Крис Касперски

САМОУЧИТЕЛЬ ИГРЫ НА WINSOCK

kk@sendmail.ru

Крис Касперски
kk@sendmail.ru

Самоучитель игры на WINSOCK

Введение

Сокеты (*sockets*) представляют собой высокоуровневый унифицированный интерфейс взаимодействия с телекоммуникационными протоколами. В технической литературе встречаются различные переводы этого слова – их называют и гнездами, и соединителями, и патронами, и патрубками, и т. д. По причине отсутствия устоявшегося русскоязычного термина, в настоящей статье сокеты будет именоваться сокетами и никак иначе.

Программирование сокетов не сложно само по себе, но довольно поверхностно описано в доступной литературе, а Windows Sockets SDK содержит многоженство ошибок как в технической документации, так и в прилагаемых к ней демонстрационных примерах. К тому же имеются значительные отличия реализаций сокетов в UNIX и в Windows, что создает очевидные проблемы.

Автор постарался дать максимально целостное и связанное описание, затрагивающее не только основные моменты, но и некоторые тонкости не известные рядовым программистам. Ограниченный объем журнальной статьи не позволяет рассказать обо всем, поэтому, пришлось сосредоточиться только на одной реализации сокетов – библиотеке Winsock 2, одном языке программирования – Си\Си++ (хотя сказанное большей частью приемлемо к Delphi, Perl и т. д.) и одном виде сокетов – блокируемых синхронных сокетах.

ALMA MATER

Основное подспорье в изучении сокетов *Windows Sockets 2 SDK*. SDK – это документация, набор заголовочных файлов и инструментарий разработчика. Документация не то, чтобы очень хороша – но все же написана достаточна грамотно и позволяет, пускай, не без труда, освоить сокеты даже без помощи какой-либо другой литературы. Причем, большинство книг, имеющиеся на рынке, явно уступают Microsoft в полноте и продуманности описания. Единственный недостаток SDK – он полностью на английском (для некоторых это очень существенно).

Из инструментария, входящего в SDK, в первую очередь хотелось бы выделить утилиту sockeye.exe – это настоящий "тестовый стенд" разработчика. Она позволяет в интерактивном режиме вызывать различные сокет-функции и манипулировать ими по своему усмотрению.

Демонстрационные программы, к сожалению, не лишены ошибок, причем порой довольно грубых и наводящих на мысли – а тестировались ли эти примеры вообще? (Например, в исходном тексте программы simples.c в вызове функций send и sendto вместо strlen стоит sizeof) В то же время, все примеры содержат множество подробных комментариев и раскрывают довольно любопытные приемы нетрадиционного программирования, поэтому ознакомится с ними все-таки стоит.

Из WEB-ресуросов, посвященных программированию сокетов, и всему, что с ними связано, в первую очередь хотелось бы отметить следующие три: sockaddr.com; www.winsock.com и www.sockets.com.

Обзор сокетов

Библиотека Winsock поддерживает два вида сокетов – ***синхронные*** (*блокируемые*) и ***асинхронные*** (*неблокируемые*). Синхронные сокеты задерживают управление на время выполнения операции, а асинхронные возвращают его немедленно, продолжая выполнение в фоновом режиме, и, закончив работу, уведомляют об этом вызывающий код.

ОС Windows 3.x поддерживает только асинхронные сокеты, поскольку, в среде с корпоративной многозадачностью захват управления одной задачей "подвешивает" все остальные, включая и саму систему. ОС Windows 9x\NT поддерживают оба вида сокетов, однако, в силу того, что синхронные сокеты программируются более просто, чем асинхронные, последние не получили большого распространения. Эта статья посвящена исключительно синхронным сокетам (асинхронные – тема отдельного разговора).

Сокеты позволяют работать со множеством протоколов и являются удобным средством межпроцессорного взаимодействия, но в данной статье речь будет идти только о сокетах семейства протоколов TCP/IP, использующихся для обмена данными между узлами сети Интернет. Все остальные протоколы, такие как IPX/SPX, NetBIOS по причине ограниченности объема журнальной статьи рассматриваться не будут.

Независимо от вида, сокеты делятся на два типа – ***потоковые*** и ***дейтаграммные***. Потоковые сокеты работают с установкой соединения, обеспечивая надежную идентификацию обоих сторон и гарантируют целостность и успешность доставки данных. Дейтаграмные сокеты работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных, зато они заметно быстрее потоковых.

Выбор того или иного типа сокетов определяется транспортным протоколом, на котором работает сервер, – клиент не может по своему желанию установить с дейтаграммным сервером потоковое соединение.

> ***Замечание**: дейтаграммные сокеты опираются на протокол UDP, а потоковые на TCP.*

Первый шаг, второй, третий...

Для работы с библиотекой Winsock 2.x в исходный тест программы необходимо включить директиву "*#include <winsock2.h>*", а в командной строке линкера указать "ws2_32.lib". В Microsoft Visual Studio для этого достаточно нажать <Alt-F7>, перейти к закладке "Link" и к списку библиотек, перечисленных в строке "Object/Library modules", добавить "ws2_32.lib", отделив ее от остальных символом пробела.

Перед началом использования функций библиотеки Winsock ее необходимо подготовить к работе вызовом функции "*int **WSAStartup** (WORD wVersionRequested, LPWSADATA lpWSAData)",* передав в старшем байта слова *wVersionRequested* номер требуемой версии, а в младшем – номер подверсии.

Аргумент *lpWSAData* должен указывать на структуру *WSADATA*, в которую при успешной инициализации будет занесена информация о производителе библиотеки. Никакого особенного интереса она не представляет и прикладное приложение может ее игнорировать. Если инициализация проваливается, функция возвращает ненулевое значение.

Первый шаг – создание объекта "сокет". Это осуществляется функцией "*SOCKET **socket** (int af, int type, int protocol)*" Первый слева аргумент указывает на семейство используемых протоколов. Для Интернет – приложений он должен иметь значение AF_INET.

Следующий аргумент задает тип создаваемого сокета – *потоковый* (*SOCK_STREAM*) или *дейтаграммный* (*SOCK_DGRAM*) (еще существуют и сырые сокеты, но они не поддерживаются Windows – см раздел "Сырые сокеты").

