Глава 1 . Взаимодействие процессов распределенного приложения

1.1. Предисловие к главе

Когда рассматривают принципы взаимодействия различных частей (процессов) распределенного приложения, то говорят о ***модели взаимодействия***, а когда рассматривают распределение ролей между различными частями (процессами) распределенного приложения, то говорят об ***архитектуре распределенного приложения***.

Для обсуждения принципов взаимодействия процессов распределенного приложения, как правило, применяется модель ***ISO/OSI*** (International Standards Organization/Open System Interconnection reference model), которая была разработана в 1980-х годах и регулируется стандартом ISO 7498. Официальное название модели ISO/OSI – ***сетевая эталонная модель взаимодействия открытых систем Международной организации по стандартизации***. Спецификации ISO/OSI используются производителями аппаратного и программного обеспечений

Наиболее популярной архитектурой для распределенного программного приложения является ***архитектура клиент-сервер***. Будем говорить, что распределенное приложение имеет архитектуру клиент-сервер, если все процессы распределенного приложения можно условно разбить на две группы. Одна группа процессов называется серверами другая – клиентами. Обмен данными осуществляется только между процессами-клиентами и процессами-серверами. Основное отличие процесса-клиента от процесса-сервера в том, что инициатором обмена данными всегда является процесс-клиент. Другими словами процесс-клиент обращается за услугой (сервисом) к процессу-серверу. Такая архитектура лежит в основе большинства современных информационных систем [1,2,3].

В этой главе рассматривается модель ISO/OSI и особенности архитектуры клиент-сервер.

1.2. Модель взаимодействия открытых систем

Функции, обеспечивающие взаимодействие открытых систем в модели ISO/OSI распределены по следующим семи уровням: 1) физический; 2) канальный; 3) сетевой; 4) транспортный; 5) сеансовый; 6) представительский; 7) прикладной. Задача каждого уровня – предоставление услуг вышестоящему уровню таким образом, чтобы детали реализации этих услуг были скрыты. Наборы правил и соглашений, описывающих процедуры взаимодействия каждого уровня модели с соседними уровнями называются ***протоколами***.

Опишем кратко назначение всех уровней модели OSI.

Физический уровень. Физический уровень определяет свойства среды передачи данных (коаксиальный кабель, витая пара, оптоволоконный канал и т.п.) и способы ее соединения с сетевыми адаптерами: технические характеристики кабелей (сопротивление, емкость, изоляция и т.д.), перечень допустимых разъемов, способы обработки сигнала и т.п.

Канальный уровень. На канальном уровне модели рассматривается два подуровня: подуровень управления доступом к среде передачи данных и подуровень управления логическим каналом. Управление доступом к среде передачи данных определяет методы совместного использования сетевыми адаптерами среды передачи данных. Подуровень управления логической связью определяет понятия канала между двумя сетевыми адаптерами, а также способы обнаружения и исправления ошибок передачи данных. Основное назначение процедур канального уровня подготовить блок данных (обычно называемый кадром) для следующего сетевого уровня.

Здесь следует отметить два момента: 1) начиная с подуровня управления логической связью и выше протоколы никак не зависят от среды передачи данных; 2) для организации локальной сети достаточно только физического и канального уровней, но такая сеть не будет масштабируемой (не сможет расширяться), т.к. имеет ограниченные возможности адресации и не имеет функций маршрутизации.

Сетевой уровень. Сетевой уровень определяет методы адресации и маршрутизации компьютеров в сети. В отличие от канального уровня сетевой уровень определяет единый метод адресации для всех компьютеров в сети не зависимого от способа передачи данных. На этом уровне определяются способы соединения компьютерных сетей. Результатом процедур сетевого уровня является пакет, который обрабатывается процедурами транспортного уровня.

Транспортный уровень. Основным назначением процедур транспортного уровня является подготовка и доставка пакетов данных между конечными точками без ошибок и в правильной последовательности. Процедуры транспортного уровня формируют файлы для сеансового уровня из пакетов, полученных от сетевого уровня.

Сеансовый уровень. Сеансовый уровень определят способы установки и разрыва соединений (называемых сеансами) двух приложений, работающих в сети.

Следует отметить, что сеансовый уровень - это точка взаимодействия программ и компьютерной сети.

Представительский уровень. На представительский уровне определяется формат данных, используемых приложениями. Процедуры этого уровня описывают способы шифрования, сжатия и преобразования наборов символов данных.

Прикладной уровень. Основное назначения уровня: определить способы взаимодействия пользователей с системой (определить интерфейс).

На рис. 1.2.1 изображена схема взаимодействия двух систем с точки зрения модели OSI. Толстой линией со стрелками на концах обозначается движение данных между системами. Данные проходят от прикладного уровня одной системы до прикладного уровня другой через все нижние уровни системы. Причем по мере своего движения от отправителя к получателю (из одной системы в другую) на каждом уровне данные подвергаются необходимому преобразованию (в соответствии с

протоколами модели): при движении от прикладного уровня к физическому данные преобразовываются в формат, позволяющий передать данные по физическому каналу; при движении от физического уровня до прикладного происходит обратное преобразование данных. При такой организации обмена данными фактически взаимодействие осуществляется между одноименными уровнями (на рисунке это взаимодействие обозначено линиями со стрелками между уровнями двух систем).

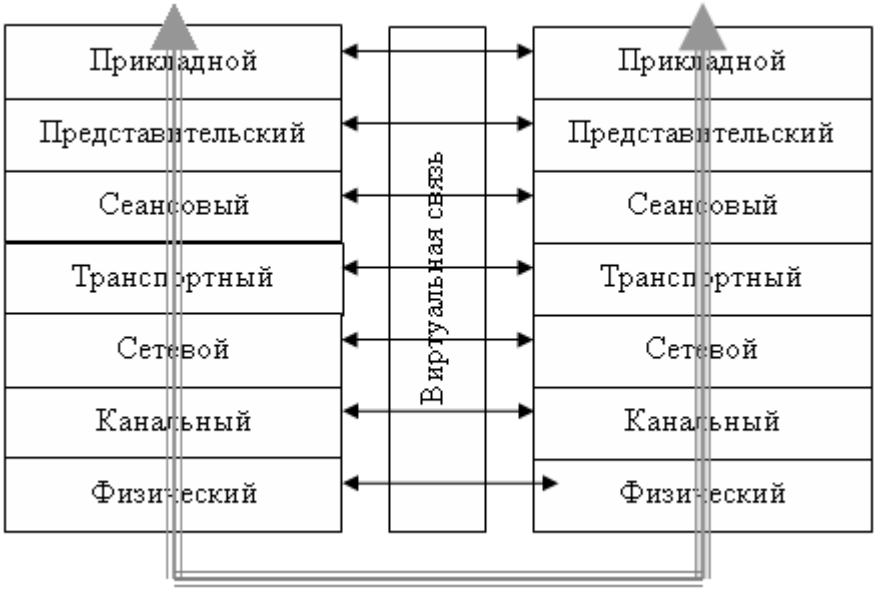

Рисунок 1.2.1. Схема взаимодействия открытых систем

Выше уже отмечалось, что точкой взаимодействия программ и компьютерной сети является сеансовый уровень. Рассмотрим этот момент более подробно для распределенного приложения состоящего из двух взаимодействующих процессов.

На рис. 1.2.2 изображены два процесса (с именами C и S), функционирующие на разных компьютерах в среде соответствующих операционных систем. В составе операционных систем имеются службы (специальные программы) обеспечивающие поддержку протоколов канального, сетевого и транспортного уровней. Протоколы физического уровня, как правило, обеспечиваются сетевыми адаптерами. На рисунке граница операционной системы условно проходит по канальному уровню. Действительно, часто (но не всегда) часть процедур канального уровня (обычно подуровня управления доступом к среде) обеспечивается аппаратно (сетевым адаптером), а другая часть процедур (обычно подуровня управления логическим каналом) реализована в виде драйвера, инсталлированного в состав операционной системы. Процессы, взаимодействуют со службами, обеспечивающими процедуры протоколов транспортного уровня с помощью набора специальных функций API (Application Program Interface), входящими в состав операционной системы. Следует отметить, что рисунок 1.2.2 носит чисто схематический характер и

служит, только для объяснения принципа взаимодействия процессов в распределенном приложении. В каждом конкретном случае распределение протокольных процедур разное и зависит от архитектуры компьютера, степени интеллектуальности сетевого адаптера, типа операционной системы и т.п. Заметим также, что функции сеансового, представительского и прикладного уровней обеспечивается самим распределенных приложением.

