Содержание

ВВЕДЕНИЕ *5*

ЛАБОРАТОРНАЯ РАБОТА №1 Создание последовательного сервера с установлением логического соединения TCP **6**

1.1 Стек протоколов TCP/IP. История и перспективы стека TCP/IP 6
1.2 Структура стека TCP/IP. Краткая характеристика протоколов 7
1.3 Протокол TCP 9
1.4 Установление TCP-соединений 9
1.5. Алгоритм работы последовательного сервера с установлениемлогического соединения 11
1.6 Методические указания по созданию последовательного сервера сустановлением логического соединения (*TCP*) 12
1.7 Индивидуальные задания 19
1.8 Контрольные вопросы 20

ЛАБОРАТОРНАЯ РАБОТА №2 Создание последовательного сервера без установления логического соединения UDP **22**

2.1 Протокол UDP 22
2.2 Сокеты дейтаграмм 22
2.3 Алгоритм работы последовательного сервера без установления логического соединения 23
2.4 Методические указания по созданию последовательногосервера без установления логического соединения UDP 24
2.5 Индивидуальные задания 27
2.6 Контрольные вопросы 29

ЛАБОРАТОРНАЯ РАБОТА №3 Создание параллельного многопоточного серверас установлением логического соединения TCP **30**

3.1 Потоки 30
3.2 Преимущества многопоточности 31
3.3 Преимущества и недостатки многопотоковых процессов 32
3.4. Алгоритм работы параллельного (многопоточного) сервера с установлением логического соединения 33
3.5. Методические указания по созданию параллельного многопоточного сервера с установлением логического соединения TCP 34
3.6 Индивидуальные задания 39
3.7 Контрольные вопросы 42

ЛАБОРАТОРНАЯ РАБОТА №4 Создание параллельного многопроцессного серверас установлением логического соединения TCP **44**

4.1 Основные сведения о процессах 44
4.2 Создание процессов 44
4.3 Различие между процессами и потоками 46
4.4 Алгоритм работы параллельного (многопроцессного) сервера сустановлением логического соединения 47
4.5 Методические указания по созданию параллельного сервера с установлением логического соединения TCP, использующего отдельный процесс для обработки запросов клиента 48
4.6 Индивидуальные задания 55
4.7 Контрольные вопросы 58

ЛАБОРАТОРНАЯ РАБОТА №5 Создание однопотоковых параллельных серверов TCP .59

5.1 Однопотоковые параллельные серверы 59
5.2 Использование функции select 60
5.3 Методические указания по созданию однопотокового параллельного сервера с установлением логического соединения 62
5.4 Индивидуальные задания 66
5.5 Контрольные вопросы 68

Лабораторная работа №6 Создание параллельного сервера с установлением логического соединения с помощью пула потоков/процессов **69**

6.1 Пул потоков/процессов 69
6.2 Методические указания по созданию параллельного сервера с установлением логического соединения TCP, использующего пул потоков для обработки запросов клиента 70
6.3 Индивидуальные задания 72
6.4 Контрольные вопросы 74
ЛИТЕРАТУРА 75

ВВЕДЕНИЕ

Дисциплина «Компьютерные сети» является базовой в цикле дисциплин, ориентированных на применение сетей и сетевых технологий в решении профессиональных задач, изучаемых студентами на последующих курсах обучения по специальности «Информационные системы и технологии (в экономике)».

Цель изучения данной дисциплины – овладение знаниями использования сетевых средств и базовых технологий программирования, а также приобретения основных навыков разработки программ сетевого взаимодействия, для использования при разработке сетевых информационных приложений.

Лабораторный практикум ориентирован на приобретение студентами знаний и базовых навыков разработки программ сетевого взаимодействия, получение навыков программной реализации протоколов среднего и верхнего уровня стека протоколов TCP/IP, а также опыта решения практических задач на основе технологии разделения функций между клиентом и сервером, что является основой организации архитектуры клиент-сервер.