Последний аргумент уточняет какой транспортный протокол следует использовать. Нулевое значение соответствует выбору по умолчанию: TCP – для потоковых сокетов и UDP для дейтаграммных. В большинстве случаев нет никакого смысла задавать протокол вручную и обычно полагаются на автоматический выбор по умолчанию.

Если функция завершилась успешно она возвращает дескриптор сокета, в противном случае INVALID_SOCKET.

> ***Примечание:*** *дальнейшие шаги зависят от того, является ли приложение сервером или клиентом. Ниже эти два случая будут описаны раздельно.*

Клиент: шаг второй – для установки соединения с удаленным узлом потоковый сокет должен вызвать функцию *"int **connect** (SOCKET s, const struct sockaddr FAR* name, int namelen)"*. Датаграмные сокеты работают без установки соединения, поэтому, *обычно* не обращаются к функции connect.

> ***Примечание***: *за словом "обычно" стоит один хитрый примем программирования – вызов connect позволяет дейтаграмному сокету обмениваться данными с узлом не только функциями sendto, recvfrom, но и более удобными и компактными send и recv. Эта тонкость описана в Winsocket SDK и широко используется как самой Microsoft, так и сторонними разработчикам. Поэтому, ее использование вполне безопасно.*

Первый слева аргумент – дескриптор сокета, возращенный функцией socket; второй – указатель на структуру "*sockaddr*", содержащую в себе адрес и порт удаленного узла с которым устанавливается соединение. Структура sockaddr используется множеством функций, поэтому ее описание вынесено в отдельный раздел "Адрес раз, адрес два...". Последний аргумент сообщает функции размер структуры sockaddr.

После вызова connect система предпринимает попытку установить соединение с указанным узлом. Если по каким-то причинам это сделать не удастся (адрес задан неправильно, узел не существует или "висит", компьютер находится не в сети), функция возвратит ненулевое значение.

Сервер: шаг третий – прежде, чем сервер сможет использовать сокет, он должен связать его с локальным адресом. Локальный, как, впрочем, и любой другой адрес Интернета, состоит из IP-адреса узла и номера порта. Если сервер имеет несколько IP-адресов, то сокет может быть связан как со вмести ними сразу (для этого вместо IP-адреса следует указать константу INADDR_ANY равную нулю), так и с каким-то конкретным одним.

Связывание осуществляется вызовом функции "*int **bind** (SOCKET s, const struct sockaddr FAR* name, int namelen)"* Первым слева аргументом передается дескриптор сокета, возращенный функций socket, за ним следуют указатель на структуру sockaddr и ее длина (см. раздел "Адрес раз, адрес два...").

Строго говоря, клиент также должен связывать сокет с локальным адресом перед его использованием, однако, за него это делает функция connect, ассоциируя сокет с одним из портов, наугад выбранных из диапазона 1024-5000. Сервер же должен "садиться" на заранее определенный порт, например, 21 для FTP, 23 для telnet, 25 для SMTP, 80 для WEB, 110 для POP3 и т.д. Поэтому ему приходится осуществлять связывание "вручную".

При успешном выполнении функция возвращает нулевое значение и ненулевое в противном случае.

Сервер: шаг четвертый – выполнив связывание, потоковый сервер переходит в режим ожидания подключений, вызывая функцию "*int **listen** (SOCKET s, int backlog)*", где *s* – дескриптор сокета, а *backlog* – максимально допустимый размер очереди сообщений.

Размер очереди ограничивает количество одновременно обрабатываемых соединений, поэтому, к его выбору следует подходить "с умом". Если очередь полностью заполнена, очередной клиент при попытке установить соединение получит отказ (TCP пакет с установленным флагом RST). В то же время максимально разумное количество подключений определяются производительностью сервера, объемом оперативной памяти и т. д.

Датаграммные серверы не вызывают функцию listen, т. к. работают без установки соединения и сразу же после выполнения связывания могут вызывать recvfrom для чтения входящих сообщений, минуя четвертый и пятый шаги.

Сервер: шаг пятый – извлечение запросов на соединение из очереди осуществляется функцией "*SOCKET **accept** (SOCKET s, struct sockaddr FAR* addr, int FAR* addrlen)*", которая автоматически создает новый сокет, выполняет связывание и возвращает его дескриптор, а в структуру *sockaddr* заносит сведения о подключившемся

клиенте (IP-адрес и порт). Если в момент вызова accept очередь пуста, функция не возвращает управление до тех пор, пока с сервером не будет установлено хотя бы одно соединение. В случае возникновения ошибки функция возвращает отрицательное значение.

Для параллельной работы с несколькими клиентами следует сразу же после извлечения запроса из очереди порождать новый поток (процесс), передавая ему дескриптор созданного функцией accept сокета, затем вновь извлекать из очереди очередной запрос и т.д. В противном случае, пока не завершит работу один клиент, север не сможет обслуживать всех остальных.

Все вместе – после того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными, вызывая функции "*int **send** (SOCKET s, const char FAR * buf, int len,int flags)*" и "*int **recv** (SOCKET s, char FAR* buf, int len, int flags)*" для посылки и приема данных соответственно.

Функция *send* возвращает управление сразу же после ее выполнения независимо от того, получила ли принимающая сторона наши данные или нет. При успешном завершении функция возвращает количество *передаваемых* (*не переданных!*) данных – т. е. успешное завершение еще не свидетельствует от успешной доставке! В общем-то, протокол TCP (на который опираются потоковые сокеты) гарантирует успешную доставку данных получателю, но лишь при условии, что соединение не будет преждевременно разорвано. Если связь прервется до окончания пересылки, данные останутся не переданными, но вызывающий код не получит об этом никакого уведомления! А ошибка возвращается лишь в том случае, если соединение разорвано ***до*** вызова функции send!

Функция же *recv* возвращает управление только после того, как получит хотя бы один байт. Точнее говоря, она ожидает прихода целой *дейтаграммы*. Дейтаграмма – это совокупность одного или нескольких IP пакетов, посланных вызовом send. Упрощенно говоря, каждый вызов recv за один раз получает столько байтов, сколько их было послано функцией send. При этом подразумевается, что функции recv предоставлен буфер достаточных размеров, в противном случае ее придется вызвать несколько раз. Однако, при всех последующих обращениях данные будет браться из локального буфера, а не приниматься из сети, т. к. TCP-провайдер не может получить "кусочек" дейтаграммы, а только ею всю целиком.