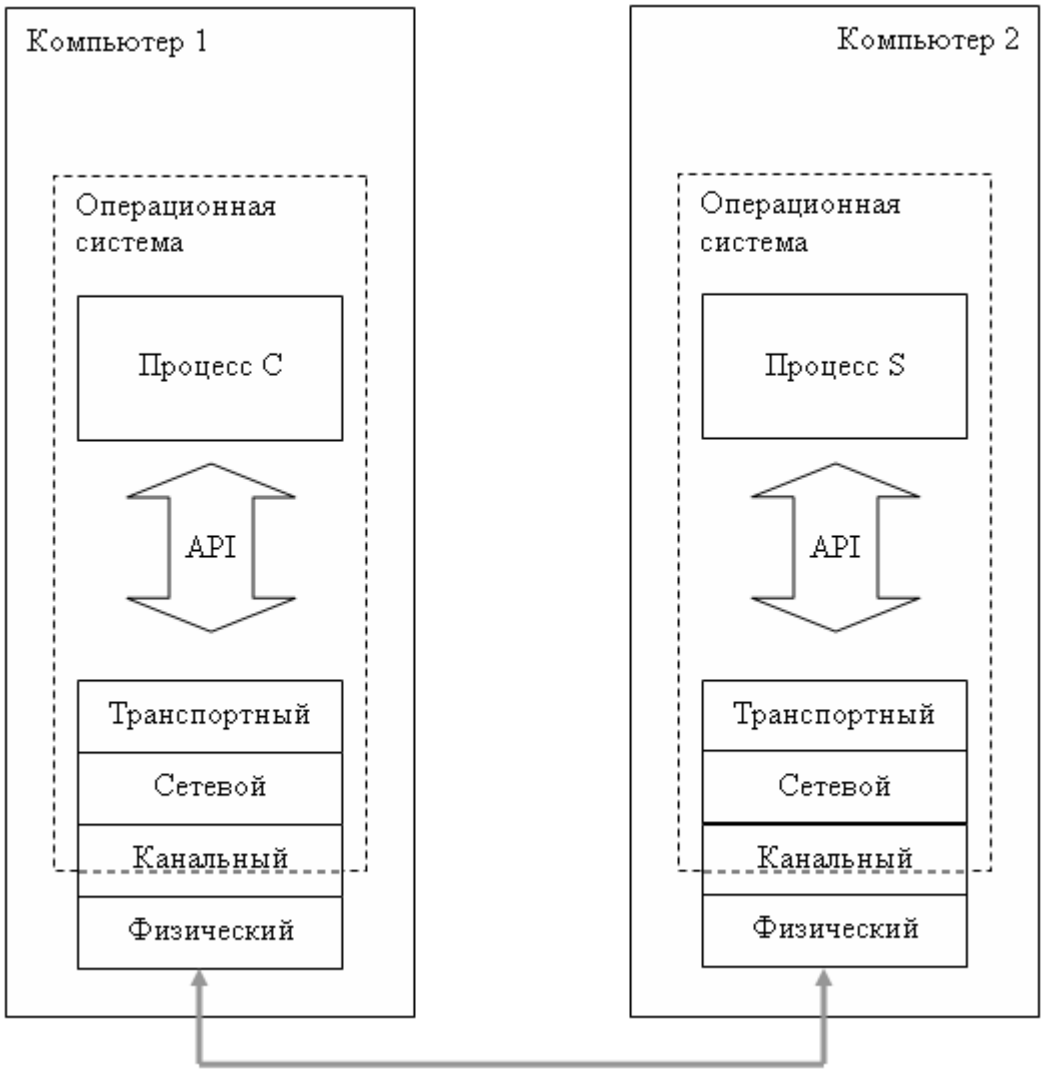

Рисунок 1.2.2. Схема взаимодействия процессов в распределенном приложении

1.3. Архитектура клиент-сервер

Распределенное приложение, имеющее архитектуру клиент-сервер, подразумевает наличие в своем составе два вида процессов: процессы-серверы и процессы-клиенты. Далее эти процессы будем назвать просто серверами и клиентами.

Следует отметить, что приведенное в предисловии определение архитектуры клиент-сервер несколько упрощено. Дело в том, что некоторые процессы распределенного приложения могут выступать клиентом для некоторых процессов-серверов и одновременно являться серверами других процессов-клиентов.

Инициатором обмена данными между клиентом и сервером всегда является клиент. Для этого клиент должен обладать информацией о месте нахождения сервера или иметь механизмы для его обнаружения. Способы связи между клиентом и сервером могут быть различными и в общем случае зависят от интерфейсов, поддерживаемых операционной средой, в которой работает распределенное приложение. Клиент должен быть тоже распознан сервером, для того, чтобы сервер, во-первых, мог его отличить от других клиентов, а во-вторых, чтобы смог обмениваться с клиентом данными. Если основная вычислительная нагрузка ложится на сервер, а клиент лишь обеспечивает интерфейсом пользователя с сервером, то такой клиент часто называют ***тонким***.

По методу облуживания серверы подразделяются на ***итеративные*** и ***параллельные*** серверы ***(iterative and concurrent servers)***. Принципиальная разница заключается в том, что параллельный сервер предназначен для обслуживания нескольких клиентов одновременно и поэтому использует специальные средства операционной системы позволяющие распараллеливать обработку нескольких клиентских запросов. Итеративный сервер, как правило, обслуживает запросы клиентов поочередно, заставляя клиентов ожидать своей очереди на обслуживание, или просто отказывает клиенту обслуживании. По всей видимости, можно говорить о итеративно-параллельных серверах, когда сервер имеет ограниченные возможности по распараллеливанию своей работы. В этом случае только часть клиентских запросов будет обслуживаться параллельно.

1.3. Итоги главы

1. Принцип взаимодействия процессов распределенного приложения следует рассматривать в рамках модели ISO/OSI.
2. Как правило, при разработке распределенного приложения, разработчику нет необходимости вникать в детали обмена данными между процессами этого приложения. В основном программист руководствуется API, предоставленный операционной системой для взаимодействия со специальными программами, обеспечивающими выполнение процедур протокола транспортного уровня.
3. Изменение в конфигурации компьютерной сети не приведут к необходимости перепрограммирования приложения, если эти изменения не касаются протокола транспортного уровня.
4. Процессы распределенного приложения могут работать на компьютерах разной архитектуры, в разных операционных средах (на разных платформах). В общем случае разработчику приходится

программировать приложение (или его части) для каждой конкретной платформы.

5. Архитектура клиент-сервер лежит в основе большинства современных информационных систем. Для разработки параллельного сервера, необходимо, чтобы операционная система предоставляла возможность распараллеливания процессов.
6. При разработке распределенных приложений могут быть использованы готовые решения: серверы систем управления базами данных (например, Microsoft SQL Server, Oracle Server), серверы приложений (Citrix Application Server), Web-серверы (Apache, Microsoft IIS, Apache Tomcat), браузеры (Microsoft Internet Explorer, Opera, Netscape Navigator) и т.д. Применение готовых решений значительно упрощает разработку распределенных приложений.

Глава 2 . Стек протоколов TCP/IP

2.1. Предисловие к главе

Семейство протоколов TCP/IP, часто именуемое ***стеком TCP/IP***, стало промышленным стандартом де-факто для обмена данными между процессами распределенного приложения и поддерживается всеми без исключения операционными системами общего назначения. Обширная коллекция сетевых протоколов и служб, называемая стеком TCP/IP намного больше, чем сочетание двух основных протоколов давшее ей имя. Тем не менее, эти протоколы являются основой стека TCP/IP: ***TCP (Transmission Control Protocol)*** обеспечивает надежную доставку данных в сети; ***IP (Internet Protocol)*** организует маршрутизацию сетевых передач от отправителя к получателю и отвечает за адресацию сетей и компьютеров.

В настоящее время существует шесть групп стандартизации координирующих TCP/IP: ISOC (Internet Society, http://www.isoc.org), IAB (Internet Architecture Board, http://www.iab.org), IETF (Internet Engineering Task Force, http://www.ietf.org), IRTF (Internet Research Task Force, http://www.irtf.org), ISTF (Internet Societal Task Force, http://www.istf.org), ICANN (Internet Corporation for Assigned Names and Numbers, http://www.icann.org). Наиболее важной организацией из перечисленных является IETF – проблемная группа по проектированию Internet, поскольку именно она занимается поддержкой документов именуемых ***RFC (Request for Comments)***, в которых описаны все правила и форматы всех протоколов и служб TCP/IP в сети Internet.

