующее техническое задание: необходимо написать программу, которая получает с удалённого HTTP сервера нужную нам HTML страничку. Проще говоря, "достать" с нужного нам сайта, кусочек HTML кода, по заранее известному адресу. Например, взять HTML страничку “**/enp/rfc/rfc768.txt”** с сайта “**www.networksorcery.com”** - значит выполнить обращение по адресу “**www.networksorcery.com/enp/rfc/rfc768.txt”**. То есть, мы пишем программу, отдалённо напоминающую интернет-браузер.

В качестве протокола обмена данными с сервером можно использовать ***HTTP*** (стандарт RFC 2616), ***FTP*** (RFC 959)и другие протоколы. Мы будем использовать ***HTTP*** протокол. В принципе, Winsock все равно, какой протокол вы используете для связи с сервером. Его задача - доставить данные серверу и получить ответ. В данной лабораторной работе мы не будем останавливаться на особенностях протокола ***HTTP***, и вам придется принять на веру все, что касается этого протокола в коде программы. Всё что нам нужно знать для того, что бы установить соединение с каким-нибудь сервером, это его адрес и порт. Например, 66.27.58.123 - это адрес ***HTTP*** сервера “Network Sorcery”. Протокол ***HTTP*** использует порт ***80***. Протокол ***FTP - 21***. Проблема в том, что некоторые сервера имеют динамические адреса. В таком случае их адреса вычисляются с помощью специальных функций Winsock API.

Для того, что бы узнать IP адрес машины, зная ёё имя, существует функция ***gethostbyname***. Для получения имени машины по ёё адресу используем функцию ***gethostbyaddr***. Рассмотрим эти функции подробнее.

**struct** hostent FAR **\***gethostbyname(

**const** **char** FAR **\***name

);

Пример вызова:

hostent\* d\_addr; *// Структура, в которую будет*

*// Помещен IP адрес, при возврате.*

hostent\* hn = gethostbyname ("www.networksorcery.com ");

...

Если функция отработала успешно, то структура ***hn*** содержит нужные нам данные, иначе gethostbyname возвращает ***NULL***;

Рассмотрим структуру ***hn***.

**struct** hostent {

**char** FAR **\*** h\_name; // Официальное имя машины

**char** FAR **\*** FAR **\*** h\_aliases; *// Массив альтернативных имен*

// машины (заканчивающийся 0)

**short** h\_addrtype; // Тип адреса (AF\_INET...)

**short** h\_length; // Длина адреса в байтах

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

// (заканчивающийся 0)

};

Итак, после вызова ***gethostbyname*** нас интересуют следующие поля этой структуры:

***h\_name*** - можем узнать официальное имя машины :)

***h\_addrtype*** - тип адреса (для TCP/IP должно быть AF\_INET)

***h\_length*** - длинна адреса (для TCP/IP V4 - 4 байта)

***h\_addr\_list*** - массив адресов (заканчивающийся 0).

Для получения первого адреса в массиве, можно использовать макрос ***h\_addr\_list[0]***.

Для того, что бы получить адрес интересующей нас машины, необходимо вызвать ***gethostbyname*** (имя машины) и проанализировав полученные назад данные, выбрать из массива ***h\_addr\_list*** ее адрес. Адрес машины сохраним в структуре ***sockaddr\_in****.*

Полный пример:

sockaddr\_in adr;

...

hostent**\*** d\_addr = gethostbyname ("www.networksorcery.com ");

adr.sin\_addr.S\_un.S\_addr = \*(DWORD\* ) hn-›h\_addr\_list[0];

...

Функция ***gethostbyaddr*** получая на вход адрес, тип адреса и его длину, возвращает имя машины.

**struct** HOSTENT FAR \* gethostbyaddr(

**const** **char** FAR **\***addr, *// Адрес машины (в сетевом виде)*

**int** len, *// Длинна адреса*

**int** type *// Тип адреса*

);

Пример:

DWORD a = inet\_addr ("192.168.0.4"); // Адрес машины в сетевом

// формате

hn = gethostbyaddr ((char\* )&a, 4, AF\_INET);

Если функция выполнилась успешно, поле ***h\_name*** структуры ***hn*** будет содержать имя машины.

