**ЛАБОРАТОРНАЯ РАБОТА №1**

**Создание последовательного сервера с установлением логического соединения TCP**

**Цель работы:** изучить методы создания серверов с установлением логического соединения *TCP*, используя алгоритм последовательной обработки запросов.

**1.1 Стек протоколов TCP/IP. История и перспективы стека TCP/IP**

В компьютерных сетях идеологической основой стандартизации являетсямногоуровневый подход к разработке средств сетевого взаимодействия.

*Протокол* - набор правил и действий (очерёдности действий), позволяющий осуществлять сетевое взаимодействие между двумя и более включёнными в [сеть](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D1%81%D0%B5%D1%82%D1%8C) устройствами в рамках одного уровня.

*Стек протоколов -* набор протоколов различных уровней, достаточный для организации взаимодействия в сети.

*Transmission Control Protocol/Интернет Protocol* *(TCP/IP)* - это промышленный стандарт стека протоколов, разработанный для глобальных сетей.

Агентство DARPA (Defense Advance Research Projects Agency) разработало стек TCP/IP (Transmission Control Protocol/Интернет Protocol) для объединения в сеть компьютеров различных подразделений министерства обороны США *(Department of Defence, DoD)* в 70-х годах прошлого века. В настоящее время стек TCP/IP является самым популярным средством организации составных сетей. До 1996 года бесспорным лидером был стек протоколов IPX/SPX компании Novell, но затем картина резко изменилась - стек TCP/IP по темпам роста числа установок намного стал опережать другие стеки, а с 1998 года вышел в лидеры и в абсолютном выражении.

Стандарты *TCP/IP* опубликованы в серии документов, названных *Request for Comments (RFC)*. Документы *RFC* описывают внутреннюю работу сети *Интернет*. Некоторые *RFC* описывают сетевые сервисы или протоколы и их реализацию, в то время как другие обобщают условия применения.

Лидирующая роль стека *TCP/IP* объясняется следующими его свойствами:

1. Это наиболее завершенный стандартный и в то же время популярный стек сетевых протоколов, имеющий многолетнюю историю.
2. Почти все большие сети передают основную часть своего трафика с помощью протокола *TCP/IP*.
3. Это метод получения доступа к сети *Интернет*.
4. Этот стек служит основой для создания *интранет* – корпоративной сети, использующей транспортные услуги сети *Интернет* и гипертекстовую технологию *WWW*.
5. Все современные операционные системы (ОС) поддерживают стек *TCP/IP*.
6. Это гибкая технология для соединения разнородных систем как на уровне транспортных подсистем, так и на уровне прикладных сервисов.
7. Это устойчивая масштабируемая межплатформенная среда для приложений клиент-сервер.

**1.2. Структура стека TCP/IP. Краткая характеристика протоколов**

Структура протоколов *TCP/IP* приведена на рисунке 1. Протоколы *TCP/IP* делятся на четыре уровня.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | **HTTP, HTTPs** | **SNMP** | **FTP** | **telnet** | **SMTP** | **TFTP** | **I** |
| **6** |
| **5** | **TCP** | | | | | **UDP** | **II** |
| **4** |
| **3** | **IP** | **ICMP** | | **RIP** | **OSPF** | **ARP**  **RARP** | **III** |
| **2** | **Не регламентируется Ethernet, Token Ring, FDDI, X.25, SLIP, PPP** | | | | | | **IV** |
| **1** |
| **Уровни модели OSI** |  | | | | | | **Уровни стека TCP/IP** |

Рисунок 1. Стек *TCP/IP*

Самый нижний (уровень IV) соответствует физическому и канальному уровням модели *OSI*. Этот уровень в протоколах *TCP/IP* не регламентируется, но поддерживает все популярные стандарты физического и канального уровня: для локальных сетей это *Ethernet, Fast Ethernet, Gigabit Eyhernet, 10GEthernet, 100GEthernet, 100VG-AnyLAN, Token Ring*, *FDDI*, для глобальных сетей – протоколы соединений «точка-точка» *SLIP* и *PPP*, протоколы территориальных сетей с коммутацией пакетов *X.25, Frame Relay*. Разработана также специальная спецификация, определяющая использование технологии *ATM* в качестве транспорта канального уровня. Обычно при появлении новой технологии локальных или глобальных сетей она быстро включается в стек *TCP/IP* за счет разработки соответствующего *RFC*, определяющего метод инкапсуляции пакетов *IP* в ее кадры.