Всем RFC присвоены номера. Например, специальный документ “Официальные стандарты протоколов сети Internet” имеет номер RFC 2700. Другой важный документ RFC 2026 определяет порядок создания самого документа RFC и процедуры, которые должны быть им пройдены для превращения в официальный стандарт группы IETF. Если есть документы, имеющее одно название, но разные номера, то документ с самым большим номером считается текущей версией.

Более подробно с историей создания TCP/IP, с назначением групп стандартизации и процедурами создания и утверждения документов RFC можно ознакомиться в [5,6].

В этой главе рассматриваются основные компоненты стека протоколов TCP/IP, необходимые для построения распределенного приложения.

2.2. Структура TCP/IP

Так как архитектура TCP/IP была разработана задолго до модели ISO/OSI, то неудивительно, что конструкция TCP/IP несколько отличается от эталонной модели. На рисунке 2.2.1. изображены уровни обеих моделей и устанавливается их соответствие. Уровни моделей похожи, но не идентичны. Структура TCP/IP является более простой: в ней не выделяются Физический, Канальный, Сетевой и Представительский уровни. В общем и целом Транспортные уровни обеих моделей соответствуют друг другу, но есть и

некоторые различия. Например, некоторые функции Сеансового уровня модели ISO/OSI берет на себя Транспортный уровень TCP/IP. Содержимое Сетевого уровня модели ISO/OSI тоже примерно соответствует Межсетевому уровню TCP/IP. В большей или меньшей степени Прикладной уровень TCP/IP соответствует трем уровням Сетевому, Представительскому и Прикладному модели ISO/OSI, а Уровень доступа к сети – совокупности Физического и Канального уровней.

Рисунок 2.2.1. Уровни моделей ISO/OSI и TCP/IP

В дальнейшем при описании протоколов стека TCP/IP будет использоваться и названия уровней модели ISO/OSI, если это помогает определить более точное место протокола в иерархии.

2.3. Протоколы Уровня доступа к сети

На Уровне доступа к сети задействованы протоколы для создания локальных сетей (***LAN***, Local-Area Networks) и для соединения с глобальными сетями (***WAN***, Wide-Area Networks). Работа протоколов этого уровня регулируются семейством стандартов ***IEEE 802*** (Institute of Electrical

and Electronic Engineers), включающее помимо прочих компоненты: IEEE 802.1 по межсетевому обмену; IEEE 802.2 по управлению логическим соединением (***LLC***, Logical Link Control); IEEE 802.3 по управлению доступом к среде (***MAC***, Media Access Control); IEEE 802.4 по множественному доступу с контролем несущей и обнаружением конфликтов (***CSMA/CD***, Carrier Sense Multiple Access with Collision Detection). Одной из основных характеристик протоколов канального уровня является ***максимальная единица передачи данных MTU*** (Maximum Transmission Unit), которая определяет максимальную длину в байтах данных передаваемых в одном кадре. От значения MTU зависит скорость передачи по каналу. Если IP-модулю требуется отправить данные (дейтаграмму) имеющие дину большую MTU, то он производит фрагментацию разбивая дейтаграмму на части имеющие длину меньшую, чем MTU. В таблице 2.3.1 приведены типичные значения MTU для некоторых сетей.

Таблица 2.3.1

Сеть	**MTU (байты)**
FDDI	4464
Ethernet	1500
IEEE802.3/802.2	1492
X.25	576
SLIP, PPP (с минимальной задержкой)	256

Протокол Ethernet. Термин Ethernet обычно связывают со стандартом опубликованным в 1982 г. совместно корпорациями DEC, Intel и Xerox (DIX). На сегодняшний день это наиболее распространенная технология локальных сетей. Ethernet применяет метод доступа CSMA/CD, использует 48-битную адресацию (стандарт IEE EUI-64) и обеспечивает передачу данных до 1 гигабита в секунду. Максимальная длина кадра, передаваемая в сети Ethernet составляет 1518 байт, при этом сами данные могут занимать от 46 до 1500 байт. Физически функции протокола Ethernet реализуются сетевой картой (***NIC***, Network Interface Card), которая может быть подключена к кабельной системе с фиксированным MAC-адресом производителя (в соответствии со стандартом ICANN).

Протокол SLIP(Serial Line IP). Аббревиатурой SLIP обозначают межсетевой протокол для последовательного канала. Раньше SLIP использовался для подключения домашних компьютеров к Internet через последовательный порт RS-232. Протокол использует простейшую инкапсуляцию кадра и имеет ряд недостатков: хост с одной стороны должен знать IP-адрес другого, т.к. SLIP не дает возможности сообщить свой IP-адрес; если линия задействована SLIP, то она не может быть использована никаким другим протоколом; SLIP не добавляет контрольной информации к пакету передаваемой информации – весь контроль возложен на протоколы

более высокого уровня. Ряд недостатков были исправлены в новой версии протокола именуемой CSLIP (Compressed SLIP).

Протокол PPP (Point-to-Point Protocol). PPP – универсальный протокол двухточечного соединения: поддерживается TCP/IP, NetBEUI, IPX/SPX, DECNet и многими стеками протоколов. Протокол может применяться для технологии ISDN (Integrated Services Digital Network) и SONET (Synchronous Optical Network). PPP поддерживает многоканальные реализации: можно сгруппировать несколько каналов с одинаковой пропускной способностью между отправителем и получателем. Кроме того, PPP обеспечивает циклический контроль для каждого кадра, динамическое определение адресов, управление каналом. В настоящее время это наиболее широко используемый протокол для последовательного канала, обеспечивающий соединение компьютера с сетью Internet и практически вытеснил протокол SLIP.

2.4. Протоколы Межсетевого уровня

Важнейшими протоколами Межсетевого уровня TCP/IP являются: ***IP*** (Internet Protocol), ***ICMP*** (Internet Control Message Protocol), ***ARP*** (Address Resolution Protocol), ***RARP*** (Reverse ARP).

Протокол IP. В семействе протоколов TCP/IP протоколу IP отведена центральная роль. Его основной задачей является доставка ***дейтограмм*** (так называется единица передачи данных в терминологии IP). При этом протокол по определению является ***ненадежным*** и ***не поддерживающим соединения***.

Ненадежность протокола IP обусловлена тем, что нет гарантии, что посланная узлом сети дейтаграмма дойдет до места назначения. Сбой, произошедший на любом промежуточном узле сети, может привести к уничтожению дейтаграмм. Предполагается, что необходимая степень надежности должна обеспечиваться протоколами верхних уровней.

IP не ведет никакого учета очередности доставки дейтаграмм: каждая дейтаграмма обрабатывается независимо от остальных. Поэтому очередность доставки может нарушаться. Предполагается, что учетом очередности дейтаграмм должен заниматься протокол верхнего уровня.

Если протоколы Уровня доступа к сети при передаче данных используют MAC-адреса, то на Межсетевом уровне применяется IP-адресация. Главной особенностью IP-адреса является его независимость от физической устройства, подключенного к сети. Это дает возможность на уровне IP одинаковым образом обрабатывать данные, полученные или отправленные с помощью модема, сетевой карты или любого другого устройства, поддерживающего интерфейс протокола IP. Все устройства, имеющие IP-адрес, в терминологии протокола IP называются ***хостами*** (host).

IP-адрес представляет собой последовательность из 32 битов. Причем старшие (левые) биты этой последовательности отводятся для адреса сети, а младшие (правые) – для адреса хоста в этой сети. Для записи IP-адреса, как

правило, используются четыре десятичных числа, разделенных точкой. Каждое десятичное число является десятичным представлением 8 битов (***октет*** в терминологии TCP/IP) адреса. На рисунке 2.4.1 разобран пример перевода IP-адреса из двоичного формата в десятичный.

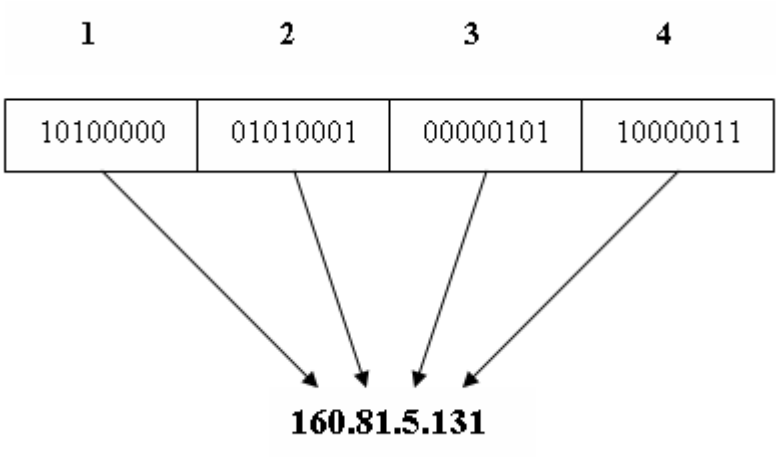

Рисунок 2.4.1. Представление IP-адреса в десятичном формате

Количество бит отведенных для адреса сети и адреса хоста определяется ***моделью адресации***. Существует две модели адресации: ***классовая*** и ***бесклассовая***. В классовой модели адресации все адреса подразделяются на пять классов: A, B, C, D, E. Принадлежность к классу определяется старшими битами адреса. На рисунке 2.4.2 приведены форматы адресов для всех классов.