Вернёмся к написанию программы.

Структура программы такова:

1. Узнаём IP адрес сервера.

2. Устанавливаем соединение.

3. Посылаем серверу запрос (в HTTP формате).

4. Получаем ответ и выводим его на экран

5. Закрываем соединение и завершаем работу.

Приступим, но сразу хочу оговорить одну важную деталь: данный код будет работать только на машинах имеющих "прямой" выход и интернет, то есть этот код не буде работать, если вы сидите за прокси-сервером и т.п.

Вот исходный код программы.

**#include** "stdafx.h"

**#include** "winsock2.h"

*// Необходимые макро-определения*

**#define** request "GET /enp/rfc/rfc768.txt HTTP/1.0\r\nHost: " \

"www.networksorcery.com\r\n\r\n" *// HTML*

*// запрос.*

**#define** MAX\_PACKET\_SIZE 1024

*// Напечатать код последней ошибки*

**void** print\_WSAGetLastError(const char\* function\_name)

{

**int** res = WSAGetLastError ();

printf ("\nERROR: \tFunction \"%s\" call failed, \n \t WSAGetLastError() return error code: dec = [%d], hex = [0x%0x]\n", function\_name ,res, res);

}

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

{

WSADATA ws;

SOCKET s;

sockaddr\_in adr;

hostent\* hn;

**char** buff [MAX\_PACKET\_SIZE];

*// Инициализация*

**if** (WSAStartup (0x0101, &ws) != 0)

{

print\_WSAGetLastError("WSAStartup");

**return** -1;

}

*// Создаём сокет*

**if** (INVALID\_SOCKET == (s = socket (AF\_INET, SOCK\_STREAM,

IPPROTO\_TCP) ) )

{

print\_WSAGetLastError("socket");

**return** -1;

}

*// Получаем адрес*

**if** (NULL == (hn = gethostbyname("www.networksorcery.com")))

{

print\_WSAGetLastError("gethostbyname");

**return** -1;

}

*// Заполняем поля структуры adr для использование ее в*

*// connect*

adr.sin\_family = AF\_INET;

adr.sin\_addr.S\_un.S\_addr = \*(DWORD\* ) hn->h\_addr\_list[0];

adr.sin\_port = htons (80);

*// Устанавливаем соединение с сервером*

**if** (SOCKET\_ERROR == connect( s, (sockaddr\*) &adr,

**sizeof**(adr) ) )

{

print\_WSAGetLastError("connect");

**return** -1;

}

*// Посылаем запрос серверу*

**if** (SOCKET\_ERROR == send (s, &request, **sizeof**(request), 0))

{

print\_WSAGetLastError("send");

**return** -1;

}

*// Ждём ответа*

**int** len = recv (s, (char \*) &buff, MAX\_PACKET\_SIZE, 0);

**if** ( (len == SOCKET\_ERROR) || (len == 0) )

{

print\_WSAGetLastError("recv");

**return** -1;

}

*// Выводим ответ сервера*

**for** (**int** i = 0; i < len; i++)

printf ("%c", buff [i]);

printf("\n");

*// Закрываем соединенеие*

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

{

print\_WSAGetLastError("closesocket");

**return** -1;

}

**return** 1;

}

Попробуйте открыть ссылку “**http://www.networksorcery.com/ enp/rfc/rfc768.txt”** - в окне вашего браузера должна появится информация с описанием “*User Datagram Protocol*”. Такой же ответ должна выдать и программа, после системной информации сервера.

**Прием фрагментированных данных.**

В предыдущем примере программы, есть такой фрагмент кода:

...

*// Ждём ответа*

**int** len = recv (s, (**char** **\***) **&**buff, MAX\_PACKET\_SIZE, 0);

...