В материалах практикума рассматриваются методика и принципы программной реализации последовательного, параллельного или параллельно-последовательного клиент-серверного взаимодействия. Значимость программных решений, представленных здесь, весьма существенная, поскольку программирование рассматривается с применением низкоуровневых программных абстракций, что позволяет достаточно хорошо изучить физическую основу процессов и методов взаимодействия узлов в компьютерных сетях. Рассмотренные схемы и алгоритмы сетевых взаимодействий дают представление о базовых концепциях, положенных в основу серверных разработок и организации синхронных взаимодействий приложений, формируют основу для понимания архитектурных принципов разработки современных распределенных информационных систем, способствуют более четкому и детальному пониманию основных алгоритмов и методов организации взаимодействия распределенных объектов и служб информационных систем.

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

Протокол – набор правил и действий (очерёдности действий), позволяющий осуществлять соединение и обмен данными между двумя и более включёнными в сеть устройствами.

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.

<table>
<tr><td>7</td><td rowspan="2">HTTP, HTTPs</td><td rowspan="2">SNMP</td><td rowspan="2">FTP</td><td rowspan="2">telnet</td><td rowspan="2">SMTP</td><td rowspan="2">TFTP</td><td rowspan="2">I</td></tr>
<tr><td>6</td></tr>
<tr><td>5</td><td colspan="5" rowspan="2">TCP</td><td rowspan="2">UDP</td><td rowspan="2">II</td></tr>
<tr><td>4</td></tr>
<tr><td>3</td><td>IP</td><td colspan="2">ICMP</td><td>RIP</td><td>OSPF</td><td>ARP
RARP</td><td>III</td></tr>
<tr><td>2</td><td colspan="6" rowspan="2">Не регламентируется
Ethernet, Token Ring, FDDI, X.25, SLIP, PPP</td><td rowspan="2">IV</td></tr>
<tr><td>1</td></tr>
<tr><td colspan="7">Уровни модели OSI</td><td>Уровни стека TCP/IP</td></tr>
</table>

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 Ethernet, 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 дейтаграммы 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 (Internet 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 хорошейзащиты.

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: име-

ются 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.

Для *UDP* соединений существует своя схема взаимодействия между приложениями, которая будет подробно рассмотрена в следующих лабораторных работах.

Схема работы сервера определяется не только протоколом взаимодействия с клиентом (*TCP* или *UDP*), но и тем, какой механизм реализован на сервере по обработке клиентских запросов (последовательный или параллельный). В данном практикуме будут рассмотрены следующие типы серверов:

- последовательный сервер с установлением логического соединения;
- последовательный сервер без установления логического соединения;
- параллельный многопоточный сервер с установлением логического соединения;
- параллельный многопроцессный сервер с установлением логического соединения;

- однопотоковый параллельный сервер с установлением логического соединения;
- параллельный сервер с установлением логического соединения, использующий пул потоков.

Определить, какой из типов серверов использовать, достаточно непросто, так как для оценки работы сервера следует анализировать несколько параметров, как минимум это, время реакции, максимальное число обслуженных запросов в секунду (интенсивность), число отказов обслуживания, и на какой интенсивности они начинают возникать. Всё определяется теми задачами, которые решает сервер, и тем окружением, в котором он функционирует. Это могут быть и области, где наилучшим решением окажется простейший последовательный сервер и такие, где это решение неприемлемо. Кроме того, следует учитывать и такие аспекты, как трудоёмкость реализации, потребление ресурсов (в частности оперативной памяти), простота отладки, сопровождения и др.

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 схема организации работы последовательного сервера с установлением логического соединения.