Работой обоих функций можно управлять с помощью *флагов*, передаваемых в одной переменной типа int третьим слева аргументом. Эта переменная может принимать одно из двух значений: *MSG_PEEK* и *MSG_OOB*.

Флаг MSG_PEEK заставляет функцию recv просматривать данные вместо их чтения. Просмотр, в отличие от чтения, не уничтожает просматриваемые данные. Некоторые источники утверждают, что при взведенном флаге MSG_PEEK функция recv не задерживает управления если в локальном буфере нет данных, доступных для немедленного получения. Это неверно! Аналогично, иногда приходится встречать откровенно ложное утверждение, якобы функция send со взведенным флагом MSG_PEEK возвращает количество уже переданных байт (вызов send не блокирует управления). На самом деле функция send игнорирует этот флаг!

Флаг MSG_OOB предназначен для передачи и приема *срочных* (*Out Of Band*) данных. Срочные данные не имеют преимущества перед другими при пересылке по сети, а всего лишь позволяют оторвать клиента от нормальной обработки потока обычных данных и сообщить ему "срочную" информацию. Если данные передавались функцией send с установленным флагом MSG_OOB, для их чтения флаг MSG_OOB функции recv так же должен быть установлен.

> ***Замечание**: настоятельно рекомендуется воздержаться от использования срочных данных в своих приложениях. Во-первых, они совершенно необязательны – гораздо проще, надежнее и элегантнее вместо них создать отдельное TCP-соединение. Во-вторых, по поводу их реализации нет единого мнения и интерпретации различных производителей очень сильно отличаются друг от друга. Так, разработчики до сих пор не пришли к окончательному соглашению по поводу того, куда должен указывать указатель срочности: или на последний байт срочных данных или на байт, следующий за последним байтом срочных данных. В результате, отправитель никогда не имеет уверенности, что получатель сможет правильно интерпретировать его запрос.*

Еще существует флаг MSG_DONTROUTE, предписывающий передавать данные без маршрутизации, но он не поддерживаться Winsock и, поэтому, здесь не рассматривается.

Дейтаграммный сокет так же может пользоваться функциями send и recv, если предварительно вызовет connect (см. "Клиент: шаг третий"), но у него есть и свои, "персональные", функции: "*int **sendto** (SOCKET s, const char FAR * buf, int len,int flags, const struct sockaddr FAR * to, int tolen)*" и "*int **recvfrom** (SOCKET s, char FAR* buf, int len, int flags, struct sockaddr FAR* from, int FAR* fromlen)*".

Они очень похожи на send и recv – разница лишь в том, что sendto и recvfrom требуют явного указания адреса узла принимаемого или передаваемого данные. Вызов recvfrom не требует предварительного задания адреса передающего узла – функция принимает все пакеты, приходящие на заданный UDP-порт со всех IP-адресов и портов. Напротив, отвечать отправителю следует на тот же самый порт откуда пришло сообщение. Поскольку, функция recvfrom заносит IP-адрес и номер порта клиента после получения от него сообщения, программисту фактически ничего не нужно делать – только передать sendto тот же самый указатель на структуру sockaddr, который был ранее передан функции recvfrem, получившей сообщение от клиента.

> ***Еще одна деталь** – транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной доставки сообщений и эта задача ложиться на плечи самого разработчика. Решить ее можно, например, посылкой клиентом подтверждения об успешности получения данных. Правда, клиент тоже не может быть уверен, что подтверждение дойдет до сервера, а не потеряется где-нибудь в дороге. Подтверждать же получение подтверждения – бессмысленно, т.к. это рекурсивно неразрешимо. Лучше вообще не использовать дейтаграммные сокеты на ненадежных каналах.*

Во всем остальном обе пары функций полностью идентичны и работают с теми самыми флагами – MSG_PEEK и MSG_OOB.

Все четыре функции при возникновении ошибки возвращают значение SOCKET_ERROR (== -1).

Примечание: *в UNIX с сокетами можно обращаться точно так, как с обычными файлами, в частности писать и читать в них функциями write и read. ОС Windows 3.1 не поддерживала такой возможности, поэтому, при переносе приложений их UNIX в Windows все вызовы write и read должны были быть заменены на send и recv соответственно. В Windows 95 с установленным Windows 2.x это упущение исправлено, теперь дескрипторы сокетов можно передавать функциям ReadFil, WriteFile, DuplicateHandle и др.*

Шаг последний – для закрытия соединения и уничтожения сокета предназначена функция "*int* ***closesocket*** *(SOCKET s)*", которая в случае удачного завершения операции возвращает нулевое значение.

Перед выходом из программы, необходимо вызвать функцию "*int* ***WSACleanup*** *(void)*" для деинициализации библиотеки WINSOCK и освобождения используемых этим приложением ресурсов.

Внимание*: завершение процесса функцией ExitProcess автоматически не освобождает ресурсы сокетов!*

Примечание: более сложные приемы закрытия соединения – *протокол TCP позволяет выборочно закрывать соединение любой из сторон, оставляя другую сторону активной. Например, клиент может сообщить серверу, что не будет больше передавать ему никаких данных и закрывает соединение "клиент → сервер", однако, готов продолжать принимать от него данные, до тех пор, пока сервер будет их посылать, т. е. хочет оставить соединение "сервер → клиент" открытым.*