На первый взгляд вроде бы всё нормально. Но это не так. В вызове функции ***recv*** присутствует параметр ***MAX\_PACKET\_SIZE***, который определяет длину буфера приёма данных. То есть, Winsock протоколы (TCP/UDP) могут "отправить" пакет размером и 65535 байт. Однако, реальный размер IP пакета, может быть меньше передаваемых данных для конкретного типа сети (величина MTU - размер наибольшего допустимого кадра в локальной сети или глобальном канале) и поэтому данные фрагментируются (см. описание TCP/IP протокола). При этом, фрагментированные данные могут идти с задержками. Какое отношение имеет данная особенность к нашей программе? Всё очень просто. Если мы запросим у удалённого Web-сервера не такой маленький кусочек HTML-кода, как в нашем примере, а немного побольше, то мы получим только первую порцию данных от сервера. Остальная часть данных будет, скорее всего, утеряна. Выход из данного положения очень прост. Он основан на знании принципа работы ***HTTP*** и ***TCP/IP*** протоколов, плюс некоторых особенностей Winsock. В нашем случае Web-сервер, передав последнюю порцию данных, закроет соединение. А функция ***recv*** в таком случае, после получения последнего фрагмента данных, возвращает ноль. Вывод напрашивается сам: мы должны читать входящие данные до тех пор, пока функция ***recv*** не вернет ноль.

Вот новый вариант кода:

*// Ждём ответа*

**int** len = 0;

**do**

{

**if** (SOCKET\_ERROR == (len = recv (s, (**char** **\***) **&**buff,

MAX\_PACKET\_SIZE, 0) ) )

{

print\_WSAGetLastError("recv");

**return** -1;

}

**for** (**int** i = 0; i < len; i++)

printf ("%c", buff [i]);

}

**while** (len!=0); // *Получаем данные по частям, пока не len != 0.*

При такой организации считывания данных, потерь не будет.

**Проблема блокировка сокета. Решение проблемы.**

Если вы выполняли программу из примера пошагово, то могли заметить, что некоторые Winsock-функции ожидают завершения выполняемых ими операций. Особенно надолго "подвисает" функция ***recv***. Другими словами, выход из функции не происходит до момента завершения текущей операции. Эта особенность не очень хорошо подходит для программ, которые кроме получения/отправки данных должны выполнять еще множество других действий (отслеживание состояния системы меню, вывод информации, опрос других устройств ввода/вывода) Избежать этого можно многими способами. Можно обойтись средствами мультизадачности, и процедуру обмена данными "повесить" на отдельную ветвь. А можно решить проблему средствами Winsock. В любом случае, выбор конкретного метода остаётся за программистом. Наша же задача, разобраться с механизмом блокировки TCP-сокетов в Winsock. Итак, рассмотрим два метода устранения проблемы блокировки:

***1. Функция ioctlsocket.***

Функция ***ioctlsocket*** позволяет менять/получать режим ввода/вывода конкретного сокета.

**int** ioctlsocket(

SOCKET s, // Сокет [in]

**long** cmd, // Комманда [in]

u\_long FAR \*argp // Параметр/значение [in/out]

);

Для перевода сокета в не блокируемое состояние (***nonblocking mode***) используется команда ***FIONBIO***. ***argp*** должно указывать на ненулевое значение.

Пример:

BOOL l = TRUE;

**if** (SOCKET\_ERROR == ioctlsocket (s, FIONBIO, (unsigned long\* ) &l) )

{

print\_WSAGetLastError("ioctlsocket");

**return** -1;

}

Кроме команды ***FIONBIO*** существуют команды ***FIONREAD*** и ***SIOCATMARK***. Если коротко, то ***FIONREAD*** позволяет получить количество байт информации, поступившей в буфер на данный момент операции чтения, а ***SIOCATMARK*** - используется при работе с OOB данными. На данный момент нас интересует только команда ***FIONBIO***. Остальные команды будем рассматривать более подробно по мере надобности в следующих статьях.