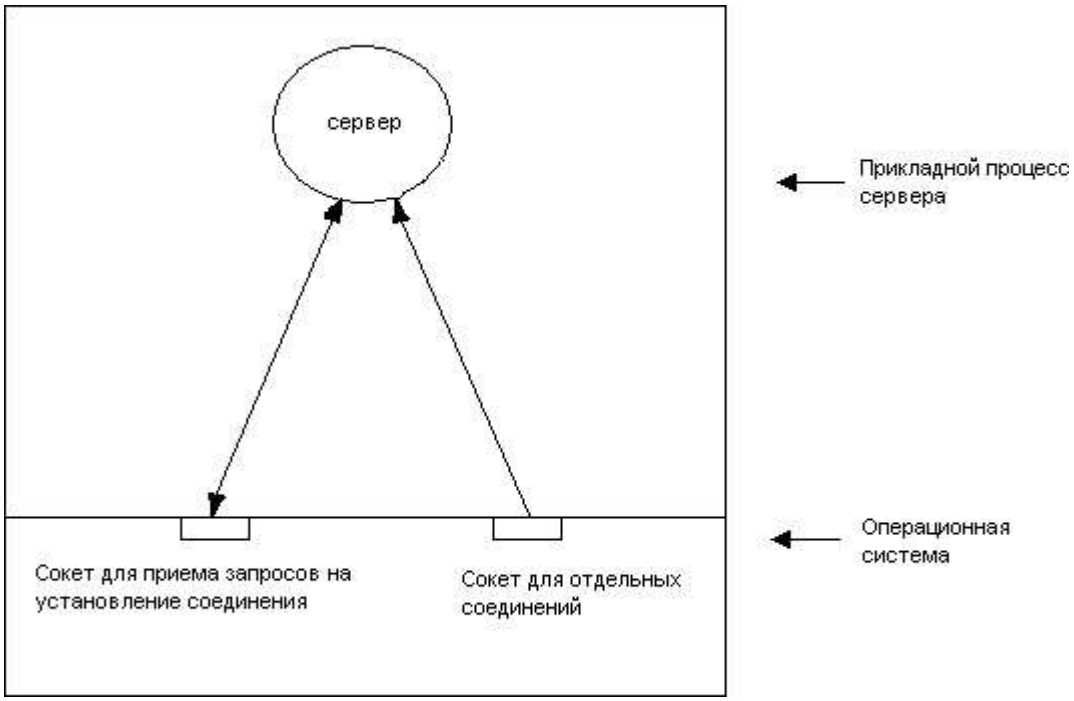

Рисунок 2 – Схема организации работы последовательного сервера с установления логического соединения

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 *C* – 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 *WS2_32.dll* поддерживает как функции *WinSock 1.1*, так и ряд дополнительных функций, впервые появившихся в спецификации *WinSock 2*. Данную библиоте-

ку 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.

Инициализация *WinSock* с помощью функции *WSAStartup()* в нашей программе описывается так:

```
int main(){
     WORD wVersionRequested;
     WSADATA wsaData;
     wVersionRequested=MAKEWORD(2,2);
     WSAStartup(wVersionRequested,&wsaData);
```

В данной работе нас интересуют сокеты потоков (`SOCK_STREAM`), которые позволяют гарантировать бесперебойную доставку данных в нужном порядке и без дублирования. В *TCP/IP*-реализации *WinSock* сокеты потоков ис-

пользуют протокол *TCP* (*Transmission Protocol*). Сокеты потоков обеспечивают пересылку 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 *WSAGetLastError ( )*.

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 пе- ременную 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 – не 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.

1 024 – 4 999 – служат для портов клиентов. Обычно сокеты приложений-клиентов используют номера портов именно из этого диапазона.

5 000 – 65 535. 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.

В 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 задерж-

ки. 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 X 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.

ЛАБОРАТОРНАЯ РАБОТА №2 Создание последовательного сервера без установлениялогического соединения UDP

Цель работы: изучить методы создания серверов без установления логического соединения *UDP*, используя алгоритм последовательной обработки запросов.

2.1 Протокол UDP

Задачей протокола транспортного уровня *UDP (User Datagram Protocol)* является передача данных между прикладными процессами без гарантий доставки, поэтому его пакеты могут быть потеряны, продублированы или прийти не в том порядке, в котором они были отправлены.

Каждый коммуникационный протокол оперирует с некоторой единицей передаваемых данных. Единицу данных протокола *UDP* называют *дейтаграммой (datagram)*. Протокол *UDP* является простейшим дейтаграммным протоколом, который используется в том случае, когда задача надежного обмена данными либо вообще не ставится, либо решается средствами более высокого уровня – системными прикладными службами или пользовательскими приложениями.

