Лабораторная работа №7
Передача данных между процессами. Часть II
(4 часа)

Содержание: структура клиентского и серверного сетевого приложений Windows Sockets 2 на базе потоковых сокетов.

Цель: изучить основы построения сетевых приложений на базе Windows Sockets 2, получить навыки использования основных функции API WinSock2 при разработке TCP-сервера и TCP-клиента.

Еще одним средством реализации обмена данными между процессами, выполняющимися на различных компьютерах, является API Windows Sockets или сокращенно WinSock. Этот интерфейс унаследовал основные функции от Berkeley Sockets и поэтому может использоваться для организации взаимодействия не только с компьютерами под управлением Windows, но и с работающими под управлением других операционных систем, например, ОС семейства Unix.

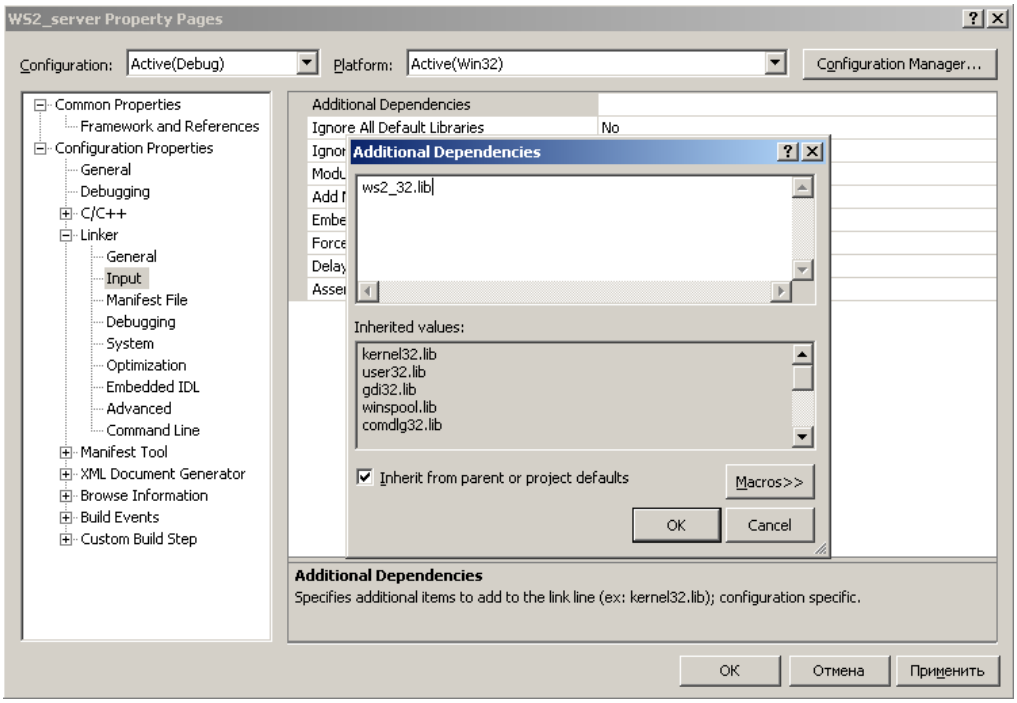
За доступ к возможностям Windows Sockets отвечает динамическая библиотека WS2_32.dll. Для её использования к проекту необходимо подключить основной заголовочный файл Windows Sockets 2 – `winsock2.h`, а также библиотеку импорта WS2_32.lib.

Сделать это можно несколькими способами:

1) при помощи директивы:

```
#pragma comment(lib, "WS2_32.lib")
```

2) через графический интерфейс Visual Studio в свойствах проекта. Для VS2008 это Configuration Properties | Linker | Input:



Перед использованием процесс должен эту динамическую библиотеку инициализировать, указав требуемую версию интерфейса функцией WSAStartup (для этой и других специфических для WinSock функции префикс WSA означает Windows Sockets Asynchronous):

```
int WSAStartup (
     _In_          WORD wVersionRequired,
     _Out_         LPWSADATA lpWSAData);
```

здесь входной параметр wVersionRequired задаёт старшую версию библиотеки WinSock, которую требуется использовать. При этом младший байт указывает основной номер версии, а старший байт – дополнительный. Современные версии Windows используют версию 2.2 (версии 1.0 и 1.1 считаются устаревшими).

Основной и дополнительный номера версии Windows Sockets задаются младшим и старшим байтами wVersionRequired. Чаще всего версию задают при помощи макроса MAKEWORD, например так MAKEWORD(2,2). Также можно напрямую указать нужное значение, например, 0x0202 для текущей версии (2.2).