Следующий уровень (уровень III) – это уровень межсетевого взаимодействия, который занимается передачей пакетов с использованием различных транспортных технологий локальных и территориальных сетей, линий специальной связи и т. п.

В качестве основного протокола сетевого уровня (в терминах модели *OSI*) в стеке используется протокол *IP*, который изначально проектировался как протокол передачи пакетов в составных сетях, состоящих из большого количества локальных сетей, объединенных как локальными, так и глобальными связями. Поэтому протокол *IP* хорошо работает в сетях со сложной топологией, рационально используя наличие в них подсистем и экономно расходуя пропускную способность низкоскоростных линий связи. Протокол *IP* является дейтаграммным протоколом, то есть он не гарантирует доставку пакетов до узла назначения, но старается это сделать.

К уровню межсетевого взаимодействия относятся и все протоколы, связанные с составлением и модификацией таблиц маршрутизации, такие как протоколы сбора маршрутной информации *RIP*(Routing Интернет Protocol) и *OSPF (Open Shortest Path First)*, а также протокол межсетевых управляющих сообщений *ICMP (Интернет Control Message Protocol)*. Последний протокол предназначен для обмена информацией об ошибках между маршрутизаторами сети и узлом – источником пакета. С помощью специальных пакетов *ICMP* сообщается о невозможности доставки пакета, о превышении времени жизни или продолжительности сборки пакета из фрагментов, об аномальных величинах параметров, об изменении маршрута пересылки и типа обслуживания, о состоянии системы и т.п.

Следующий уровень (уровень II) называется основным. На этом уровне функционируют протокол управления передачей*TCP* и протокол дейтаграмм пользователя *UDP*. Протокол *TCP* обеспечивает надежную передачу сообщений между удаленными прикладными процессами за счет образования виртуальных соединений. Протокол *UDP* обеспечивает передачу прикладных пакетов дейтаграммным способом, как и *IP*, но выполняет только функции связующего звена между сетевым протоколом и многочисленными прикладными процессами.

Верхний уровень (уровень I) называется прикладным. За долгие годы использования в сетях различных стран и организаций стек *TCP/IP* накопил большое количество протоколов и сервисов прикладного уровня. К ним относятся такие широко используемые протоколы, как протокол копирования файлов *FTP*, протокол эмуляции терминала *telnet*, почтовый протокол *SMTP*, используемый в электронной почте сети *Интернет*, гипертекстовые сервисы доступа к удаленной информации, такие как *WWW,* и многие другие. Остановимся несколько подробнее на некоторых из них.

Протокол пересылки файлов *FTP (File Transfer Protocol)* реализует удаленный доступ к файлу. Для того, чтобы обеспечить надежную передачу, *FTP* использует в качестве транспорта протокол с установлением соединений – *TCP*. Кроме пересылки файлов протокол *FTP* предлагает и другие услуги.

В стеке *TCP/IP* протокол *FTP* предлагает наиболее широкий набор услуг для работы с файлами, однако он является и самым сложным для программирования. Приложения, которым не требуются все возможности *FTP*, могут использовать другой, более экономичный протокол – простейший протокол пересылки файлов *TFTP (Trivial File Transfer Protocol)*. Этот протокол реализует только передачу файлов, причем в качестве транспорта используется более простой, чем *TCP*, протокол без установления соединения – *UDP*.

Протокол *telnet* обеспечивает передачу потока байтов между процессами, а также между процессом и терминалом. Наиболее часто этот протокол используется для эмуляции терминала удаленного компьютера. При использовании сервиса *telnet* пользователь фактически управляет удаленным компьютером так же, как и локальный пользователь, поэтому такой вид доступа требует хорошей защиты.