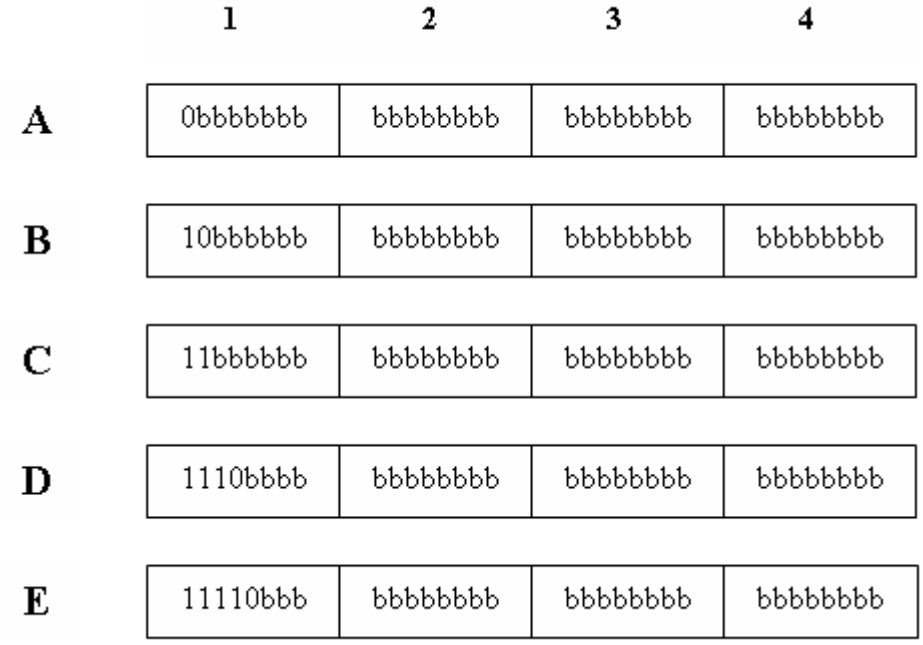

Рисунок 2.4.2 Форматы IP-адресов в классовой модели адресации

Классы D и E имеют специальное назначение: D – предназначен для использования групповых адресов, позволяющих отправлять сообщения группе хостов; E – исключительно для экспериментального применения. Более подробно о применении адресов классов D и E можно ознакомиться в [5,6]. Распределение октетов IP-адреса на адрес сети и адрес хоста для классов A, B и C приводится на рисунке 2.4.3. Закрашенные октеты обозначают часть IP-адреса отведенную для адреса сети, не закрашенные – часть адреса, используемую для адресов хостов. Правый столбец показывает формат адресов в десятичном виде: символом n обозначается сетевая часть адреса, символом h – часть адреса для идентификации хоста.

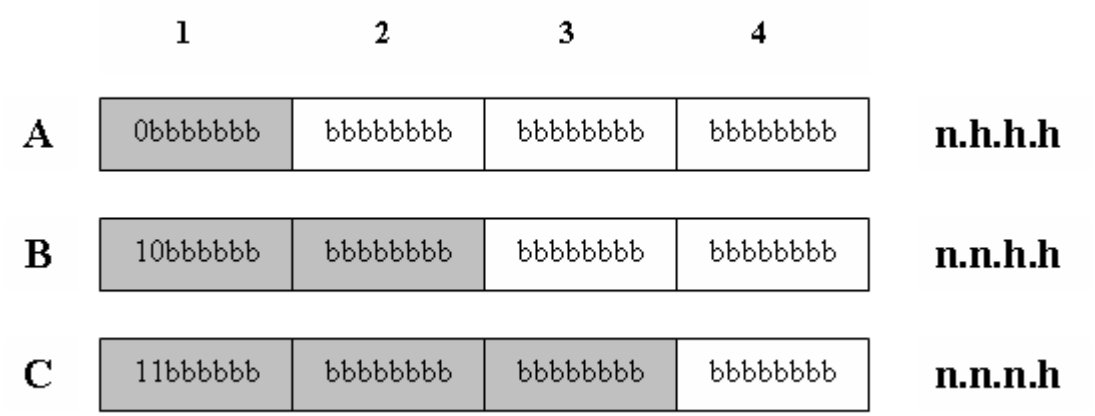

Рисунок 2.4.3. Распределение разрядов IP-адреса на адрес сети и адрес хоста для классовой модели IP-адресов

Таблица 2.4.1

Класс	**Диапазон адресов**	**Диапазон частных адресов**
A	0.0.0.0 – 127.255.255.255	10.0.0.0 – 10.255.255.255
B	128.0.0.0 – 191.255.255.255	172.16.0.0 – 172.31.255.255
C	192.0.0.0 – 223.255.255.255	192.168.0.0 – 192.168.255.255
D	224.0.0.0 – 239.255.255.255	не предусмотрен
E	240.0.0.0 – 247.255.255.255	не предусмотрен

Для каждого класса адресов зарезервирован диапазон, используемый для частного применения (это оговаривается в документе RFC 1918). Эти адреса предназначены для неконтролируемого использования в организациях. Уникальность этих адресов в сети Internet не гарантируется и поэтому их маршрутизация в сети Internet не возможна. Кроме того, адреса вида 127.n.n.n предназначены для выполнения возвратного тестирования (***loopback testing***). Существует некоторая путаница с применением адресов состоящих из всех нулей или единиц. До выхода документа RFC 1878 (1995 г.) не разрешалось использование в качестве адреса хоста нулевую последовательность битов (этот адрес считался адресом самой сети) и

последовательность битов, состоящую из одних единиц (этот адрес использовался для широковещательных сообщений в сети). RFC 1878 разрешает использование таких адресов. Диапазоны IP- адресов для каждого класса сведены в таблице 2.4.1.

Применение классовой модели адресации не всегда удобно. Альтернативой классовой модели является ***бесклассовая междоменная маршрутизация*** – ***CIDR*** (Classless Inter-Domain Routing). CIDR позволяет произвольным образом назначать границу сетевой и хостовой части IP-адреса. Для этого каждому IP-адресу прилагается 32-битовая маска, которую часто называют ***маской сети*** (net mask) или ***маской подсети*** (subnet mask). Сетевая маска конструируется по следующему правилу: на позициях, соответствующих адресу сети, биты установлены; на позициях, соответствующих адресу хоста, биты сброшены. Установка маски подсети осуществляется при настройке протоколов TCP/IP на компьютере. Вычисление адреса сети выполняется с помощью операции конъюнкции между IP-адресом и маской подсети. На рисунке 2.4.4. разобран пример вычисления адреса сети с помощью маски подсети.