Параметр lpWSAData при вызове функции должен содержать указатель на структуру WSADATA, в которой возвращается информация о настройках библиотеки Windows Sockets (http://msdn.microsoft.com/en-us/library/windows/desktop/ms741563%28v=vs.85%29.aspx)

В случае успешного завершения функция возвращает 0, в случае ошибки – её код (вызов WSAGetLastError или GetLastError не требуется).

По окончании использования функций из библиотеки Windows Sockets нужно отключить библиотеку вызовом WSACleanup:

```
int WSACleanup(void);
```

Функции WSAStartup и WSACleanup вызываются в обязательном порядке в любой программе, использующей WinSock.

После инициализации библиотеки WinSock и клиент и сервер создают сокет вызовом функции socket:

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

Тип данных SOCKET в Windows Sockets аналогичен типу данных HANDLE и описывает одну сторону соединения (два сокета в свою очередь описывают сетевое соединение). Подобно значению типа HANDLE SOCKET может использоваться в функциях ReadFile и WriteFile, хотя чаще всего для обмена данными по сети используются send и recv.

Параметр af (address family) задаёт семейство адресов, используемых программой. Чаще всего значением этого параметра будут константы AF_INET (для IPv4) или AF_INET6 (соответственно, для IPv6).

type – задаёт тип создаваемого сокета. Чаще всего используются значения SOCK_STREAM для сокетов ориентированных на соединение

(потоковых) или SOCK_DGRAM для сокетов без установления соединения - дейтаграммных (будут рассмотрены в следующей лабораторной работе). Эти типы сокетов по их работе можно сравнить соответственно с именованными каналами и мэйлслотами, рассмотренными ранее.

Параметр protocol – задаёт используемый при передаче данных протокол. Для данной лабораторной работы будет использоваться значение, задаваемое символьной константой IPPROTO_TCP. Также можно указать в качестве значения 0 и нужный протокол на основании семейства адресов и типа сокета будет выбран автоматически.

В случае ошибки функция вернет значение INVALID_SOCKET или при успешном завершении – дескриптор созданного сокета.

Дальнейшие вызовы Windows Sockets различаются для сервера (стороны сетевого взаимодействия, принимающей запросы на создание сетевого соединения) и клиента (стороны, выполняющей подключение).

Сервер

Краткое описание дальнейших действий сервера выглядит так:

– связываем созданный сокет с нужным сетевым интерфейсом компьютера и некоторым портом функцией bind;
– начинаем прослушивать входящие соединения (listen).

Для каждого входящего соединения будет повторяться следующая последовательность:

– принимаем входящее соединение и получаем новый сокет, через который можно обмениваться данными с клиентом (accept);
– выполняем приём и передачу данных функциями recv/send;
– закрываем соединение с клиентом (shutdown) и сокет, использованный для обмена данными (closesocket);
– возвращаемся к приёму следующего входящего подключения.

Рассмотрим подробнее названные функции.

Функция bind устанавливает для сокета связь с определенным сетевым интерфейсом (или всеми интерфейсами) и **номером порта**. Это аналогично заданию имени у именованного канала.

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

Здесь s – несвязанный сокет, возвращенный функцией socket;

name – структура, задающая семейство адресов и соответствующую адресную информацию (номер порта и привязку к определенному интерфейсу);

namelen – задаёт размер структуры в параметре name. Чаще всего задаётся как sizeof(sockaddr).

В случае успешного выполнения функция возвращает 0, иначе – SOCKET_ERROR.

Структура sockaddr в WinSock определена следующим образом:

```
struct sockaddr {
        ushort  sa_family;
        char    sa_data[14];
};
```

В этой структуре поле sa_family задаёт семейство адресов, а второе поле зависит от семейства и используется для хранения адресной информации.

Для AF_INET (IPv4) за хранение нужной информации отвечает структура sockaddr_in, представленная в API типом данных SOCKADDR_IN:

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

Здесь

sin_family должно быть равно AF_INET,

параметр sin_port – задаёт прослушиваемый порт,

структура sin_addr должна содержать IP-адрес интерфейса, с которого будут приниматься входящие подключения,

sin_zero – заполнитель, выравнивающий размер структуры в соответствие с размером sockaddr. (Подробнее смотрите пояснения по структуре sockaddr и её использованию: http://msdn.microsoft.com/en-us/library/windows/desktop/ms740496%28v=vs.85%29.aspx).