Протокол*SNMP (Simple Network Management Protocol)* используется для организации сетевого управления. Изначально протокол *SNMP* был разработан для удаленного контроля и управления маршрутизаторами *Интернет*, которые традиционно часто называют также шлюзами. С ростом популярности протокол *SNMP* стали применять и для управления любым коммуникационным оборудованием – концентраторами, мостами, сетевыми адаптерами и т.д.

**1.3. Протокол TCP**

Протокол *TCP (Transmission Control Protocol)* работает, как и протокол *UDP*, на транспортном уровне. Он обеспечивает надежную транспортировку данных между прикладными процессами путем установления логического соединения.

Единицей данных протокола *TCP* является сегмент. Информация, поступающая к протоколу *TCP* в рамках логического соединения от протоколов более высокого уровня, рассматривается протоколом *TCP* как неструктурированный поток байтов. Поступающие данные буферизуются средствами *TCP*. Для передачи на сетевой уровень из буфера «вырезается» некоторая непрерывная часть данных, называемая сегментом.

Не все сегменты, посланные через соединение, будут одного и того же размера, однако оба участника соединения должны договориться о максимальном размере сегмента, который они будут использовать. Этот размер выбирается таким образом, чтобы при упаковке сегмента в *IP*-пакет он помещался туда целиком, то есть максимальный размер сегмента не должен превосходить максимального размера поля данных *IP*-пакета. В противном случае пришлось бы выполнять фрагментацию, то есть делить сегмент на несколько частей для того, чтобы он вместился в *IP*-пакет.

Аналогичные проблемы решаются и на сетевом уровне. Для того чтобы избежать фрагментации, должен быть выбран соответствующий максимальный размер *IP*-пакета. Однако при этом должны быть приняты во внимание максимальные размеры поля данных кадров (*MTU*) всех протоколов канального уровня, используемых в сети. Максимальный размер сегмента не должен превышать минимальное значение на множестве всех *MTU* составной сети.

**1.4. Установление TCP-соединений**

В протоколе *TCP*, как и в *UDP*, для связи с прикладными процессами используются порты. Номера портам присваиваются следующим образом: имеются стандартные, зарезервированные номера (например, номер 21 закреплен за сервисом *FTP*, 23 – за *telnet*), а менее известные приложения пользуются произвольно выбранными локальными номерами.

Как говорилось выше, для организации надежной передачи данных предусматривается установление *логического соединения* между двумя прикладными процессами. В рамках соединения осуществляется обязательное подтверждение правильности приема для всех переданных сообщений, и при необходимости выполняется повторная передача. Соединение в *TCP* позволяет вести передачу данных одновременно в обе стороны, то есть полнодуплексную передачу.

Соединение в протоколе *TCP* идентифицируется парой полных адресов обоих взаимодействующих процессов (оконечных точек). Адрес каждой из оконечных точек включает *IP*-адрес (номер сети и номер компьютера) и номер порта. Одна оконечная точка может участвовать в нескольких соединениях.

Установление соединения выполняется в следующей последовательности:

1. При установлении соединения одна из сторон является инициатором. Она посылает запрос к протоколу *TCP* на открытие порта для передачи (*active open*).
2. После открытия порта протокол *TCP* на стороне процесса-инициатора посылает запрос процессу, с которым требуется установить соединение.
3. Протокол *TCP* на приемной стороне открывает порт для приема данных (*passive open*) и возвращает квитанцию, подтверждающую прием запроса.
4. Для того чтобы передача могла вестись в обе стороны, протокол на приемной стороне также открывает порт для передачи (*active port*) и также передает запрос к противоположной стороне.
5. Сторона-инициатор открывает порт для приема и возвращает квитанцию. Соединение считается установленным. Далее происходит обмен данными в рамках данного соединения.

**1.5. Алгоритм работы последовательного сервера с установлением логического соединения**

Покажем обобщенный алгоритм работы последовательного сервера с установлением логического соединения:

1. Создать сокет и установить связь с локальным адресом.
2. Перевести сокет в пассивный режим, подготавливая его для использования сервером.
3. Принять из сокета следующий запрос на установление соединения и получить новый сокет для соединения.
4. Считывать в цикле запросы от клиента, формировать ответы и отправлять клиенту.
5. После завершения обмена данными с конкретным клиентом закрыть соединение и возвратиться к этапу 3 для приема нового запроса на установление соединения.