В стеке протоколов *TCP/IP* протокол *UDP* обеспечивает основной механизм, используемый прикладными программами для передачи дейтаграмм другим приложениям. *UDP* предоставляет протокольные порты, используемые для различения нескольких процессов, выполняющихся на одном компьютере. Помимо посылаемых данных каждое *UDP*-сообщение содержит номер порта-приемника и номер порта-отправителя, делая возможным для программ *UDP* на машине-получателе доставку сообщения соответствующему реципиенту, а для получателя – посылку ответа соответствующему отправителю. Этот протокол обеспечивает ненадежную доставку данных «по возможности» в отличие от протокола *TCP*, обеспечивающего гарантированную доставку. *UDP* не использует подтверждения прихода сообщений, не упорядочивает приходящие сообщения и не обеспечивает обратной связи для управления скоростью передачи информации между машинами. Поэтому *UDP*-сообщения могут быть потеряны, размножены или приходить не по порядку. Кроме того, пакеты могут приходить раньше, чем получатель сможет обработать их.

2.2 Сокеты дейтаграмм

Сокеты дейтаграмм (datagram sockets) – это средства поддержки не очень надежного обмена пакетами. Ненадежность в данном контексте означает отсутствие гарантии их доставки по назначению в требуемом порядке. Фактически один и тот же пакет дейтаграммы может быть доставлен несколько раз.

Хотя *WinSock* поддерживает и другие протоколы, во многих случаях, на-

пример, при обмене данными между двумя процессами на одном и том же компьютере или между двумя компьютерами в локальной сети с небольшой нагрузкой, исключена путаница, неправильное название пакетов или доставка их не по адресу. Однако полной гарантии от подобных неприятностей приложение не обеспечивает.

Если же приложение управляет обменом данными по более сложной и загруженной сети, ненадежный характер сокетов дейтаграмм проявится очень быстро. При отсутствии в приложении обработки ошибок оно просто не справится с задачей. Тем не менее сокеты дейтаграмм очень полезны для пересылки данных, состоящих из отдельных пакетов или записей. Они также являются удобным средством рассылки циркулярных пакетов по нескольким адресам одновременно.

Примерами сетевых приложений, использующих *UDP*, являются *NFS* (*Network File System* – сетевая файловая система) и *SNMP* (*Simple Network Management Protocol* – простой протокол управления сетью).

2.3 Алгоритм работы последовательного сервера без установления логического соединения

Опишем обобщенный алгоритм работы последовательного сервера без установления логического соединения:

1. Создать сокет сервера (связав его с доступным портом и локальным адресом).

2. Считывать в цикле запросы от клиента, формировать ответы и отправлять клиенту в соответствии с прикладным протоколом.

На рисунке 3 показана схема организации работы последовательного сервера без установления логического соединения. Требуется только один поток выполнения, который обеспечивает взаимодействие сервера со многими клиентами с использованием одного сокета.

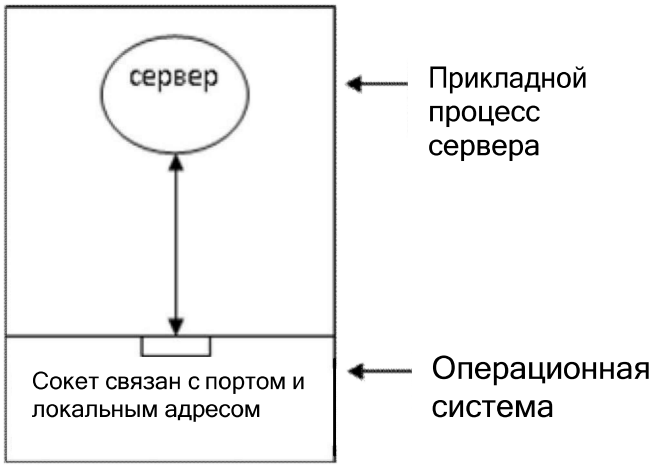

Рисунок 3 – Схема организации работы последовательного сервера без установления логического соединения

2.4 Методические указания по созданию последовательного сервера без установления логического соединения UDP

Для того чтобы продемонстрировать особенности работы протокола *UDP,* рассмотрим следующую задачу.