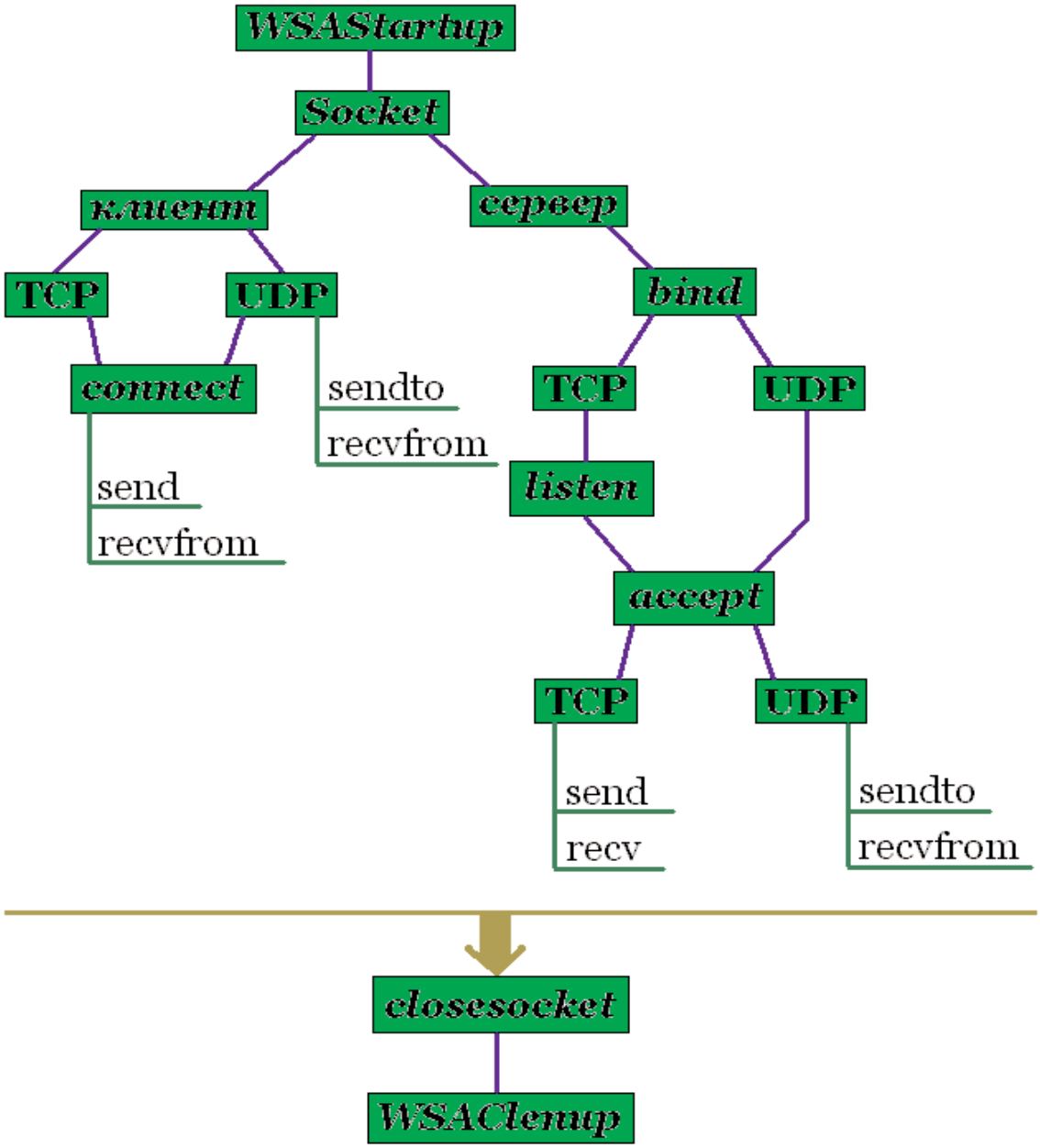
Для этого необходимо вызвать функцию "int ***shutdown*** *(SOCKET s ,int how)", передав в аргументе how одно из следующих значений: SD_RECEIVE для закрытия канала "сервер → клиент", SD_SEND для закрытия канала "клиент → север", и, наконец, SD_BOTH для закрытия обоих каналов.*

Последний вариант выгодно отличается от closesocket "мягким" закрытием соединения – удаленному узлу будет послано уведомление о желании разорвать связь, но это желание не будет воплощено в действительность, пока тот узел не возвратит свое подтверждение. Таким образом, можно не волноваться, что соединение будет закрыто в самый неподходящий момент.

Внимание*: вызов shutdown не освобождает от необходимости закрытия сокета функцией closesocket!*

Дерево вызовов

Для большей наглядности демонстрации взаимосвязи socket-функций друг с другом, ниже приведено дерево вызовов, показывающее в каком порядке должны следовать вызовы функций в зависимости от типа сокетов (потоковый или дейтаграммный) и рода обработки запросв (клиент или сервер).



Адрес раз, адрес два...

С адресами как раз и наблюдается наибольшая путаница, в которую не помешает внести немного ясности. Прежде всего структура sockaddr определенная так:

```
struct sockaddr
{
        u_short sa_family;            // семейство протоколов (как правило AF_INET)
        char    sa_data[14];          // IP-адрес узла и порт
};
```

Однако, теперь уже считается устаревшей, и в Winsock 2.x на смену ей пришла структура sockaddr_in, определенная следующим образом:

```
struct sockaddr_in
{
        short   sin_family;           // семейство протоколов (как правило AF_INET)
        u_short sin_port;             // порт
        struct  in_addr sin_addr;     // IP-адрес
        char    sin_zero[8];          // хвост
};
```

В общем-то ничего не изменилось (и стоило огород городить?), замена беззнакового короткого целого на знаковое короткое целое для представления семейства протоколов ничего не дает. Зато теперь адрес узла представлен в виде трех полей – sin_port (номера порта), sin_addr (IP-адреса узла) и "хвоста" из восьми нулевых байт, который остался от четырнадцати символьного массива sa_data. Для чего он нужен? Дело в том, что структура sockaddr не привязана именно к Интернет и может работать и с другими сетями. Адреса же некоторых сетей требуют для своего представления гораздо больше четырех байт – вот и приходится брать с запасом!

Структура in_addr определяется следующим в образом:

```
struct in_addr {
        union {
                struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;      // IP-адрес
                struct { u_short s_w1,s_w2; } S_un_w;               // IP-адрес
                u_long S_addr;                                      // IP-алрес
        } S_un;
}
```

Как видно, она состоит из одного IP-адреса, записанного в трех "ипостасях" – четырехбайтовой последовательности (S_un_b), пары двухбайтовых слов (S_un_W) и одного длинного целого (S_addr) – выбирай на вкус... Но не все так просто! Во многих программах, технических руководствах и даже демонстрационных примерах, прилагающихся к Winsock SDK, встречается обращение к "таинственному" члену структуры s_addr, который явно не описан в SDK! Например, вот строка из файла "Simples.h": "local.sin_addr.**s_addr** = (!interface)?INADDR_ANY:inet_addr(interface);"

Это что такое?! Заглянув в файл "winsock2.h" можно обнаружить следующее: "#define s_addr S_un.S_addr". Ага, да ведь это эквивалент s_addr, т. е. IP-адресу, записанному в виде длинного целого!