На рисунке 2 показана упрощенная схема организации работы последовательного сервера с установлением логического соединения.

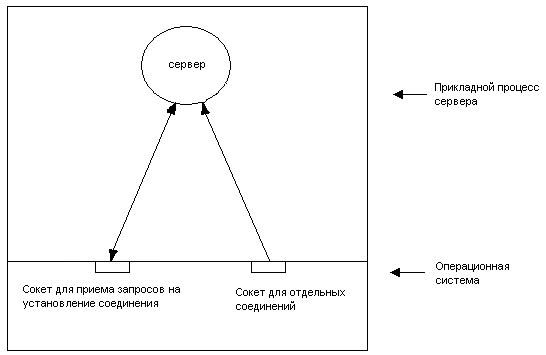


Рисунок. 2. Схема организации работы последовательного

сервера с установления логического соединения

**1.6. Методические указания по созданию последовательного сервера с установлением логического соединения (*TCP*)**

В качестве примера приведем следующую задачу.

Осуществить взаимодействие клиента и сервера на основе протокола TCP. Функционирование клиента и сервера реализовать следующим образом: клиент посылает произвольный набор символов серверу и получает назад количество символов «а» в этом наборе.

Необходимо написать два проекта на языке программирования *С* – клиент и сервер. Начнем с серверной части.

Серверная часть

При разработке приложений для клиента и сервера для обмена структурами данных или пакетами используются сокеты. Сокет – это абстрактный объект для обозначения одного из концов сетевого соединения. Он предназначен для создания механизма обмена данными. Реализация сокетов осуществляется в *API WinSock*.

В версии 1.1 WinSock любого поставщика имеется библиотека *WSOCK32.DLL* (или *winsock .dll* для 16-разрядных операционных систем), позволяющая реализовать программный интерфейс *WinSock*. Интерфейс версии 2 в системе Windows поддерживается одной динамической библиотекой *WS2\_32. dll*, которая для обслуживания различных сетей может использовать протоколы и системы распознавания имен различных разработчиков. Библиотека *WS**2\_32.dll* поддерживает как функции *WinSock 1.1*, так и ряд дополнительных функций, впервые появившихся в спецификации *WinSock 2*. Данную библиотеку необходимо подключить к проекту, выполненному в *VC++*: *Project –>Settings – вкладка Links* – к списку подключаемых библиотек через пробел добавляем *ws2\_32.lib* .

В тексте программы этот интерфейс разработки приложений подключается с помощью директивы *#include*:

#include <winsock2.h>

Кроме того, подключим уже известные заголовочные файлы :

#include <iostream.h>

#include <stdio.h>

#include <stdlib.h>

Для того чтобы можно было использовать интерфейс программирования *WinSock*, его необходимо инициализировать с помощью функции *WSAStartup(wVersionRequested,&wsaData)*.

Первый параметр функции *WSAStartup()* — это значение типа *word*, которое определяет максимальный номер версии *WinSock*, доступный приложению. Первая цифра версии находится в младшем байте, вторая – в старшем.

Функция *WSAStartup()* возвращает значение *wsasysnotready*, если динамическая библиотека поддержки *WinSock* или соответствующая подсистема сети не инициализирована, инициализирована некорректно или не найдена. Кроме того, с помощью этой функции приложение сообщает системе версию *WinSock*, которая должна использоваться. Как правило, при вызове функции *WSAStartup()* необходимо указывать максимальный допустимый номер версии. Если он меньше, чем версии, поддерживаемые данной динамической библиотекой, функция *WSAStartup()* возвратит значение *wsavernotsupported*.

Второй параметр – структура *wsaData* – содержит номер версии, которая должна использоваться (поле *wVersion*), максимальный номер версии, поддерживаемый данной библиотекой (поле *wHighVersion*), текстовые строки с описанием реализации *WinSock*, максимальное число сокетов, доступных процессу и максимально допустимый размер дейтаграмм.