Осуществить взаимодействие клиента и сервера на основе протокола *UDP*. Функциональные возможности клиента реализовать следующим образом: клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Функциональные возможности сервера реализовать следующим образом: сервер, получив эту строку, должен поменять в ней местами символы на четных и нечетных позициях. Результат возвратить назад клиенту.

Так же как и в предыдущей лабораторной работе, создадим два проекта – клиент и сервер.

Серверная часть

```
#include<winsock2.h>
#include<iostream.h>
#include<stdlib.h>
#include<process.h>

void main(void){
     WORD wVersionRequested;
     WSADATA wsaData;
     int err;
     wVersionRequested=MAKEWORD(2,2);
     err = WSAStartup(wVersionRequested,&wsaData);
```

Вызов функции *socket()* выглядит следующим образом:

```
     SOCKET s;
     s = socket(AF_INET,SOCK_DGRAM,0);
```

Так как в данном случае создается сокет дейтаграмм, использующий протокол *UDP*, вторым аргументом, задающим тип создаваемого сокета, должен быть *SOCK_DGRAM.*

```
     struct sockaddr_in ad;
     ad.sin_port = htons(1024);
     ad.sin_family = AF_INET;
     ad.sin_addr.s_addr = 0;//подставляет подходящий ip
     bind(s,(struct sockaddr*) &ad,sizeof(ad));

     char b[200],tmp='\0';
          int l;
     l = sizeof(ad);
```

После создания сокета в приложении-сервере и привязки его к определенному адресу и порту можно принимать данные от приложения-клиента. Это делается с помощью функции *recvfrom()*, имеющей следующий прототип:

```
int recvfrom(SOCKET s,char FAR * buf,int len,int flags,
struct sockaddr FAR *from,int FAR *fromlen);
```

Первый параметр – это дескриптор сокета, возвращаемый функцией *socket()*; за ним идут указатель на буфер для приема новой дейтаграммы и длина буфера. Параметр *flags* может быть равен *msg_peek* для заполнения буфера таким образом, что дейтаграмма остается во входной очереди. Последние два параметра используются для получения адреса сокета, пославшего дейтаграмму. По этому адресу можно послать ответ.

В нашем примере функция *recvfrom()* примет следующий вид (параметр *flags* устанавливается как 0):

```
int rv = recvfrom(s, b, lstrlen(b), 0, (STRUCT SOCKADDR*)
&ad, &l);
```

При успешном считывании дейтаграммы функция *recvfrom()* возвращает количество прочитанных байтов. При ошибочном приеме дейтаграммы возвращается значение *socket_error*.

Если размер буфера, переданный в функцию *recvfrom()*, слишком мал для приема всей дейтаграммы целиком, буфер заполняется теми данными, которые в него помещаются, а оставшаяся часть дейтаграммы сбрасывается в подсистему сборки мусора и пропадает безвозвратно. В этом случае функция *recvfrom()* возвращает значение *socket error*.

```
b[rv]='\0';
cout<<b<<endl;

for (unsigned i=0;b[i];i++)
     if (i%2==0)
          if (b[i+1]!='\0'){

               tmp=b[i];
               b[i]=b[i+1];
               b[i+1]=tmp;
          }
```

Отправка данных выполняется функцией *sendto()*, имеющей следующий прототип:

```
int sendto (SOCKET s, const char FAR *buf, int len, int
```

```
flags, const struct sockaddr FAR *to, int tolen) ;
```

Первый параметр, как и раньше, – это дескриптор сокета; за ним идет указатель на буфер, содержащий пересылаемые данные, и длина этого буфера. Последние два параметра используются для указания адреса и порта сокета назначения.

Если функция *sendto()* срабатывает корректно, она возвращает количество посланных байтов, которое может отличаться от значения параметра len. В случае ошибки функция *sendto()* возвращает *socket_error*.