Итак, вернёмся к нашему примеру. После вызова ***ioctlsocket*** сокет ***s*** стал не блокируемым, то есть, Winsock-функции для этого сокета не дожидаются окончания операций ввода/вывода, что в свою очередь не вызывает нежелательных пауз в работе программы. Однако, не всё так просто. Например, возврат из функции ***recv*** может произойти до момента получения данных. Как определить, что текущая операция ввода/вывода полностью завершена? Способ есть. Имя ему - ***WSAEWOULDBLOCK***. Что это такое? ***WSAEWOULDBLOCK*** - это код ошибки, которую возвращают Winsock-функции для ***nonblocked*** сокета, если текущая операция не завершена. То есть, если функция ***revc*** вернула этот код ошибки, значит, данные еще не готовы для чтения, и операцию придется повторить позже. В таком случае, ваша программа может выполнять другие действия, попутно проверяя, не завершена ли текущая операция ввода/вывода.

Пример:

// Устанавливаем nonblocked mode

**BOOL** l = TRUE;

**if** (SOCKET\_ERROR == ioctlsocket (s, FIONBIO, (**unsigned long**\* ) &l))

{

print\_WSAGetLastError("ioctlsocket");

**return** -1;

}

// Получаем данные...

**int** len = 0;

**do**

{

**if** (SOCKET\_ERROR == (len = recv (s, (**char** \*) &buff, MAX\_PACKET\_SIZE, 0) ) )

{

**int** res = WSAGetLastError();

**if** (res != WSAEWOULDBLOCK)

{

print\_WSAGetLastError("recv");

**return** -1;

}

**else** len = 1;

}

**else**

**for** (**int** i = 0; i<len; i++)

printf ("%c", buff [i]);

// Если recv вернул ошибку WSAEWOULDBLOCK, то можем

// продолжать работу.

// Перед следующим вызовом recv (...) делаем все

// необходимые нам действия...

}

**while** (len!=0);

***2. Функция select***

Функция ***select*** позволяет определить текущее состояние одного или более сокетов. То есть, из какого-то входящего множества сокетов, она формирует выходящее множество сокетов, готовых к операциям чтения/записи/....

**int** select(

**int** nfds, // Не используется (оставлен для совместимости)

fd\_set FAR **\***readfds, // Множество сокетов, проверяемых на

// готовность к чтению

fd\_set FAR **\***writefds, // Множество сокетов, проверяемых на

// готовность к отсылке

fd\_set FAR **\***exceptfds, // Множество сокетов, проверяемых на

// ошибку/OOB данные

**const** **struct** timeval FAR \*timeout // Таймаут проверки

);

Каждый из параметров ***readfds***, ***writefds***, ***exceptfds***, ***timeout*** есть необязательным, и может быть проигнорирован (установлен в ***NULL***). В случае ***readfds/writefds/exceptfds*** == ***NULL*** проверка на опущенные типы сокетов просто не будет производиться. В случае ***timeout*** ==***NULL***, функция ***select*** вызовет блокировку (до первого готового к вводу/выводу сокета). Функция возвращает общее количество сокетов (во всех заданных множествах ***readfds/writefds/exceptfds***), готовых к операциям ввода/вывода.

Параметры ***readfds/writefds/exceptfds*** - указатели на тип ***fd\_set*** (представляющий собою множество сокетов). Для работы с этим типом данных объявлены такие макросы:

FD\_CLR (s, \*set) // Удаляет дескриптор s из set.

FD\_ISSET(s, \*set) // Возвращает ненулевое значение, если s

// присутствует в set. Иначе, возвращает ноль.

FD\_SET(s, \*set) // Добавляет s к set.

FD\_ZERO(\*set) // Очищает множество set

Параметр ***timeout*** - указатель на структуру ***timeval***, позволяет задать таймаут, в течении которого сокеты будут проверяться на готовность.

**struct** timeval {

**long** tv\_sec; // Секунды

**long** tv\_usec; // Микросекунды

};

С помощью функции ***select*** и этого набора макросов, мы можем проверять конечное множество сокетов на готовность к считыванию/отсылке данных, выполнения ***connect***, на предмет входящих соединений, наличия OOB сообщений и т.п. На данном этапе нас интересует проверка сокета на возможность считывания данных, поэтому пока ограничимся самым простым вызовом ***select***. Для этого нам необходимо поместить наш сокет в множество на которое будет указывать ***readfds*** (в примере это ***read\_s***), задать ***timeout*** и выполнить ***select***.