В нашей программе это описывается так:

int main(){

WORD wVersionRequested;

WSADATA wsaData;

wVersionRequested=MAKEWORD(2,2);

WSAStartup(wVersionRequested,&wsaData);

В данной работе нас интересуют сокеты потоков, которые позволяют гарантировать бесперебойную доставку данных в нужном порядке и без дублирования. В *TCP/IP*-реализации *WinSock* сокеты потоков используют протокол *TCP* (*Transmission Protocol*). Сокеты потоков особенно полезны для пересылки больших объемов данных без потерь и нарушения порядка. Кроме того, при закрытии соединения приложения получат извещение об этом событии.

Для создания сокета используется функция *socket(domain,type,protocol)*. Она принимает три параметра: домен, тип сокета и протокол. Домен – это абстракция, подразумевающая конкретную структуру адресации и протоколы, определяющие типы сокетов внутри домена. Примерами коммуникационных доменов могут быть: *UNIX* домен, *Интернет* домен, и т.д. В*Интернет* домене сокет – это комбинация *IP*-адреса и номера порта, которая однозначно определяет отдельный сетевой процесс во всей глобальной сети *Интернет*. Два сокета, один для хоста-получателя, другой – для хоста-отправителя, определяют соединение для протоколов, ориентированных на установление связи, таких, как *TCP*.

Вызов функции *socket()* выглядит следующим образом:

SOCKET s = socket(AF\_INET, SOCK\_STREAM, 0);

Первый параметр означает, что с этим сокетом будут использоваться адреса *Интернет*; следующие два аргумента задают тип создаваемого сокета и протокол обмена данными через него. В приведенном примере создается сокет потока, использующий протокол *TCP*.

Если третий параметр функции *socket()* сделать равным нулю, протокол будет выбран автоматически в зависимости от семейства адресов и типа сокета. Можно явно указать константы:

*IPPROTO\_UDP* – протокол *UDP* (смотри лабораторную работу №2),

*IPPROTO\_TCP* – протокол *TCP/IP*.

Если функция *socket()* выполняется успешно, она возвращает дескриптор нового сокета. Если же ее работа завершается аварийно, возвращается значение 0, и для получения подробной информации об ошибке необходимо вызвать функцию *WSA**GetLastError ( ).*

Для связывания конкретного адреса с сокетом используется функция *bind (s, addr, addrlen)*. В нее передается дескриптор сокета, указатель на структуру адреса и длина этой структуры. Дескриптор сокета – это значение, которое возвращает функция *socket()*. Структура адреса – это структура типа *sockaddr\_in*.

struct sockaddr\_in local;

local.sin\_family=AF\_INET;

local.sin\_port=htons(1280);

local.sin\_addr.s\_addr=htonl(INADDR\_ANY);

int c=bind(s,(struct sockaddr\*)&local,sizeof(local));

В поле *sin\_addr* структуры *sockaddr\_in* хранится физический *IP*-адрес компьютера в формате структуры *in\_addr*, описанной в заголовочном файле *winsock2.h*. Вместо поля *s\_addr* можно подставлять *INADDR\_ANY*. это позволяет сокету посылать или принимать данные через любой *IP*-адрес данного компьютера. Обычно компьютер имеет только один *IP*-адрес, хотя в принципе на нем может быть установлено несколько сетевых адаптеров, каждый со своим *IP*-адресом. Если сокет должен использовать только один из них, его необходимо указать явно. Для этого нередко используется функция *inet\_addr(“…” )* , которая принимает в качеств аргумента *ASCII*-строку десятичной нотации *IP*-адреса с точкой и возвращает переменную типа u *long*, содержащую этот адрес в формате поля *s\_addr*. Кроме нее, существует функция *inet\_ntoa(address),* которая выполняет обратное преобразование, принимая переменную типа *u\_long* и возвращая адрес в виде *ASCII*-строки.

Поле *sin**\_family* всегда имеет значение *AF\_INET*. Поле *sin\_port* определяет порт, который будет ассоциирован с сокетом.