На практике можно с одинаковым успехом пользоваться как "устаревшей" sockaddr, так и "новомодной" sockaddr_in. Однако, поскольку, прототипы остальных функций не изменились, при использовании sockaddr_in придется постоянно выполнять явные преобразования, например так: *"**sockaddr_in** dest_addr; connect (mysocket, (**struct sockaddr***) &dest_addr, sizeof(dest_addr)"*.

Для преобразования IP-адреса, записанного в виде символьной последовательности наподобие "127.0.0.1" в четырехбайтовую числовую последовательность предназначена функция "*unsigned long **inet_addr** (const char FAR * cp)*". Она принимает указатель на символьную строку и в случае успешной операции преобразует ее в четырехбайтовый IP-адрес или –1 если это невозможно. Возвращенный функцией результат можно присвоить элементу структуры sockaddr_in следующим образом: "*struct sockaddr_in dest_addr; dest_addr.sin_addr.S_addr=inet_addr("195.161.42.222");*". При использовании структуры sockaddr это будет выглядеть так: "*struc sockaddr dest_addr; ((unsigned int *)(&dest_addr.sa_data[0]+2))[0] = inet_addr("195.161.42.222");*"

Попытка передать inet_addr доменное имя узла приводит к провалу. Узнать IP-адрес такого-то домена можно с помощью функции "*struct hostent FAR * **gethostbyname** (const char FAR * name);*". Функция обращается к DNS и возвращает свой ответ в структуре hostent или нуль если DNS сервер не смог определить IP-адрес данного домена.

Структура hostent выглядит следующим образом:

```
struct hostent
{
        char FAR *       h_name;      // официальное имя узла
        char FAR * FAR * h_aliases;   // альтернативные имена узла (массив строк)
        short            h_addrtype;  // тип адреса
        short            h_length;    // длина адреса (как правило AF_INET)
        char FAR * FAR * h_addr_list; // список указателей на IP-адреса
                                      // ноль - конец списка
};
```

Как и в случае с in_addr, во множестве программ и прилагаемых к Winsock SDK примерах активно используется недокументированное поле структуры h_addr. Например, вот строка из файла *"simplec.c" "memcpy(&(server.sin_addr),hp->**h_addr**,hp->h_length);"* Заглянув в "winsock2.h" можно найти, что оно обозначает: "*#define h_addr h_addr_list[0]*".

А вот это уже интересно! Дело в том, что с некоторыми доменными именами связано сразу несколько IP-адресов. В случае неработоспособности одного узла, клиент может попробовать подключится к другому или просто выбрать узел с наибольшей скоростью обмена. Но в приведенном примере клиент использует только первый IP-адрес в списке и игнорирует все остальные! Конечно, это не смертельно, но все же будет лучше, если в своих программах вы будете учитывать возможность подключения к остальным IP-адресам, при невозможности установить соединение с первым.

Функция gethostbyname ожидает на входе ***только*** доменные имена, но не цифровые IP-адреса. Между тем, правила "хорошего тона" требуют предоставления клиенту возможности как задания доменных имен, так и цифровых IP-адресов.

Решение заключается в следующем – необходимо проанализировать переданную клиентом строку: если это IP адрес, то передать его функции inet_addr в противном случае – gethostbyaddr, полагая, что это доменное имя. Для отличия IP-адресов от доменных имен многие программисты используют нехитрый трюк: если первый символ строки – цифра, это IP-адрес, иначе – имя домена. Однако, такой трюк не совсем честен – доменные имя могут начинаться с цифры, например, "666.ru", могут они и заканчиваться цифрой, например, к узлу "666.ru" члены субдомена "666" могут так и обращаться – "666". Самое смешное, что (теоретически) могут существовать имена доменов, синтаксически неотличимые от IP-адресов! Поэтому, на взгляд автора данной статьи, лучше всего действовать так: передаем введенную пользователем строку функции inet_addr, если она возвращает ошибку, то вызываем gethostbyaddr.

Для решения обратной задачи – определении доменного имени по IP адресу предусмотрена функция "*struct HOSTENT FAR * **gethostbyaddr** (const char FAR * addr, int len, int type)*", которая во всем аналогична gethostbyname, за тем исключением, что ее аргументом является не указатель на строку, содержащую имя, а указатель на четырехбайтовый IP-адрес. Еще два аргумента задают его длину и тип (соответственно, 4 и AF_INET).

Определение имени узла по его адресу бывает полезным для серверов, желающих "в лицо" знать своих клиентов.

Для преобразования IP-адреса, записанного в сетевом формате в символьную строку, предусмотрена функция "*char FAR * **inet_ntoa** (struct in_addr)*", которая принимает на вход структуру in_addr, а возвращает указатель на строку, если преобразование выполнено успешно и ноль в противном случае.

Сетевой порядок байт

Среди производителей процессоров нет единого мнения на счет порядка следования младших и старших байт. Так например, у микропроцессоров Intel младшие байты располагаются по меньшим адресам, а у микропроцессоров Motorola 68000 – наоборот. Естественно, это вызывает проблемы при межсетевом взаимодействии, поэтому, был введен специальный *сетевой порядок байт*, предписывающий старший байт передавать первым (все не так, как у Intel).

Для преобразований чисел из сетевого формата в формат локального хоста и наоборот предусмотрено четыре функции - первые две манипулируют короткими целыми (16-битными словами), а две последние – длинными (32-битными двойными словами): *u_short **ntohs** (u_short netshort); u_short **htons** (u_short hostshort); u_long **ntohl** (u_long netlong); u_long **htonl** (u_long hostlong);*

Чтобы в них не запутаться, достаточно запомнить, что за буквой "n" скрывается сокращение "**n**etwork", за "h" – "**h**ost" (подразумевается локальный), "s" и "l" соответственно короткое (**s**hort) и длинное (**l**ong) беззнаковые целые, а "to" и обозначает преобразование. Например, "htons" расшифровывается так: "**H**ost **N**etwork (**s**hort)" т. е. преобразовать короткое целое из формата локального хоста в сетевой формат.

***Внимание**: все значения, возвращенные socket-функциями уже находятся в сетевом формате и "вручную" их преобразовывать **нельзя**! Т. к. это преобразование исказит результат и приведен к неработоспособности.*

Чаще всего к вызовам этих функций прибегают для преобразования номера порта согласно сетевому порядку. Например: *dest_addr.sin_port = htons(110)*.