Пример:

// Ждём ответа

**int** len, res;

fd\_set read\_s;

timeval time\_out;

time\_out.tv\_sec = 0; time\_out.tv\_usec = 500000; // 0.5 sec.

**do**

{

FD\_ZERO (&read\_s); // Обнуляем множество

FD\_SET (s, &read\_s); // Заносим в него наш сокет

**if** (SOCKET\_ERROR == (res = select (0, &read\_s, **NULL**, **NULL**, &time\_out) ) )

{

print\_WSAGetLastError("select");

**return** -1;

}

**if** ((res!=0) && (FD\_ISSET (s, &read\_s)) )

{

// Данные готовы к чтению...

**if** (SOCKET\_ERROR == (len = recv (s, (**char** \*) &buff, MAX\_PACKET\_SIZE, 0) )

{

print\_WSAGetLastError("recv");

**return** -1;

}

// Выводим их на экран.

**for** (**int** i = 0; i<len; i++)

printf ("%c", buff [i]);

}

// Делаем все необходимые нам действия...

}

**while** (len!=0);...

Итак, мы рассмотрели некоторые нюансы работы с сокетами и простейшие способы решения проблемы блокировки.

***Приложение 1***

**Коды ошибок TCP/IP**

| **Код / Обозначение** | **Описание** |
| --- | --- |
| 6  WSA\_INVALID\_HANDLE | *Specified event object handle is invalid* (Неверный указатель объекта события). |
| 8  WSA\_NOT\_ENOUGH  \_MEMORY | *Insufficient memory available* (Недостаточно памяти). |
| 87  WSA\_INVALID  \_PARAMETER | *One or more parameters are invalid* (Недопустимые параметры).  Приложение, использующее функцию WinSock, напрямую обращается к функции Win32. Функция Win32 указывает на проблему с одним или несколькими параметрами. |
| 996  WSA\_IO\_INCOMPLETE | *Overlapped I/O event object not in signaled state* |
| 997  WSA\_IO\_PENDING | *Overlapped operations will complete later* (Перекрывающиеся операции будут завершены позже).  Приложение инициировало перекрывающуюся (overlapped) операцию, которая не может быть завершена немедленно. О завершении операции будет сигнализирвоано позднее. |
| 10004  WSAEINTR | *Interrupted function call* (Прерван вызов функции).  Блокирующая операция прервана вызовом WSACancelBlockingCall(). |
| 10013  WSAEACCESS | *Permission denied* (Доступ запрещен).  Попытка доступа к сокету способом, запрещенным привилегиями доступа. Например, использование широковещательного адреса в функции sendto() без установки соответствующего разрешения с помощью setsockopt(SO\_BROADCAST). |
| 10014  WSAEFAULT | *Bad address* (неверный адрес).  Система обнаружила неверный указатель на адрес при попытке использовать его в вызове функции. Эта ошибка происходит при передачи приложением неверного указателя, или если размер буфера слишком мал - например, если длина аргумента, представляющего собой структуру типа sockaddr, меньше, чем sizeof(struct sockaddr). |
| 10022  WSAEINVAL | *Invalid argument* (Недопустимый аргумент).  Передан недопустимый аргумент( например, указан неверный уровень в функции setsockopt()). В некоторых случаях это также ссылается на состояние сокета; например, вызов accept() на сокете, который не слушает (listen()) |
| 10024  WSAEMFILE | *Too many open files* (Слишком много открытых файлов).  Слишком много открытых сокетов. Каждая реализация имеет свое максимальное количество открытых сокетов, либо глобально, либо для каждого процесса/потока. |
| 10035  WSAEWOULDBLOCK | *Resource temporarily unavailable* (Ресурс временно недоступен).  Эта ошибка возвращается операциями с неблокирующими сокетами, которые не могут быть немедленно завершены. Это не фатальная ошибка. Обычно WSAEWOULDBLOCK возвращается как результат вызова connect() на неблокирующем сокете SOCK\_STREAM, поскольку для установления соединения требуется некоторое время. |
| 10036  WSAEINPROGRESS | *Operation now in progress* (Операция выполняется).  Выполняется блокирующая операция. Сокеты Windows позволяют только одну блокирующую операцию на задачу или поток. Если выполняется вызов любой другой функции (независимо от того, ссылается она на этот или другой сокет), то возникает эта ошибка. |