Для привязки приложение может использовать любой номер порта от 1 до 65535, хотя этот диапазон обычно делится на следующие поддиапазоны:

0 – нe используется. Если передать 0 в качестве номера порта, будет автоматически выбран используемый порт с номером между 1 024 и 5 000.

1 – 255 – зарезервированы для сетевых служб: *FTP*, *telnet*, *finger* и т.д.

256 – 1 023 – зарезервированы для других служб общего назначения, например функций маршрутизации.

1024 – 4999 – служат для портов клиентов. Обычно сокеты приложений-клиентов используют номера портов именно из этого диапазона.

5000 – 65535. Используются для определяемых пользователем портов приложений-серверов.

Вместо простого присвоения констант полей *sin\_port* и *sin\_addr* использовалось преобразование типов с помощью функций *htons (n )* и *htonl(n)* Эти функции предназначены для изменения порядка следования байтов в параметрах порта и адреса, для преобразования их в общий сетевой формат для 16-разрядных и 32-разрядных значений соответственно.

После создания сокета и привязки его к адресу необходимо каким-то образом установить соединение с клиентом. Для этого используется функция *listen (s, l )* , которая помещает сокет в состояние прослушивания:

int r=listen(s,5);

Вызов этой функции инициирует ожидание запроса клиента на открытие соединения. Параметр *l* содержит количество запросов, которое должно поступить для того, чтобы приложение согласилось установить соединение. Например, если этот параметр равен 2 и приложение по каким-то причинам отказалось открыть соединение, третий клиент, который попытается подключиться к серверу, получит код ошибки *wsaeconnrefused*. Первые два запроса будут отправлены в очередь для их последующей обработки сервером.

При получении запроса клиента открытие соединения выполняется с помощью функции *accept()* :

SOCKET accept (SOCKET s, struct sockaddr FAR \* addr, int FAR \* addrlen)

Как обычно, в качестве первого параметра передается сокет, ожидающий запроса. Второй и третий параметры используются для получения адреса сокета клиента, который запрашивает соединение. Если соединение открывается успешно, функция *accept()* возвращает дескриптор на новый сокет, который будет использоваться для управления новым соединением. Если произошла ошибка, функция *accept()* возвращает код *invalid\_socket*, и для получения более подробной информации об ошибке необходимо вызвать функцию *WSAGetLastError()* .

Исходный сокет продолжит ожидание запросов на новые соединения, которые затем открываются снова с помощью функции *accept**()*. Каждое открытое соединение управляется отдельным сокетом, дескриптор которого возвращается из этой функции.

На нашем примере это выглядит так:

while (true){

char buf[255],res[100],b[255],\*Res;

//структура определяет удаленный адрес,

//с которым соединяется сокет

sockaddr\_in remote\_addr;

int size=sizeof(remote\_addr);

SOCKET s2=accept(s,(struct sockaddr\*)&remote\_addr,&size);

Для выполнения задачи нам необходимо осуществлять прием и передачу данных. Ввод исходной строки выполнит клиент и передаст ее серверу, чтобы тот проанализировал ее и отослал назад клиенту количество букв «а» в этой строке.

Для приема данных через сокет потока используется функция *recv*(). Вот ее прототип: *int recv (SOCKET* *s, char FAR**\* buf**, int* *len**, int flags);* Параметры *buf* и *len* определяют соответственно буфер для приема данных и его длину. Параметр *flags* может принимать значения *MSG\_OOB* для приема привилегированных данных или *MSG\_ PEEK* для заполнения буфера без удаления данных из входной очереди, но, как правило, мы пишем его равным нулю.

Если во входной очереди находятся данные для сокета, функция *recv()* возвращает количество прочитанных байтов, которое равно объему доступных данных во входной очереди и не превосходит значения *len*. При корректном закрытии соединения возвращается значение 0, а при аварийном – значение SOCKET\_ERROR. Для определения точного кода ошибки необходимо вызвать функцию *WSAGetLastError()* .

Пересылка данных выполняется с помощью функции *send*() : int send (SOCKET s, const char FAR \*buf, int len, int flags).