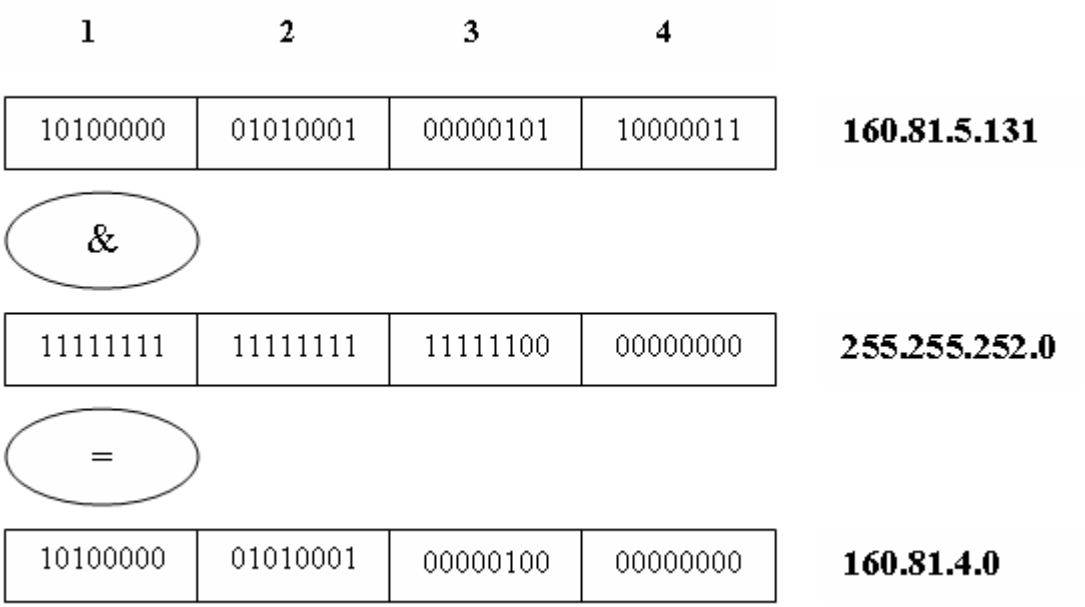

Рисунок 2.4.4. Вычисление адреса сети с помощью маски сети

Протокол IP обеспечивает доставку дейтаграмм в пределах всей ***составной*** IP-сети. Составной IP-сетью называются объединение нескольких IP-сетей с помощью специальных устройств, называемых ***шлюзами***. Обычно шлюз представляет собой компьютер, на котором установлены несколько интерфейсов IP и специальное программное обеспечение, реализующее протоколы Межсетевого уровня. На рисунке 2.4.5 изображен пример составной IP-сети. Шлюз имеет два интерфейса: один принадлежит сети 172.16.5.0 другой сети – сети 172.16.12.0. Обе сети имеют маску 255.255.252.0, что соответствует 22 битовому адресу сети. Для обмена данными с хостом, который находится в другой сети, используется ***таблица***

маршрутов. Таблица маршрутов имеется на каждом ***узле*** сети (хост или шлюз) и содержит информацию об адресах сетей, адресах шлюзов и т.п. Процесс определения адреса следующего узла в пути следования дейтаграммы и пересылка ее по этому адресу называется ***маршрутизацией***. С процессом маршрутизации в IP-сети можно ознакомиться в [5,6].

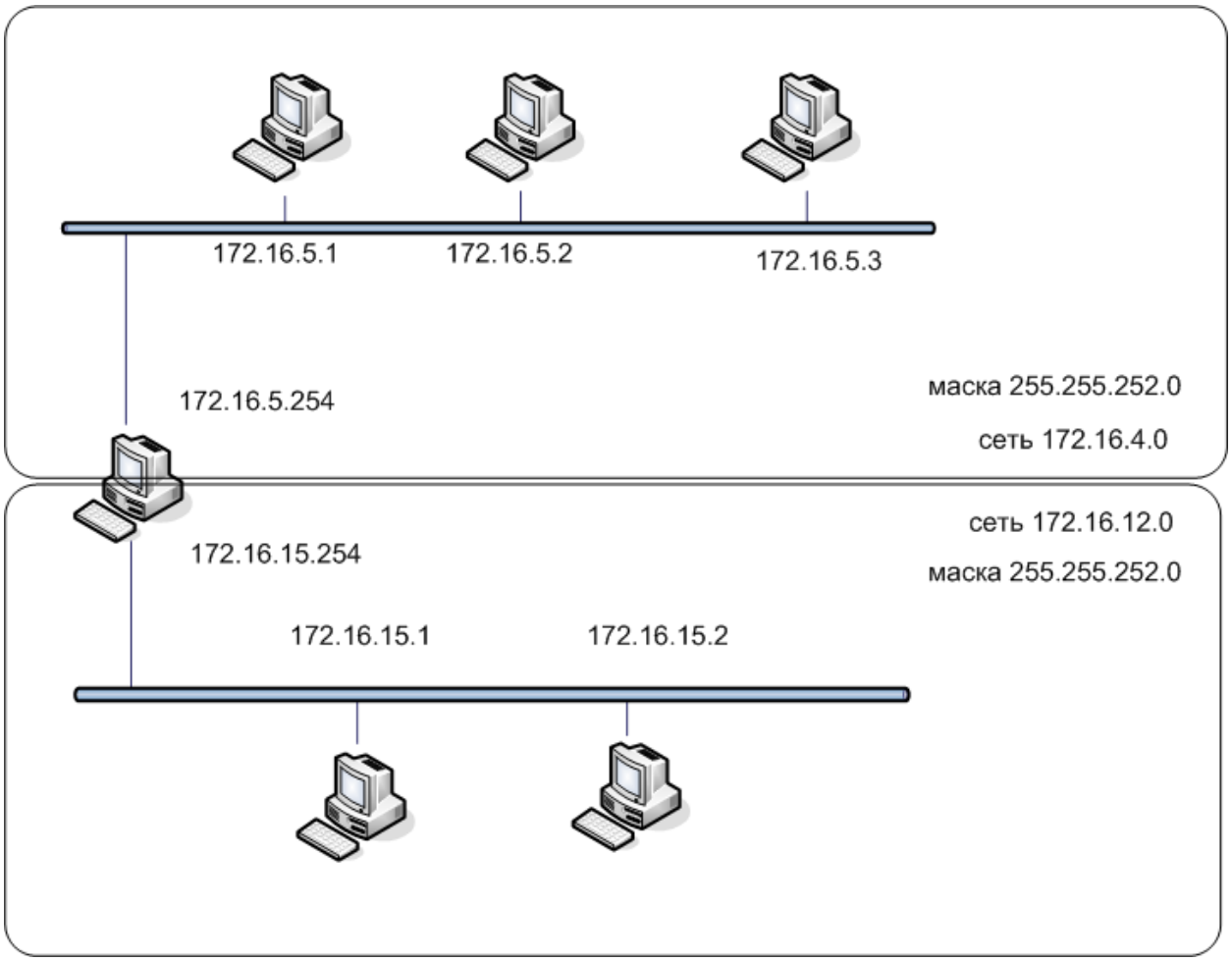

Рисунок 2.4.5. Пример составной IP-сети

Протокол ICMP. Спецификация протокола ICMP (Протокол контроля сообщений в Internet) изложена в документе RFC 792. ICMP является неотъемлемой частью TCP/IP и предназначен для транспортировки информации о сетевой деятельности и маршрутизации. IPCM сообщения представляют собой специально отформатированные IP-дейтаграммы, которым соответствуют определенные типы (15 типов) и коды сообщений. Описание типов и кодов IPCM-сообщений содержится в [5]. С помощью протокола ICMP осуществляется деятельность утилит достижимости (***ping***, ***traceroute***); регулируется частота отправки IP-дейтаграмм, оптимизируется MTU для маршрута передачи IP-дейтаграмм; доставляется хостам, маршрутизаторам и шлюзам всевозможная служебная информация;

осуществляется поиск и переадресация маршрутизаторов; оптимизируются маршруты; диагностируются ошибки и оповещаются узлы IP-сети.

Протокол ARP. IP-адреса могут восприниматься только на сетевом уровне и вышестоящих уровнях TCP/IP. На канальном уровне всегда действует другая схема адресации, которая зависит от используемого протокола. Например, в сетях Ethernet используются 48-битные адреса. Для установки соответствия между 32-разрядными IP-адресами и теми или иными MAC-адресами, действующими на канальном уровне, применяется механизм привязки адресов по протоколу ARP, спецификация которого приведена в документе RFC 826. Основной задачей ARP является динамическая (без вмешательства администратора, пользователя, прикладной программы) проекция IP-адресов в соответствующие MAC-адреса аппаратных средств. Эффективность работы ARP обеспечивается тем, что каждый хост кэширует специальную ARP-таблицу. Время существования записи в этой таблице составляет обычно 10- 20 минут с момента ее создания и может быть изменено с помощью параметров реестра [6]. Просмотреть текущее состояние ARP-таблицы можно с помощью команда ***arp***. Кроме того, протокол ARP используется для проверки существования в сети дублированного IP-адреса и разрешения запроса о собственном MAC-адресе хоста во время начальной загрузки.

Протокол RARP. Протокол RARP, как следует из названия (Reverse ARP), по своей функции противоположен протоколу ARP. RARP применятся для получения IP-адреса по MAC-адресу. В настоящее время протокол заменен на протокол Прикладного уровня DHCP, предлагающий более гибкий метод присвоение адресов.

Другие протоколы Межсетевого уровня. Следует отметить, что на Межсетевом уровне TCP/IP могут использоваться и другие протоколы: ***RIP*** (RFC 1058) – основной дистанционно-векторный протокол маршрутизации; ***OSPF*** (RFC 2328) – протокол первоначального открытия кратчайших маршрутов; ***BGP*** (RFC 1771) – пограничный межсетевой протокол и т.д. Сведения о назначении этих протоколов описаны в [5.6].