| 10037  WSAEALREADY | *Operation already in progress* (Операция уже осуществляется).  Неблокирующий сокет, на котором предпринята операция, уже выполняет операцию. Такая ошибка происходит, например, при повторном вызове *connect*() на неблокирующем сокете, который находится в процессе подключения или отмены асинхронного запроса *WSAAsyncGetXbyY*(). |
| 10038  WSAENOTSOCK | *Socket operation on non-socket.*  Попытка операции на чем-то, что не является сокетом. Либо это операция с указателем, который не ссылается на допустимый сокет, или, в случае select(), недопустимый член fd\_set. |
| 10039  WSAEDESTADDRREQ | *Destination address required* (Требуется указание удаленного адреса).  При операции с сокетом не указан требуемый адрес. Например, эта ошибка возвращается при вызове sendto() с адресом INADDR\_ANY. |
| 10040  WSAEMSGSIZE | *Message too long* (Слишком длинное сообщение).  Сообщение, посланное в датаграммный сокет, превышает длину внутреннего буфера или другие сетевые огранчиения, или буфер, используемый для приема датаграмм, меньше чем датаграмма. |
| 10041  WSAEPROTOTYPE | *Protocol wrong type for socket* (Неверный тип протокола для сокета). При вызове функции socket() указан протокол, который не поддерживает семантику запрошенного типа сокета. Например, протокол UDP нельзя указывать с типом сокета SOCK\_STREAM |
| 10042  WSAENOPROTOOPT | *Bad protocol option* (Неверная опция протокола).  При вызове getsockopt() или setsockopt() указана неизвестная, недопустимая или неподдерживаемая опция. |
| 10043  WSAEPROTONOSUPPORT | *Protocol not supported* (Пртокол не поддерживается)..  Запрашиваемый протокол не сконфигурирован в системе или не существует его реализации. |
| 10044  WSAESOCKTNOSUPPORT | *Socket type not supported* (Неподдерживаемый тип сокета). |
| 10045  WSAEOPNOTSUPP | *Operation not supported* (Операция не поддерживается).  Предпринятая операция не поддерживается для ссылающегося объекта; например, попытка принять соединение на сокете датаграмм. |
| 10046  WSAEPFNOSUPPORT | *Protocol family not supported* (Семейство протоколов не поддерживается).  Семейство протоколов не сконфигурировано в системе или для него вообще не существует реализации. Эта ошибка слегка отличается от WSAEAFNOSUPPORT, но в большинстве случаев означает то же самое. Все функции Windows Sockets, возвращающие эту ошибку, возвращают WSAEAFNOSUPPORT |
| 10047  WSAEAFNOSUPPORT | *Address family not supported by protocol family* (Семейство адресов не поддерживается семейством портов).  Адрес не совместим с используемым протоколом. Все создаваемые сокеты ассоциируются с некоторым семейством адресов (например, AF\_INET для протоколов интернет) и общим типом протокола (например, SOCK\_STREAM). Эта ошибка возникает, если в вызове socket() указан неверный протокол или указано неверное семейство адресов. |
| 10048  WSAEADDRINUSE | *Address already in use* (Адрес уже используется).  Обычно разрешено только одно использование адреса сокета (проткол/адрес IP/порт). Эта ошибка возникает, когда приложение пытается привязаться к сокету функцией bind(), но комбинация *адрес IP/порт* уже используется существующим сокетом, или сокет не был корректно закрыт, или продолжается процесс закрытия сокета. Для серверных приложений, требующих привязки нескольких сокетов к одному и тому же номеру порта следует использовать setsockopt(SO\_REUSEADDR). Клиентские приложения обычно не используют bind() - функция connect() автоматически выбирает неиспользуемый порт. |