Функция *send()* принимает в качестве аргументов указатель на буфер, содержащий пересылаемые данные, и его длину, а также параметр *flags*. Если этот параметр равен *msg\_dontroute*, в пересылаемый набор данных не включается информация о маршрутизации; если его значение равно *msg\_oob*, посылается поток привилегированных (*out-of-band*) данных.

Объем данных, пересылаемых одним вызовом функции *send()* , не должен превышать размера пакета, максимально допустимого в данной сети. При попытке пересылки большего объема данных функция *send()* завершится аварийно, а функция *WSAGetLastError()* возвратит код ошибки *WSAEMSGSIZE*.

Работу с одним клиентом поместим в цикл, чтобы была возможность вводить несколько строк.

while (recv(s2,b,sizeof(b),0)!=0){

int i=0;

for (unsigned j=0;j<=strlen(b);j++)

if (b[j]=='a') i++;

\_itoa(i,res,10);

Res=new char[strlen(res)+1];

strcpy(Res,res);

Res[strlen(res)]='\0';

//Посылает данные на соединенный сокет

send(s2,Res,sizeof(Res)-2,0);

}

Для завершения работы сокета его необходимо закрыть с помощью функции *closesocket*() : *int closesocket (SOCKET s)*. Эта функция принимает единственный аргумент — дескриптор закрываемого сокета, но ее поведение определяется также параметрами сокета, установленными с помощью функции *setsockopt()*. Текущие параметры можно узнать, вызвав функцию *getsockopt()*. Результат работы функции *closesocket()* определяется параметрами *SO\_LINGER* и *SO\_DONTLINGER*.

Если параметр *SO\_DONTLINGER* равен *TRUE*, функция *closesocket()* возвратит значение «немедленно», но перед закрытием сокета будет предпринята попытка пересылки всех оставшихся данных. Обычно это называется корректным закрытием.

Если параметр *SO\_LINGER* равен *TRUE* и установлена ненулевая задержка, также выполняется корректное закрытие с попыткой пересылки всех оставшихся в буфере данных, но функция *closesocket()* не возвращает значения до тех пор, пока не будут пересланы все данные или пока не истечет срок задержки. Если параметр *SO\_LINGER* равен *TRUE* и задержка равна нулю, сокет закрывается немедленно, а все оставшиеся в буфере данные теряются.

Закроем сокет в нашей программе:

closesocket(s2);

}

Другие способы закрытия сокета. Если сокет больше не используется, процесс может закрыть его с помощью функции *close (s)*, вызвав ее с соответствующим дескриптором сокета: *close(s)*.

Если данные были ассоциированы с сокетом, обещающим доставку (сокет типа *stream*), система будет пытаться осуществить передачу этих данных. Тем не менее, по истечении довольно-таки длительного промежутка времени, если данные все еще не доставлены, они будут отброшены. Если пользовательский процесс желает прекратить любую передачу данных, он может сделать это с помощью вызова *shutdown* на данном сокете для его закрытия. Вызов *shutdown* вызывает «моментальное» отбрасывание всех стоящих в очереди данных. Формат вызова следующий: *shutdown(s, how)*, где *how* имеет одно из следующих значений:

0 – если пользователь больше не желает читать данные;

1 – если данные больше не будут посылаться;

2 – если данные не будут ни посылаться, ни получаться.

Завершая программу, нужно прекратить работу с *WinSock DLL*, вызвав функцию:

WSACleanup();

}

Клиентская часть

Ниже приведена программа клиента.

#include <winsock2.h>

#include <iostream.h>

#include <stdlib.h>

int main(){

WORD wVersionRequested;

WSADATA wsaData;

wVersionRequested=MAKEWORD(2,2);

WSAStartup(wVersionRequested,&wsaData);

struct sockaddr\_in peer;

peer.sin\_family=AF\_INET;

peer.sin\_port=htons(1280);

//т.к. клиент и сервер на одном компьютере,

// пишем адрес 127.0.0.1

peer.sin\_addr.s\_addr=inet\_addr("127.0.0.1");