Параметры sin_port и sin_addr должны задаваться с соблюдением сетевого порядка следования байтов (так называемый big-endian), при котором старший байт помещается в крайней позиции слева.

Для перевода числового значения порта (тип short int) используется функция htons (мнемоника «host to network short»). Существуют также функции для перевода в сетевой порядок других типов данных, например, htonl («host to network long»). Номером порта теоретически может быть любое число из диапазона значений short int, но для пользовательских приложений чаще всего используются числа из диапазона от 1025 до 5000.

Для задания IP-адреса используется вложенная структура in_addr. В случае функции bind в неё заносится адрес сетевого интерфейса, с которого должны приниматься входящие соединения. Для отладки на локальном компьютере таким адресом будет «127.0.0.1». В случае приёма с любого

интерфейса используется символьная константа INADDR_ANY. Для преобразования IP-адреса в сетевой порядок вызывается функция inet_addr.

Структура in_addr используемая для задания IP-адреса задаётся следующим образом:

```
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;
};
```

здесь S_un_b – адрес в виде четырех u_char;
S_un_w – адрес в виде двух u_short;
S_addr – адрес в виде одного u_long.

Пример использования:

```
     // объявляем переменную для хранения адреса
     sockaddr_in sa;
     // очищаем ее
     ZeroMemory(&sa, sizeof(s_addr));
     // задаем семейство адресов Internet
     sa.sin_family = AF_INET;
     /* задаем адрес узла, к которому выполняем подключение.
Не забываем преобразовать адрес из строкового десятичного
формата в бинарный. */
     sa.sin_addr.S_un.S_addr = inet_addr("192.168.1.1");
     // или sa.sin_addr.S_addr = htonl(INADDR_ANY);
     /*   задаем   порт   приложения-сервера.   Не   забываем
преобразовывать номер порта в сетевой порядок байт. */
     sa.sin_port = htons(101010);
```

Выполнив связывание, TCP-сервер переходит в режим ожидания подключений клиентов с помощью:

```
int listen(SOCKET s, int backlog);
```

s – сокет сервера;
backlog – максимальный размер очереди ожидающих подключений. Ограничивает количество одновременных соединений. При его превышении очередному клиенту будет отказано в подключении к серверу. В WinSock 2 не имеет ограничения сверху, в версии 1.1 оно ограничено значением константы SOMAXCON. Функция возвращает нуль в случае успешного выполнения или SOCKET_ERROR в противном случае.

После начала прослушивания серверного сокета приложение может начать извлекать запросы на соединение из очереди ожидающих подключения запросов. Это выполняется вызовом функции `accept`, которая автоматически создает новый сокет для входящего подключения, выполняет связывание и возвращает его дескриптор, а в структуру `sockaddr` заносит сведения о подключившемся клиенте (IP-адрес и порт):

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

здесь `s` – серверный «прослушиваемый» сокет (то есть сокет, для которого последовательно вызвали `socket`, `bind` и `listen`);

`addr` – адрес извлеченного из очереди ожидающих подключений сокета (структуры `sockaddr_in` для IPv4); указывает на адресную информацию подключенного к этому сокету клиентского узла;

`addrlen` – адрес переменной, в которую помещается длина структуры `sockaddr`. Перед вызовом эта переменная должна быть инициализирована значением `sizeof(struct sockaddr_in)`.

Функция возвращает дескриптор сокета для обмена данными с подключившимся клиентом или `INVALID_SOCKET`, если произошла ошибка. Если очередь входящих подключений пуста, то функция блокируется и не возвращает управление, пока с сервером не будет установлено соединение.

Для передачи данных по установленному соединению используется функция `send`:

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

где `s` – подключенный сокет;

`buf` – указывает на буфер с данными для передачи;

`len` – длина передаваемых данных;

`flags` – флаги, задающие способ, которым выполняется вызов; могут задавать уровень срочности отправляемых данных;

функция возвращает количество отправленных байтов или `SOCKET_ERROR` в случае ошибки.

Например:

```
if (send(s, (char *)&buff, 512,0) == SOCKET_ERROR)
{
…
error = WSAGetLastError();
…
}
```

Длина пакета данных ограничена протоколом. Функция не возвращает значение до тех пор, пока данные не будут отправлены.

Для приема данных от узла, с которым установлено соединение, используется функция `recv`:

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

здесь s – дескриптор подключенного сокета;
buf – буфер для входящих данных;
len – длина буфера в байтах;
flags – флаги, задающие способ, которым выполняется вызов;
функция возвращает число полученных байт, если соединение было закрыто – нуль, SOCKET_ERROR в случае ошибки. Если данных нет, то функция не возвращает управление, пока не придёт пакет (или пока не истечет тайм-аут).