| 10049  WSAEADDRNOTAVAIL | *Cannot assign requested address* (Невозможно назначить требуемый адрес). Запрашиваемый адрес недопустим в его контексте. Обычно это возникает при вызове bind() для адреса, который недопустим для локальной машины, или вызов connect()/sendto() с адресом или портом, недоспутимыми для удаленной машины (например, номер порта 0). |
| 10050  WSAENETDOWN | *Network is down* (Сеть отключена).  Операция с сокетом обнаружила мертвую сеть. Это может означать серьезную проблему в сети, например, проблему со стеком протокола WinSock DLL, с сетевым интерфейсом, с локальной сетью |
| 10051  WSAENETUNREACH | *Network is unreachable (*Сеть недостижима).  Попытка осуществить операцию с сокетом на недостижимой сети. Обычно это означает, что локальные программы не имеют маршрута к удаленному хосту. |
| 10052  WSAENETRESET | *Network dropped connection on reset* (Сеть сбросила соединение).  Хост, к которому вы подключены, перезагрузился или на нем произошла авария. Эта ошибка может возвращаться функцией setsockopt() при попытке установить SO\_KEEPALIVE на соединении, установление которого уже завершилось неудачей. |
| 10053  WSAECONNABORTED | *Software caused connection abort* (Программа вызвала аварийное завершение соединения).  Установленное соединение прервано программным обеспечением на вашей хост-машине, возможно вследствие тайм-аута передачи данных или ошибки протокола. |
| 10054  WSAECONNRESET | *Connection reset by peer* (Соединение сброшено удаленной системой).  Существующее соединение принудительно закрыто удаленной стороной. Обычно это случается в случае неожиданного останова приложения на удаленной стороне, при перезагрузке удаленной машины, или в случае, когда удаленный хост использует "жесткое закрытие" ( setsockopt(SO\_LINGER)) удаленного сокета. |
| 10055  WSAENOBUFS | *No buffer space available* (Закончились буферы).  Невозможно осуществить операцию с сокетом, поскольку системе не хватает буферного пространства или переполнена очередь. Это означает, что WinSock временно не хватает буфров. Это не должно вызывать проблем, если не продолжается долгое время. |
| 10056  WSAEISCONN | *Socket is already connected* (Сокет уже подключен).  На уже подключенный сокет сделан запрос соединения. Некоторые реализации также возвращают эту ошибку, если sendto() вызывается на подключенном сокете SOCK\_DGRAM. Для сокетов SOCK\_STREAM параметр to в функции sendto() игнорируется, хотя в другиз реализациях это допустимо. |
| 10057  WSAENOTCONN | *Socket is not connected* (Сокет не подключен).  Была предпринята попытка передать или принять данные через неподключенный сокет или попытка посылки датаграммы с помощью sendto() без указания адреса. Эту ошибку может также вернуть любой другой тип операции, например, установка SO\_KEEPALIVE в setsockopt() на сброшенном соединении. |
| 10058  WSAESHUTDOWN | *Cannot send after socket shutdown* (Невозможно послать данные после закрытия сокета). |
| 10059  WSAETOOMANYREFS | *Too many references* (Слишком много ссылок).  На какой-то объект ядра создано слишком много ссылок, превышающих системные ресурсы. |
| 10060  WSAETIMEDOUT | *Connection timed out* (Истекло время ожидания соединенеия).  Попытка соединения завершилась неудачей, поскольку удаленная сторона не ответила в течении определенного времени. |
| 10061  WSAECONNREFUSED | *Connection refused* (Соедиенение отклонено).  Невозможно установить соединение, поскольку удаленная машина его отвергает. Обычно это происходит при попытке подключиться к службе, которая не выполняется на удаленной машине. |
| 10062  WSAELOOP | *Too many levels of symbolic links*  *A pathname lookup involved more than eight symbolic links.* (*Too many links were encountered in translating a pathname*.) |
| 10063  WSAENAMETOOLONG | *Name too long* (слишком длинное имя) |