SOCKET s=socket(AF\_INET,SOCK\_STREAM,0);

connect(s,(struct sockaddr\*) &peer,sizeof(peer));

char buf[255],b[255];

cout<<"Enter the string, please"<<endl;

cin.getline(buf,100,'\n');

send(s,buf,sizeof(buf),0);

if (recv(s,b,sizeof(b),0)!=0){

b[strlen(b)]='\0'; //Удаление ненужных символов

// в конце строки

cout<<b<<endl;

}

closesocket(s);

WSACleanup();

return 0;

}

Клиентская часть использует функции, которые описаны ранее. Новая функция, которая НЕ вызывается в серверной части, – *connect (s, addr, addrlen).*С помощью этой функции приложение-клиент посылает запрос на открытие соединения. Параметры *addr*, *addrlen* используются для указания адреса и порта, к которому необходимо подсоединиться. Структура *sockaddr*, передаваемая в функцию *connect()*, должна быть идентичной структуре, передаваемой в функцию *bind()* на сервере.

**1.7. Индивидуальные задания**

Разработать приложение, реализующее архитектуру «клиент-сервер». Необходимо реализовать последовательный сервер с установлением логического соединения (*TCP*). Логику взаимодействия клиента и сервера реализовать следующим образом:

1. Клиент посылает два числа серверу и одну из математических операций: «\*», «/», «+», «–» ,– сервер соответственно умножает, делит, складывает либо вычитает эти два числа и посылает ответ назад клиенту.
2. Клиент посылает слово серверу, сервер возвращает назад в обратном порядке следования букв это слово клиенту.
3. Клиент посылает два числа серверу *m* и *n*, сервер возвращает *m!+n!* этих чисел назад клиенту.
4. Клиент посылает два слова серверу, сервер их сравнивает и возвращает «истина», если они одинаковы по количеству и порядку следования в них букв, и «ложь»– при невыполнении хотя бы одного из этих условий.
5. Клиент посылает произвольный набор латинских букв серверу и получает их назад упорядоченными по алфавиту.
6. Клиент посылает серверу произвольный набор символов, сервер замещает каждый четвертый символ на «%».
7. Сервер генерирует прогноз погоды на неделю. Клиент посылает день недели и получает соответствующий прогноз.
8. Клиент посылает серверу произвольные числа и получает назад количество чисел, кратных трем.
9. Клиент посылает серверу символьную строку, содержащую пробелы, и получает назад ту же строку, но в ней между словами должен находиться только один пробел.
10. Клиент посылает серверу слово. Сервер определяет, является ли это слово палиндромом (*палиндром* – слово, читающееся одинаково как слева направо и справа налево).
11. Клиент посылает серверу два числа и получает назад НОД (наибольший общий делитель) этих чисел.
12. Клиент посылает серверу число от 0 до 10 и получает назад название этого числа прописью.
13. Клиент посылает серверу координаты точки Х и У в декартовой системе координат. Сервер определяет, в какой координатной четверти находится данная точка и посылает результат назад клиенту.
14. Клиент посылает серверу координаты прямоугольной области и точки в декартовой системе координат. Сервер определяет, лежит ли данная точка в прямоугольной области, и посылает результат назад клиенту.
15. Клиент посылает серверу шестизначный номер билета. Сервер определяет, является ли этот билет «счастливым». «Счастливым» называется такой билет, у которого сумма первых трех цифр равна сумме последних трех. Сервер посылает результат назад клиенту.

**1.8. Контрольные вопросы**

1. Какая технология называется межсетевым обменом (*Интернетworking*)?
2. Объясните понятие «протоколы» в контексте технологий обмена данными. Что они включают? Приведите примеры.
3. Назовите отличия *TCP/IP* от других средств передачи данных.
4. Дайте определение понятию «сокет».
5. Опишите функцию, которая используется для приема данных через сокет потока (протокол *TCP*).
6. Назовите функцию, используемую для создания сокета. Опишите ее параметры.
7. Опишите функцию, которая используется для пересылки данных через сокет потока (протокол *TCP*).
8. Что возвращает функция *accept()* в том случае, если соединение открывается успешно?
9. Назовите функцию, которая используется в приложении-клиенте для посылки запроса на открытие соединения. Опишите ее параметры.