Главное, что нужно помнить при работе с функциями send и recv это то, что они не сохраняют границы отправляемых или принимаемых буферов.

Пример использования:

```
int recv_len = 0;
recv_len = recv(s, (char *)&buff, MAX_PACKET_SIZE, 0);
if (recv_len == SOCKET_ERROR)
{
…
error = WSAGetLastError();
…
}
```

После завершения передачи данных соединение должно быть закрыто. Для этого используются функции shutdown и closesocket:

```
int shutdown(SOCKET s, int how);
```

s – закрываемый сокет;
how – способ закрытия (чаще всего SD_BOTH).
Функция не освобождает ресурсы, связанные с сокетом, но гарантирует завершение отправки и приёма данных до закрытия сокета.

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

s – закрываемый сокет.
Функция закрывает сокет и освобождает связанные с ним ресурсы. Сервер закрывает сокеты, связывающие его с клиентами по завершению обмена информации. Серверный сокет, который прослушивается функцией listen, закрывается только тогда, когда завершается работа серверного приложения.
Оба вызова возвращают нуль в случае успеха или SOCKET_ERROR, если произошла ошибка.
Существуют более сложные способы закрытия соединения (выборочное закрытие соединения на одной из сторон, оставляя другую сторону в рабочем состоянии). Для этого используется функция shutdown с нужным значением флага how (SD_RECEIVE закрывает канал «сервер-клиент», SD_SEND закрывает канал «клиент-сервер», в то время как SD_BOTH – оба). Это «мягкое» закрытие соединения – удаленному узлу посылается

уведомление о желании разорвать соединение, но соединение не разрывается, пока клиент не возвратит свое подтверждение.

Вызов `shutdown` не освобождает от последующего вызова `closesocket`.

Клиент

Клиентское приложение после создания сокета может его использовать для обмена данными с другими узлами. Для этого ему нужно установить соединение с другим узлом, задав его IP-адрес и порт, который на серверной стороне прослушивает приложение-сервер:

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

`s` – неподключенный сокет;

`name` – задает сокет, к которому выполняется подключение. Структура `sockaddr` зависит от используемого семейства протоколов. Для IPv4 используется `sockaddr_in` (см. выше);

`namelen` – размер параметра name (структуры) в байтах;

Функция возвращает нуль, если соединение создано и `SOCKET_ERROR` в противном случае. Пояснения и пример применения доступны на: http://msdn.microsoft.com/en-us/library/windows/desktop/ms737625%28v=vs.85%29.aspx

Пример использования:

```
// объявляем переменную для хранения адреса
sockaddr_in s_addr;
// очищаем ее
ZeroMemory(&s_addr, sizeof(s_addr));
// задаем семейство адресов Internet
s_addr.sin_family = AF_INET;
/* задаем адрес узла, к которому выполняем подключение. Не забываем преобразовать адрес из строкового десятичного формата в бинарный. */
s_addr.sin_addr.S_un.S_addr = inet_addr("192.168.1.1");
/* задаем порт приложения-сервера. Не забываем преобразовывать номер порта в сетевой порядок байт. */
s_addr.sin_port = htons(2014);
…
// открываем соединение
int con_status;
con_status = connect(s, (sockaddr *)&s_addr, sizeof(s_addr));
if (con_status == SOCKET_ERROR)
{
…
```

```
error = WSAGetLastError();
…
}
```

После установки соединения можно начинать обмен данными при помощи функций `send`/`recv`.

Задание. На основе API Windows Sockets 2 разработайте клиент-серверное сетевое приложение для игры в «крестики-нолики» с выделенным сервером. Сервер должен реализовать создание партии (ожидать подключения двух игроков), после чего контролировать правильность ходов игроков и поддерживать у них информацию о состоянии игрового поля. За очередностью ходов также следит сервер. Клиент должен иметь возможность подключиться к указанному серверу, передать ход игрока на сервер и получить от сервера состояние игрового поля. По окончании игры должен быть предусмотрен вывод результатов обоим игрокам.

Клиентская программа может быть реализована как приложение с графическим интерфейсом или как консольное приложение с псевдографическим интерфейсом, например, такого вида:

```
X O _
X O _
_ _ _
```

***Дополнительно**: попытайтесь предусмотреть возможность проведения нескольких партий одновременно – многопоточность сервера.*