При попытке послать дейтаграмму большего размера, чем максимально допустимый в данной реализации *WinSock*, код ошибки будет равен *wsaemsgsize*. Максимально допустимый размер дейтаграммы можно определить путем вызова функции *WSAStartup()*.

```
        sendto(s, b, lstrlen(b), 0, (STRUCT SOCKADDR*) &ad, l);

    closesocket(s);

    WSACleanup();
}
```

Клиентская часть

Ниже приведена программа клиента.

```
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>
#include <iostream.h>

int main(void){
    char buf[100], b[100];
    WORD wVersionRequested;
    WSADATA wsaData;
    int err;
    wVersionRequested = MAKEWORD(2,2);
    err=WSAStartup(wVersionRequested, &wsaData);
    if(err != 0){return 0;}

    SOCKET s;
    s = socket(AF_INET,SOCK_DGRAM,0);
    sockaddr_in add;
    add.sin_family = AF_INET;
    add.sin_port = htons(1024);
```

В следующей строке происходит явное указание *IP*-адреса при помощи строкового представления (точечной нотации) *inet_addr*, возвращающего число в формате поля *s_addr* .

```
        add.sin_addr.s_addr = inet_addr("127.0.0.1");


        int t;
        t = sizeof(add);
        cout<<"Enter the string, please"<<endl;
        cin.getline(buf,100,'\n');

        sendto(s, buf, lstrlen(buf), 0, (struct sockaddr*) &add,
    t);

        int rv = recvfrom(s, b, lstrlen(b), 0, (struct sockaddr*)
&add, &t);
        b[rv] = '\0';
        cout<<b<<endl;

        closesocket(s);

        WSACleanup();
        return 0;
      }
```

2.5 Индивидуальные задания

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

1. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна 3, то удаляются все числа, которые делятся на 3. Результаты преобразований этой строки возвращаются назад клиенту.

2. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина четная, то удаляются 3 первых и 2 последних символа. Результаты преобразований этой строки возвращаются назад клиенту.

3. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен выяснить, имеются ли среди символов этой строки все буквы, входящие в слово WINDOWS. Количество вхождений символов в строку передать назад клиенту.

4. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина не-

четная, то удаляется символ, стоящий посередине строки. Преобразованная строка передается назад клиенту.

5. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен заменить в этой строке каждый второй символ @ на #. Результаты преобразований передаются назад клиенту.

6. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен заменить в этой строке символов все пробелы на символ *. Преобразованная строка передается назад клиенту.

7. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина больше 15, то из нее удаляются все цифры. Клиент получает преобразованную строку и количество удаленных цифр.

8. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина кратна 4, то удаляются все числа, делящиеся на 4. Клиент получает преобразованную строку и количество таких чисел.

9. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна 5, то подсчитывается количество скобок всех видов. Их количество посылается клиенту.

10. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если эта длина кратна 4, то первая часть строки меняется местами со второй. Результаты преобразований возвращаются назад клиенту.

11. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если значение этой длины равно 10, то удаляются все символы от A до Z. Результаты преобразований такой строки и количество удалений возвращаются назад клиенту.

12. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина больше 15, то удаляются все символы от a до z. Преобразованная строка и количество удаленных символов возвращаются назад клиенту.

13. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен в полученной строке символов поменять местами символы на четных и нечетных позициях. Полученную строку возвратить назад клиенту.

14. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки, и, если длина больше 7, то выделяется подстрока в { } скобках и возвращается назад клиенту.

15. Клиент вводит с клавиатуры строку символов и посылает ее серверу. Признак окончания ввода строки – нажатие клавиши «Ввод». Сервер, получив эту строку, должен определить длину введенной строки и, если длина больше 15, то выделяется подстрока до первого пробела и возвращается назад клиенту.

2.6 Контрольные вопросы

1. Что содержит *UDP*-сообщение помимо посылаемых данных?
2. Что называется дейтаграммой?
3. Какие возможности не предоставляет *UDP* (в отличие от *TCP*)?
4. Чем функции *sendto()* и *recvfrom()* отличаются от функций *send()* *recv()*?
5. Что происходит, если размер буфера, переданный в функцию *recvfrom*(), слишком мал для приема всей дейтаграммы целиком?
6. Приведите примеры сетевых приложений, использующих *UDP*.
7. В каких случаях применение *UDP*-протокола может быть предпочтительней, чем *TCP*?