Протокол IPv6. Наиболее распространенной на настоящий момент версией протокола IP является IPv4 – именно об этой версии говорилось выше. Этот протокол оказался самым удачным сетевым протоколом из всех, когда-либо созданных. Поэтому IPv4 быстро превратился в стандарт. Можно сказать, что протокол IPv4 стал жертвой собственной популярности, т.к. предлагаемое полезное пространство адресов практически исчерпано. Как результат усилий направленных на решение этой проблемы появился протокол IPv6, в котором попутно было реализовано много других новых возможностей. Главным отличительным признаком протокола IPv6 является 128-битный адрес, позволяющий увеличить адресное пространство боле чем на 20 порядков. Основная концепция IPv6: каждый отдельный узел должен иметь собственный уникальный идентификатор интерфейса. Кроме того, протокол IPv6 требует соответствие идентификаторов интерфейсов формату IEEE EUI-64, позволяющему применять фиксированные (“зашитые” при

изготовлении в специальную память сетевой платы) MAC-адреса сетевых плат. Например, 48-битный MAC-адрес платы Ethernet изначально предназначен для глобальной идентификации. Первые 24 бита этого адреса обозначают производителя платы (в соответствии с кодировкой ICANN) и индивидуальную партию изделия, а остальные 24 бита определяются производителем, с таким расчетом, чтобы каждый номер был уникален в пределах всей его продукции. Таким образом уникальный идентификатор интерфейса IPv6 на основе Ethernet содержит в младших 64-х разрядах 128-битного адреса MAC-адрес платы Ethernet. Причем дополнение 48-бит адреса MAC-адреса до 64 бит осуществляется добавлением 16 бит (0xFFFF) между двумя его половинами. На первоначальном этапе внедрения IPv6 предполагается совместное использование обеих версий IP-протокола. При этом предполагается использование, так называемых, IPv4-совместимых и IPv4-преобразованных адресов. Другой интересной особенностью IPv6 является возможность ***автоконфигурации***. Автокофигурация – это процесс, позволяющий хосту находить информацию для настройки собственных IP-параметров. В версии IPv6 основным средством позволяющим выполнять подобную настройку является протокол DHCP. Пересмотр процесса автоконфигурации вызван сложность администрирования сетей с большим количеством хостов и в связи с необходимостью поддерживать мобильных (перемещающихся) пользователей. Большое внимание в новой версии протокола уделяется вопросам безопасности Более подробно о перспективном протоколе IPv6 можно узнать из [6].

2.5. Протоколы Транспортного уровня

Основным назначением протоколов транспортного уровня является сквозная доставка данных произвольного размера по сети между прикладными процессами, запущенными на узлах сети. Транспортный уровень TCP/IP представлен двумя протоколами: ***TCP*** (Transmission Control Protocol), чье имя присутствует в названии всего стека; **UDP** (User Datagram Protocol) – протокол передачи дейтаграмм пользователя. Процесс, получающий или отправляющий данные с помощью Транспортного уровня, идентифицируется номером, который называется ***номером порта***. Таким образом, адресат в сети TCP/IP полностью определяется тройкой: IP-адресом, номером порта и типом протокола транспортного уровня (UDP или TCP). Основным отличием протоколов UDP и TCP является, то, что UDP – протокол без установления соединения (ориентированным на сообщения), а TCP – протокол на основе соединения (ориентированный на поток). На рисунке 2.5.1 изображен стек протоколов TCP в двух разрезах и отображены названия принятые для обозначения блоков данных в протоколах TCP и UDP. Движение информации с верхних уровней на нижние сопровождается ***инкапсуляцией данных*** (упаковкой по принципу матрешки), а движение в обратном направлении – распаковкой. Отправляемый кадр данных содержит в себе всю необходимую информацию, чтобы быть доставленным (на

Уровне доступа к сети) и правильно распакованным на каждом уровне получателя и, наконец, предоставленным на Прикладном уровне в виде, пригодным для использования процессом.

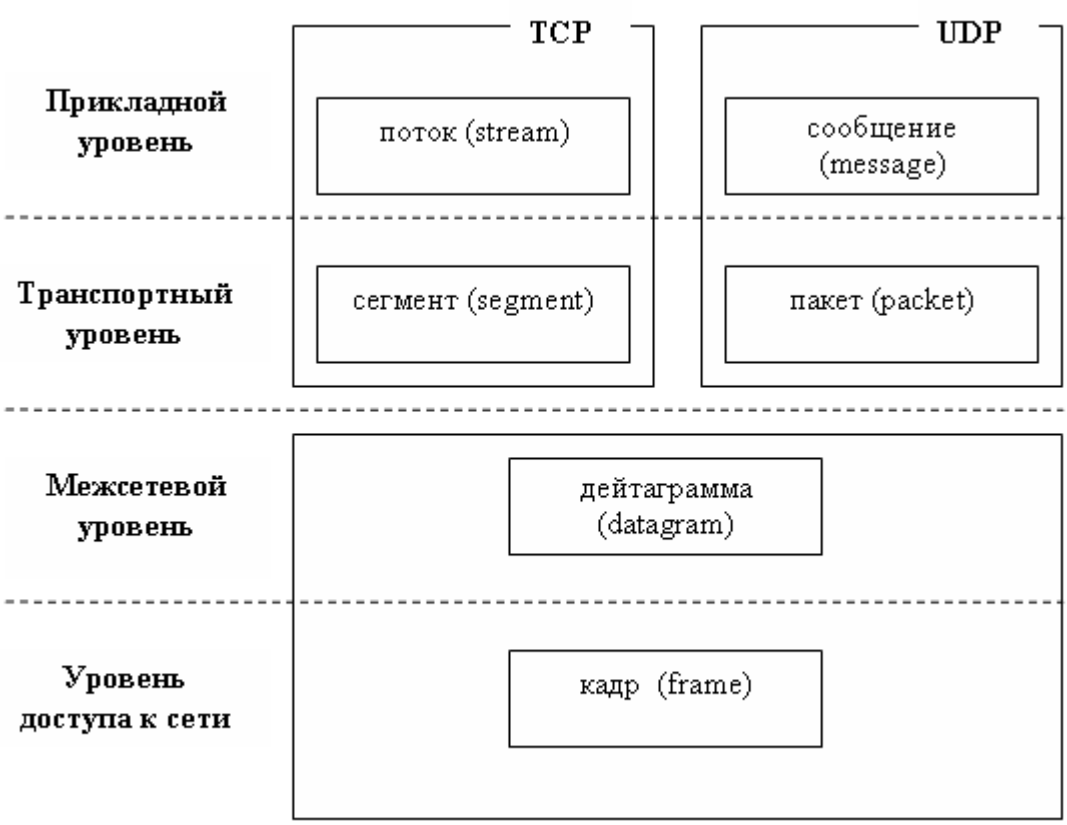

Рисунок 2.5.1. Протоколы TCP и UDP в стеке TCP/IP

Номера портов, используемые для идентификации прикладных процессов (в соответствии с документами IANA), делятся на три диапазона: ***хорошо известные номера портов (well-known port number)***, ***зарегистрированные номера портов (registered port number)***, ***динамически номера портов (dynamic port number)***. Распределение номеров портов по диапазонам приведено в таблице 2.5.1.

Таблица 2.5.1

Хорошо известные номера портов	0 – 1023
Зарегистрированные номера портов	1024 – 49151
Динамические номера портов	49152– 65535

Хорошо известные номера портов присваиваются ***базовым системным службам (core services)***, имеющие системные привилегии. Зарегистрированные номера портов присваиваются промышленным приложениям и процессам. Распределение некоторых хорошо известных и зарегистрированных номеров портов приведено в таблице 2.5.2. Динамические номера портов (их часто называют ***эфемерными портами***) выделяются, как правило, прикладным процессам специализированной службой операционной системы. Некоторые системы TCP/IP применяют

диапазон значений от 1024 до 5000 для назначения эфемерных номеров портов.