Дополнительные возможности

Для "тонкой" настойки сокетов предусмотрена функция "*int **setsockopt** (SOCKET s, int level, int optname, const char FAR * optval, int optlen)*". Первый слева аргумент – дескриптор сокета, который собираются настраивать, *level – уровень* настойки. С каждым уровнем связан свой набор опций. Всего определено два уровня – SOL_SOCKET и IPPROTO_TCP. В ограниченном объеме журнальной статьи перечислить все опции невозможно, поэтому, ниже будет рассказано только о самых интересных из них, а сведения обо всех остальных можно почерпнуть из Winsock SDK.

Третий слева аргумент представляет собой указатель на переменную, содержащую значение опции. Ее размер варьируется в зависимости от рода опции и передается через четвертый слева аргумент.

Уровень SOL_SOCKET:

SO_RCVBUF (int) – задает размер входного буфера для приема данных. К TCP-окну никакого отношения не имеет, поэтому, может безболезненно варьироваться в широких пределах.

SO_SNDBUF (int) – задает размер входного буфера для передачи данных. Увеличение размера буферов на медленных каналах приводит к задержкам и снижает производительность.

Уровень IPPROTO_TCP

TCP_NODELAY (BOOL) – выключает *Алгоритм Нагла*. Алгоритм Нагла был разработан специально для прозрачного кэширования крохотных пакетов (*тинограмм*). Когда один узел посылает другому несколько байт, к ним дописываются заголовки TCP и IP, которые в совокупности обычно занимают более 50 байт. Таким образом, при побайтовом обмене между узлами свыше 98% передаваемой по сети информации будет приходиться на служебные данные!

Алгоритм Нагла состоит в следующем: отправляем первый пакет и, до тех пор, пока получатель не возвратит TCP-уведомление успешности доставки, не передаем в сеть никаких пакетов, а накапливаем их на локальном узле, собирая в один большой пакет. Такая техника совершенно прозрачна для прикладных приложений, и в то же время позволяет значительно оптимизировать трафик, но в некоторых (достаточно экзотических) случаях, когда требуется действительно побайтовый обмен, Алгоритм Нагла приходится отключать (по умолчанию он включен).

Для получения текущих значений опций сокета предусмотрена функция "*int getsockopt (SOCKET s, int level, int optname, char FAR* optval, int FAR* optlen)*" которая полностью аналогична предыдущей за исключением того, что не устанавливает опции, а возвращает их значения.

Сырые сокеты

Помимо потоковых и дейтаграммных сокетов существуют, так называемые, сырые (RAW) сокеты. Они предоставляют возможность "ручного" формирования TCP\IP-пакетов, равно как и полного доступа к содержимому заголовков полученных TCP\IP-пакетов, что необходимо многим сетевым сканерам, FireWall-ам, брандмаузерам и, разумеется, атакующим программам, например, устанавливающим в поле "адрес отправителя" адрес самого получателя.

Спецификация Winsock 1.x категорически не поддерживала сырых сокетов. В Winsock 2.x положение как будто было исправлено: по крайней мере формально такая поддержка появилась, и в SDK даже входил пример, демонстрирующий использование сырых сокетов для реализации утилиты ping. Однако, попытки использования сырых сокетов для всех остальных целей с треском проваливались – система упрямо игнорировала сформированный "вручную" IP- (или TCP-) пакет и создавала его самостоятельно.

Документация объясняла, что для самостоятельной обработки заголовков пакетов, опция IP_HDRINCL должна быть установлена. Весь фокус в том, что вызов "setsockopt(my_sock,IPPROTO_IP, IP_HDRINCL, &oki, sizeof(oki))" возвращал ошибку!

Таким образом, на прикладном уровне получить непосредственный доступ к заголовкам TCP/IP невозможно. Это препятствует переносу многих приложений из UNIX в Windows, более того, определенным образом ущемляет возможности самой Windows, не позволяя ей решать целый ряд задач, требующих поддержки сырых сокетов.

Законченные реализации

Ниже приведено четыре подробно комментированных исходных текста, реализующих простых TCP и UDP эхо-сервера и TCP- и UDP-клиентов. (Эхо-сервер просто возвращает клиенту, полученные от него данные).

Для их компиляции с помощью Microsoft Visual C++ достаточно отдать команду: "cl.exe имя_файла.cpp ws2_32.lib".

Проверка работоспособности TCP-сервера: запустите TCP-сервер и наберите в командной строке Windows "telnet.exe 127.0.0.1 666", где 127.0.0.1 обозначает локальный адрес вашего узла (это специально зарезервированный для этой цели адрес и он выглядит одинаково для всех узлов), а 666 – номер порта на который

"сел" сервер. Если все работает успешно, то telnet установит соединение и на экране появится приветствие "Hello, Sailor!". Теперь можно набирать на клавиатуре некоторый текст и получать его назад от сервера.

Проверка работоспособности TCP-клиента: запустите TCP-сервер и затем одну или несколько копий клиента. В каждом из них можно набирать некоторый текст на клавиатуре и после нажатия на Enter получать его обратно от сервера.

Проверка работоспособности UDP-сервера и клиента: запустите UDP-сервер и одну или несколько копий клиента – в каждой из них можно набирать на клавиатуре некоторые данные и получать их обратно от сервера.

> ***Внимание**: работая с серверными приложениями, вы (если не предпримите дополнительных мер) предоставляете возможность воспользоваться ими каждому абоненту Интернет (если в момент работы сервера вы подключены к Интернет). Проблема в том, что ошибки реализации, в особенности переполняющиеся буфера, могут позволить удаленному злоумышленнику выполнить на вашей машине **любой** код, со всеми вытекающими от сюда последствиями.*
>
> *Будьте очень внимательны, а еще лучше, не входите в Интернет, пока не будете полностью уверенны, что сервера отлажены и не содержат ошибок!*