| 10064  WSAEHOSTDOWN | *Host is down*.  Операция с сокетом неуспешна, поскольку удаленный хост не отвечает. Операция с сокетом обнаружила мертвый хост. Сетевая активность на локальном хосте не инициируется. Это чаще всего происходит при ошибке WSAETIMEDOUT. |
| 10065  WSAEHOSTUNREACH | *No route to host* (Нет маршрута к хосту).  Попытка обращения к хосту, к которому невозможно определить маршрут. См. WSAENETUNREACH. |
| 10066  WSAENOTEMPTY | *Directory not empty* |
| 10067  WSAEPROCLIM | *Too many processes* (Слишком много процессов).  Реализация Windows Sockets может иметь предельное количество приложений, способных работать одновременно. WSAStartup() может завершиться неудачей, если этот предел достигнут. |
| 10068  WSAEUSERS | *Too many users*.  Слишком много пользователей. |
| 10069  WSAEDQUOT | *Disk quota exceeded*.  Превышена дисковая квота |
| 10070 | *Stale NFS file handle.* Попытка получить доступ к файлу, находящемуся в NFS, который стал недоступным. Возможно, файл удален на сервере NFS. |
| 10091  WSASYSNOTREADY | *Network subsystem is unavailable* (Сетевая подсистема недоступна).  Эта ошибка возвращается функцией WSAStartup(), если Windows Sockets не может вызвать нужную функцию в данное время, поскольку низлежащая система, предоставляющая сетевые службы, недоступна. Пользователи должны проверить следующее:  Файл WINSOCK.DLL находится в текущем маршруте поиска  WINSOCK.DLL того же производителя, что и стек протоколов. Их нельзя смешивать.  Используется одновременно только одна реализация WinSock. Если в системе есть несколько реализаций WINSOCK DLL, убедитесь, что загружена нужная версия.  Убедитесь, что инсталлированы и сконфигурированы все необходимые компоненты реализации WinSock |
| 10092  WSAVERNOTSUPPORTED | *WINSOCK.DLL version out of range* (Неверная версия файла WINSOCK.DLL).  Текущая реализация WinSock не поддерживает версию спецификации, запрашиваемую приложением. Убедитесь, что у вас нет старых версий файла WINSOCK.DLL или обратитесь к поставщику стека за получением обновленной версии. |
| 10093  WSANOTINITIALISED | *Successful WSAStartup() not yet performed* (Не сделан вызов функции WSAStartup()).  Либо приложение еще не сделало вызов WSAStartup(), либо вызов WSAStartup() завершился неудачей. Приложение могло также потребовать доступ к чужому сокету, владельцем которого не является. |
| 10094  WSAEDISCON | *Graceful shutdown in progress* (Процесс аккуратного закрытия).  Возвращается функциями recv(), WSARecv() для обозначения того, что удаленная сторона инициировала процедуру аккуратного закрытия соединения. |
| 11001  WSAHOST\_NOT\_FOUND | *Host not found* (Хост не найден).  Указанный хост неизвестен: имя не является официальным hostname или псевдонимом (alias), или его не удается найти в запрашиваемых базах данных. Эта ошибка может возвращаться при запросах протоколов и служб; она указывает на то, что указанное имя нельзя найти в соответствующей базе данных. |
| 11002  WSATRY\_AGAIN | *Non-authoritative host not found.*  Обычно это временная ошибка, возникающая в процессе разрешения имени и означающая, что локальный сервер не получил ответа от авторитетного сервера. Последующая попытка может быть успешной. |
| 11003  WSANO\_RECOVERY | *This is a non-recoverable error* (Невосстановимая ошика).  При просмотре базы данных произошла невосстановимая ошибка. Это может случиться при отсутствии файлов базы данных - файлов hosts, services или protocols, или запрос DNS вернул ошибку. |
| 11004  WSANO\_DATA | Запрашиваемое имя правильное и найдено в базе данных, но имеет неожиданный тип данных. Обычно это происходит при трансляции имени в адрес (функциями gethostbyname() или WSAAsyncGetHostByName()), которое использует DNS. Возвращается запись MX, а не запись A, что свидетельствует о том, что хост существует, но напрямую не достижим. |