Таблица 2.5.2

Номер порта	Протокол	Описание
20	TCP	File Transport Protocol (FTP)
21	TCP	File Transport Protocol (FTP)
22	TCP	Secure Shell (SSH)
23	TCP	Telnet
25	TCP	Simple Mail Transfer Protocol (SMTP)
53	TCP	Domain Name Server (DNS)
53	UDP	Domain Name Server (DNS)
66	TCP	Oracle SQL*NET
67	UDP	Dynamic Host Configuration Protocol (DHCP) Server
68	UDP	Dynamic Host Configuration Protocol (DHCP) Client
80	TCP	World Wide Web (WWW)
110	TCP	Post Office Protocol, Version 3 (POP3)
111	TCP	Remote Procedure Call (RPC)
111	UDP	Remote Procedure Call (RPC)
143	TCP	Internet Message Access Protocol (IMAP4)
1352	TCP	Lotus Notes
1433	TCP	Microsoft SQL Server
1522	TCP	Oracle SQL Sever

Протокол UDP. Протокол UDP является протоколом без установления соединения. Спецификация протокола описывается в документе RFC 768. Основными свойствами протокола являются:

1) отсутствие механизмов обеспечения надежности: пакеты не упорядочиваются, и их прием не подтверждается;
2) отсутствие гарантий доставки: пакеты оправляются без гарантии доставки, поэтому процесс Прикладного уровня (программа пользователя) должен сам отслеживать и обеспечивать (если это необходимо повторную передачу);
3) отсутствие обработки соединений: каждый оправляемый или получаемый пакет является независимой единицей работы; UDP не имеет методов установления, управления и завершения соединения между отправителем и получателем данных;
4) UDP может по требованию вычислять контрольную сумму для пакета данных, но проверка соответствия контрольной сумы ложится на процесс Прикладного уровня;
5) отсутствие буферизации: UDP оперирует только одним пакетом и вся работа по буферизации ложится на процесс Прикладного уровня;

6) UDP не содержит средств, позволяющих разбивать сообщение на несколько пакетов (фрагментировать) – вся эта работа возложена на процесс Прикладного уровня.

Следует обратить внимания, что протокол UDP характеризуется тем, что он не обеспечивает. Все перечисленные отсутствующие характеристики присутствуют в протоколе TCP. Фактически UDP – это тонкая прослойка интерфейса, обеспечивающая доступ процессов Прикладного уровня непосредственно к протоколу IP.

Протокол TCP. Протокол TCP является ***надежным*** байт-ориентированным протоколом с ***установлением соединения***. При получении дейтаграммы, в поле Protocol (со структурой IP-дейтаграммы можно ознакомиться в [5,6]) которой указан код 6 (код протокола TCP) IP-протокол извлекает из дейтаграммы данные, предназначенные для Транспортного уровня, и переправляет их модулю протокола TCP. Модуль TCP анализирует служебную информацию заголовка сегмента (структура TCP-сегмента приведена в [5,6]), проверяет целостность (по контрольной сумме) и порядок прихода данных, а также подтверждает их прием отправляющей стороне. По мере получения правильной последовательности неискаженных данных процесса отправителя, используя поле Destination Port Number заголовка сегмента, модуль TCP переправляет эти данные процессу получателя.

Протокол TCP рассматривает данные отправителя как непрерывный не интерпретируемый (не содержащий управляющих для TCP команд) поток октетов. При этом TCP при отправке разделяет (если это необходимо) этот поток на части (TCP-сегменты) и объединяет полученные от протокола IP-дейтаграммы при приеме данных. Немедленную отправку данных может быть затребовано процессом с помощью специальной функции PUCH, иначе TCP сам решает, когда отправлять данные отправителя и когда их передавать получателю.

Модуль TCP обеспечивает защиту от повреждения, потери, дублирования и нарушения очередности получения данных. Для выполнения этих задач все октеты в потоке данных пронумерованы в возрастающем порядке. Заголовок каждого сегмента содержит число октетов и порядковый номер первого октета данных в данном сегменте. Каждый сегмент данных сопровождается контрольной суммой, позволяющей обнаружить повреждение данных. При отправлении некоторого числа последовательных октетов данных, отправитель ожидает подтверждение приема. Если подтверждения не приходит, то предполагается, что группа октетов не дошла по назначению или была повреждена – в этом случае предпринимается повторная попытка переслать данные.

Протокол TCP обеспечивает одновременно нескольких соединений. Поэтому говорят о ***разделении каналов***. Каждый процесс Прикладного уровня идентифицируется номером порта. Заголовок TCP-сегмента содержит номера портов отправителя и получателя.

На рисунке 2.5.2 прерывистыми линиями изображены каналы между процессами Прикладного уровня A, B, C, D и E. Процесс D работает на хосте с IP-адресом 172.16.5.2 (рисунок 2.5.1) и использует порты 2777 и 2888 для связи с двумя процессами A (порт 2000) и B (порт 2500), функционирующими на хосте с IP-адресом 172.16.15.1. Процессы C и E образуют канал между хостами 172.16.5.1 и 172.16.5.3 и используют порты 2500 и 2000 соответственно.

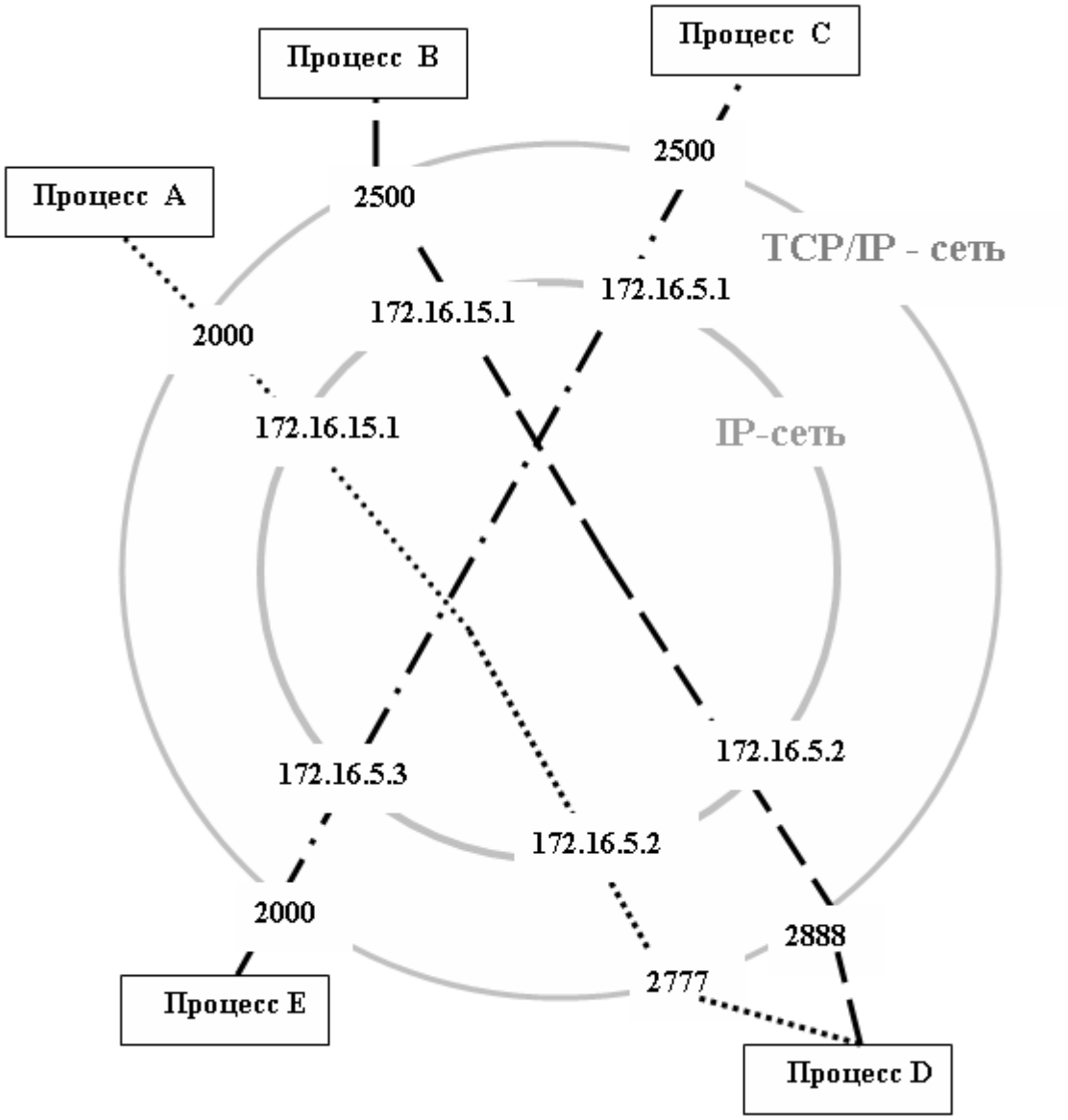

Рисунок 2.5.2. Разделение каналов в сети TCP/IP

Совокупность IP-адреса и номера порта называется ***сокетом***. Сокет однозначно идентифицирует прикладной процесс в сети TCP/IP. Следует помнить, что одни и те же номера портов могут быть использованы как для протокола UDP, так и для протокола TCP.