Пример реализации TCP-эхо-сервера

```
// Пример простого TCP-эхо-сервера

#include <stdio.h>
#include <winsock2.h>                    // Wincosk2.h должен быть раньше windows!
#include <windows.h>

#define MY_PORT         666              // Порт, который слушает сервер

// макрос для печати количества активных пользователей
#define PRINTNUSERS if (nclients) printf("%d user on-line\n",nclients);else printf("No User on line\n");

// прототип функции, обслуживающий подключившихся пользователей
DWORD WINAPI SexToClient(LPVOID client_socket);

// глобальная переменная - количество активных пользователей
int nclients = 0;

int main(int argc, char* argv[])
{
         char buff[1024];                // Буфер для различных нужд

         printf("TCP SERVER DEMO\n");

         // Шаг 1 - Инициализация Библиотеки Сокетов
         // Т. к. возвращенная функцией информация не используется
         // ей передается указатель на рабочий буфер, преобразуемый к указателю
         // на структуру WSADATA.
         // Такой прием позволяет сэкономить одну переменную, однако, буфер
         // должен быть не менее полкилобайта размером (структура WSADATA
         // занимает 400 байт)
```

```
if (WSAStartup(0x0202,(WSADATA *) &buff[0]))
{
        // Ошибка!
        printf("Error WSAStartup %d\n",WSAGetLastError());
        return -1;
}

// Шаг 2 - создание сокета
SOCKET mysocket;
// AF_INET            - сокет Интернета
// SOCK_STREAM        - потоковый сокет (с установкой соединения)
// 0                  - по умолчанию выбирается TCP протокол
if ((mysocket=socket(AF_INET,SOCK_STREAM,0))<0)
{
               // Ошибка!
               printf("Error socket %d\n",WSAGetLastError());
               WSACleanup();          // Деинициализация библиотеки Winsock
               return -1;
}

// Шаг 3 связывание сокета с локальным адресом
sockaddr_in local_addr;
local_addr.sin_family=AF_INET;
local_addr.sin_port=htons(MY_PORT);   // не забываем о сетевом порядке!!!
local_addr.sin_addr.s_addr=0;         // сервер принимаем подключения
                                      // на все свои IP-адреса

// вызываем bind для связывания
if (bind(mysocket,(sockaddr *) &local_addr, sizeof(local_addr)))
{
        // Ошибка
        printf("Error bind %d\n",WSAGetLastError());
        closesocket(mysocket);        // закрываем сокет!
        WSACleanup();
        return -1;
}

// Шаг 4 ожидание подключений
// размер очереди - 0x100
if (listen(mysocket, 0x100))
{
        // Ошибка
        printf("Error listen %d\n",WSAGetLastError());
        closesocket(mysocket);
        WSACleanup();
        return -1;
}

printf("Ожидание подключений...\n");

// Шаг 5 извлекаем сообщение из очереди
SOCKET client_socket;                 // сокет для клиента
sockaddr_in client_addr;              // адрес клиента (заполняется системой)
```

```
        // функции accept необходимо передать размер структуры
        int client_addr_size=sizeof(client_addr);

        // цикл извлечения запросов на подключение из очереди
        while((client_socket=accept(mysocket, (sockaddr *) &client_addr, &client_addr_size)))
        {
                nclients++;             // увеличиваем счетчик подключившихся клиентов

                // пытаемся получить имя хоста
                HOSTENT *hst;
                hst=gethostbyaddr((char *) &client_addr.sin_addr.s_addr,4,AF_INET);

                // вывод сведений о клиенте
                printf("+%s [%s] new connect!\n",
                (hst)?hst->h_name:"",inet_ntoa(client_addr.sin_addr));
                PRINTNUSERS

                // Вызов нового потока для обслуживания клиента
                // Да, для этого рекомендуется использовать _beginthreadex
                // но, поскольку никаких вызов функций стандартной Си библиотеки
                // поток не делает, можно обойтись и CreateThread
                DWORD thID;
                CreateThread(NULL,NULL,SexToClient,&client_socket,NULL,&thID);
        }
        return 0;
}

// Эта функция создается в отдельном потоке
// и обсуживает очередного подключившегося клиента независимо от остальных
DWORD WINAPI SexToClient(LPVOID client_socket)
{
        SOCKET my_sock;
        my_sock=((SOCKET *) client_socket)[0];
        char buff[20*1024];
        #define sHELLO "Hello, Sailor\r\n"

        // отправляем клиенту приветствие
        send(my_sock,sHELLO,sizeof(sHELLO),0);

        // цикл эхо-сервера: прием строки от клиента и возвращение ее клиенту
        while( (int bytes_recv=recv(my_sock,&buff[0],sizeof(buff),0)) &&
        bytes_recv !=SOCKET_ERROR)
        send(my_sock,&buff[0],bytes_recv,0);

        // если мы здесь, то произошел выход из цикла по причине
        // возращения функцией recv ошибки - соединение с клиентом разорвано
        nclients--;                     // уменьшаем счетчик активных клиентов
        printf("-disconnect\n"); PRINTNUSERS

        // закрываем сокет
        closesocket(my_sock);
        return 0;
}
```

Пример реализации TCP-клиента

```
// Пример простого TCP-клиента
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>


#define PORT 666
#define SERVERADDR "127.0.0.1"

int main(int argc, char* argv[])
{
        char buff[1024];
        printf("TCP DEMO CLIENT\n");

        // Шаг 1 - инициализация библиотеки Winsock
        if (WSAStartup(0x202,(WSADATA *)&buff[0]))
        {
                printf("WSAStart error %d\n",WSAGetLastError());
                return -1;
        }

        // Шаг 2 - создание сокета
        SOCKET my_sock;
        my_sock=socket(AF_INET,SOCK_STREAM,0);
        if (my_sock<0)
        {
                printf("Socket() error %d\n",WSAGetLastError());
                return -1;
        }

        // Шаг 3 - установка соединения
        // заполнение структуры sockaddr_in - указание адреса и порта сервера
        sockaddr_in dest_addr;
        dest_addr.sin_family=AF_INET;
        dest_addr.sin_port=htons(PORT);
        HOSTENT *hst;

        // преобразование IP адреса из символьного в сетевой формат
        if (inet_addr(SERVERADDR)!=INADDR_NONE)
                dest_addr.sin_addr.s_addr=inet_addr(SERVERADDR);
        else
                // попытка получить IP адрес по доменному имени сервера
                if (hst=gethostbyname(SERVERADDR))
                        // hst->h_addr_list содержит не массив адресов,
                        // а массив указателей на адреса
                        ((unsigned long *)&dest_addr.sin_addr)[0]=
                                ((unsigned long **)hst->h_addr_list)[0][0];
                else
                {
                        printf("Invalid address %s\n",SERVERADDR);
                        closesocket(my_sock);
                        WSACleanup();
                        return -1;
                }
```

```
                // адрес сервера получен - пытаемся установить соединение
                if (connect(my_sock,(sockaddr *)&dest_addr,sizeof(dest_addr)))
                {
                        printf("Connect error %d\n",WSAGetLastError());
                        return -1;
                }

                printf("Соединение с %s успешно установлено\n\
                Type quit for quit\n\n",SERVERADDR);

        // Шаг 4 - чтение и передача сообщений
        int nsize;
        while((nsize=recv(my_sock,&buff[0],sizeof(buff)-1,0))!=SOCKET_ERROR)
        {
                // ставим завершающий ноль в конце строки
                buff[nsize]=0;

                // выводим на экран
                printf("S=>C:%s",buff);

                // читаем пользовательский ввод с клавиатуры
                printf("S<=C:"); fgets(&buff[0],sizeof(buff)-1,stdin);

                // проверка на "quit"
                if (!strcmp(&buff[0],"quit\n"))
                {
                        // Корректный выход
                        printf("Exit...");
                        closesocket(my_sock);
                        WSACleanup();
                        return 0;
                }

                // передаем строку клиента серверу
                send(my_sock,&buff[0],nsize,0);
        }

        printf("Recv error %d\n",WSAGetLastError());
        closesocket(my_sock);
        WSACleanup();
        return -1;
}
```

Пример реализации UDP-сервера

```
// Пример простого UDP-эхо сервера
#include <stdio.h>
#include <winsock2.h>

#define PORT 666                        // порт сервера
#define sHELLO "Hello, %s [%s] Sailor\n"
```

```
int main(int argc, char* argv[])
{
        char buff[1024];

        printf("UDP DEMO echo-Server\n");

        // Шаг 1 - подключение библиотеки
        if (WSAStartup(0x202,(WSADATA *) &buff[0]))
        {
                printf("WSAStartup error: %d\n",WSAGetLastError());
                return -1;
        }

        // Шаг 2 - создание сокета
        SOCKET my_sock;
        my_sock=socket(AF_INET,SOCK_DGRAM,0);
        if (my_sock==INVALID_SOCKET)
        {
                printf("Socket() error: %d\n",WSAGetLastError());
                WSACleanup();
                return -1;
        }

        // Шаг 3 - связывание сокета с локальным адресом
        sockaddr_in local_addr;
        local_addr.sin_family=AF_INET;
        local_addr.sin_addr.s_addr=INADDR_ANY;
        local_addr.sin_port=htons(PORT);

        if (bind(my_sock,(sockaddr *) &local_addr, sizeof(local_addr)))
        {
                printf("bind error: %d\n",WSAGetLastError());
                closesocket(my_sock);
                WSACleanup();
                return -1;
        }

        // Шаг 4 обработка пакетов, присланных клиентами
        while(1)
        {
                sockaddr_in client_addr;
                int client_addr_size = sizeof(client_addr);
                int bsize=recvfrom(my_sock,&buff[0],sizeof(buff)-1,0,
                (sockaddr *) &client_addr, &client_addr_size);
                if (bsize==SOCKET_ERROR)
                printf("recvfrom() error: %d\n",WSAGetLastError());

                // Определяем IP-адрес клиента и прочие атрибуты
                HOSTENT *hst;
                hst=gethostbyaddr((char *) &client_addr.sin_addr,4,AF_INET);
                printf("+%s [%s:%d] new DATAGRAM!\n",
                (hst)?hst->h_name:"Unknown host",
                inet_ntoa(client_addr.sin_addr),
                ntohs(client_addr.sin_port));
```

```
                // добавление завершающего нуля
                buff[bsize]=0;

                // Вывод на экран
                printf("C=>S:%s\n",&buff[0]);

                // посылка датаграммы клиенту
                sendto(my_sock,&buff[0],bsize,0,
                (sockaddr *)&client_addr, sizeof(client_addr));
        }
        return 0;
}
```

Пример реализации UDP-клиента

```
// пример простого UDP-клиента
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>

#define PORT 666
#define SERVERADDR "127.0.0.1"

int main(int argc, char* argv[])
{
        char buff[10*1014];
        printf("UDP DEMO Client\nType quit to quit\n");

        // Шаг 1 - иницилизация библиотеки Winsocks
        if (WSAStartup(0x202,(WSADATA *)&buff[0]))
        {
                printf("WSAStartup error: %d\n",WSAGetLastError());
                return -1;
        }

        // Шаг 2 - открытие сокета
        SOCKET my_sock=socket(AF_INET, SOCK_DGRAM, 0);
        if (my_sock==INVALID_SOCKET)
        {
                printf("socket() error: %d\n",WSAGetLastError());
                WSACleanup();
                return -1;
        }

        // Шаг 3 - обмен сообщений с сервером
        HOSTENT *hst;
        sockaddr_in dest_addr;

        dest_addr.sin_family=AF_INET;
        dest_addr.sin_port=htons(PORT);
```

```
        // определение IP-адреса узла
        if (inet_addr(SERVERADDR))
                dest_addr.sin_addr.s_addr=inet_addr(SERVERADDR);
        else
                if (hst=gethostbyname(SERVERADDR))
                        dest_addr.sin_addr.s_addr=((unsigned long **) hst->h_addr_list)[0][0];
                else
                {
                        printf("Unknown host: %d\n",WSAGetLastError());
                        closesocket(my_sock);
                        WSACleanup();
                        return -1;
                }

        while(1)
        {
                // чтение сообщения с клавиатуры
                printf("S<=C:");fgets(&buff[0],sizeof(buff)-1,stdin);
                if (!strcmp(&buff[0],"quit\n")) break;

                // Передача сообщений на сервер
                sendto(my_sock,&buff[0],strlen(&buff[0]),0,
                (sockaddr *) &dest_addr,sizeof(dest_addr));

                // Прием сообщения с сервера
                sockaddr_in server_addr;
                int server_addr_size=sizeof(server_addr);

                int n=recvfrom(my_sock,&buff[0],sizeof(buff)-1,0,
                (sockaddr *) &server_addr, &server_addr_size);

                if (n==SOCKET_ERROR)
                {
                        printf("recvfrom() error: %d\n",WSAGetLastError());
                        closesocket(my_sock);
                        WSACleanup();
                        return -1;
                }

                buff[n]=0;

                // Вывод принятого с сервера сообщения на экран
                printf("S=>C:%s",&buff[0]);
        }

        // Шаг последний - выход
        closesocket(my_sock);
        WSACleanup();

        return 0;
}
```