2.6. Интерфейс внутренней петли

Большинство реализаций TCP/IP поддерживает ***интерфейс внутренней петли (loopback interface)***, который позволяет двум прикладным процессам, находящимся на одном хосте, обмениваться данными посредством протокола TCP/IP. При этом, как обычно, формируются дейтаграммы, но они не покидают пределы одного хоста. Для интерфейса внутренней петли, как уже упоминалось выше, зарезервирована сеть 127.0.0.0. В соответствии с общепринятыми соглашениями, большинство операционных систем назначают для интерфейса внутренней петли адрес 127.0.0.1 и присваивают символическое имя ***localhost***.

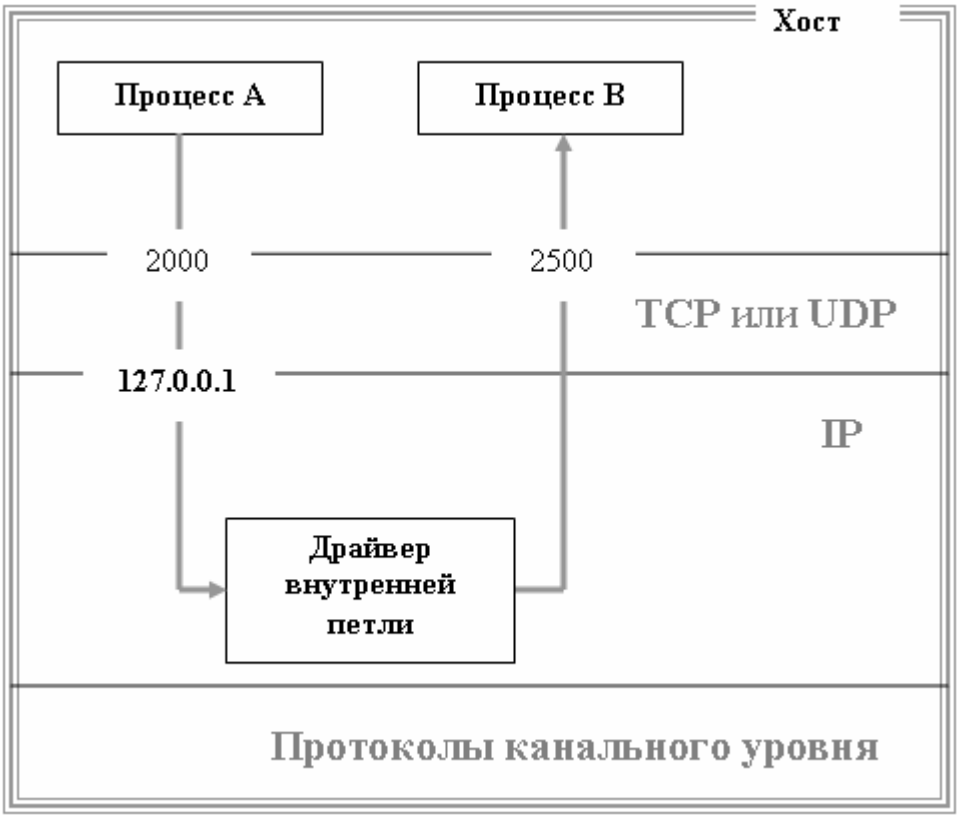

Рисунок 2.6.1. Схема работы интерфейса внутренней петли

На рисунке 2.6.1 приведена упрощенная схема обработки данных интерфейсом внутренней петли. Прикладной процесс А, изображенный на рисунке, используя номер порта 2000, отправляет данные процессу В. Указав в параметрах сокета процесса В сетевой адрес 127.0.0.1, процесс А обеспечил обработку посылаемых дейтаграмм на Межсетевом уровне драйвером внутренней петли, который направляет эти дейтаграммы во входную очередь модуля IP. IP- модуль, следуя обыкновенной логике своей работы, доставляет данные на Транспортный уровень. Далее протокол

Транспортного уровня в соответствии с номером порта 2500 в заголовке сегмента (или пакета) направляет данные процессу B.

Следует обратить внимание на следующее: все данные пересылаемые по интерфейсу внутренней петли не только не покидают пределов хоста, но и не затрагивают никаких внешних механизмов за пределами стека TCP/IP.

2.7. Интерфейсы сокетов и RPC

Для предоставления возможности разработчикам программного обеспечения использовать процедуры стека TCP/IP в состав операционных систем включаются специальные интерфейсы (***Application Programming Interface, API***), представляющие собой, как правило, набор специальных функций и технологических инструкций, обеспечивающих доступ к модулям протокола TCP/IP.

Наиболее распространенными API для обмена данными в сети, являются ***интерфейс сокетов*** и ***RPC (Remote Procedure Call)*** – вызов удаленных процедур.

API сокетов. API сокетов – это название программного интерфейса, предназначенного для обмена данными между процессами, находящимися на одном или на разных объединенных сетью компьютерах. Сокетом, кроме того, называют абстрактный объект, представляющий оконечную точку соединения. Впервые этот интерфейс появился в 1980-х годах в операционной системе BSD Unix (Berkeley Software Distribution), разработанной в университете Беркли (США, Калифорния) и описан в стандарте ***POSIX*** (Portable Operating System Interface for Unix).

Стандарт POSIX – это набор документов, описывающих интерфейсы между прикладной программой и операционной системой. Стандарт создан для обеспечения совместимости различных Unix-подобных операционных систем и переносимости исходных программ на уровне исходного кода. Официально стандарт определен как IEEE 1003, международное название стандарта ISO/IEC 9945.

В той или иной мере интерфейс сокетов поддерживается большинством современных операционных систем. Это дает возможность, например, Unix-процессу, реализованному на языке C с использованием API сокетов, обмениваться данными с Windows-приложением или с приложением, работающим на мэйнфрейме, если эти приложения используют API сокетов. Например, в операционной системе Windows интерфейс сокетов имеет название ***Windows Sockets API***. API сокетов включает в себя функции создания сокета (имеется в виду объект операционной системы, описывающий соединение), установки параметров сокета (сетевой адрес, номер порта и т.д.), функции создания канала и обмена данными между сокетами. Кроме того, есть набор функций позволяющий управлять передачей данных, синхронизировать процессы передачи и приема данных, обрабатывать ошибки и т.п. Следует отметить, что интерфейсы сокетов, поддерживаемые различными операционными

системами, отличаются друг от друга, но все обеспечивают работу сокетов в стандартном режиме. В этом случае говорят о ***BSD-сокетах.***

Интерфейс сокетов используется большинством программных систем имеющих архитектуру клиент-сервер. К ним относятся сетевые службы, Web-серверы, серверы баз банных, серверы приложений и т.п.

Интерфейс RPC. Интерфейс RPC определяет программный механизм, который первоначально был разработан в компании Sun Microsystems и предназначался для того, чтобы упростить разработку распределенных приложений. Спецификация RPC компании Sun Microsystems содержится в документах RFC 1059, 1057, 1257.

RPC Sun Microsystems реализована в двух модификациях: одна выполнена на основе API сокетов для работы над TCP и UDP, другая, названная ***TI-RPC (Transport Independent RPC)***, использует API TLI (Transport Layer Interface, компании AT&T) и способна работать с любым транспортным протоколом, поддерживаемый ядром операционной системы.

Идея, положенная в основу RPC, заключается в разработке специального API, позволяющего осуществлять вызов ***удаленной процедуры*** (процедуры, которая находится и исполняется на другом хосте) способом, по возможности, ничем не отличающимся от вызова локальной процедуры из динамической библиотеки. Реализация этой идеи осложняется необходимостью учитывать возможность различия операционных сред, в которых работают вызывающая и вызываемая процедуры (отсюда, различные типы данных, невозможность обрабатывать адресные указатели и т.п.). Кроме того, следует предусмотреть обработку внепланового завершения процедуры на одной из сторон распределенного приложения. Все эти проблемы сделали интерфейс RPC достаточно сложным. Прозрачность механизма вызова достигается созданием вместо вызываемой и вызывающей процедур специальных программных заглушек, называемых ***клиентским*** и ***серверным стабами.***

Клиентским стабом называется тот стаб, который находится на хосте с вызывающей процедурой. Его основной задачей является преобразовать передаваемые параметры в формат стандарта ***XDR (External Data Representation)*** и скрыть (подменив вызываемую удаленную процедуру локальным вызовом стаба) от пользователя механизм RPC.

Серверный стаб находится на том же хосте, что и вызываемая процедура и предназначен для преобразования полученных параметров из формата XDR в формат, воспринимаемый вызываемой процедуры, а также для сокрытия (серверный стаб подменяет вызывающую процедуру на стороне сервера) RPC-механизма от вызываемой процедуры.

Стандарт XDR предназначен кодирования полей в запросах и ответах интерфейса RPC. Стандарт регламентирует все типы данных и уточняет способ их передачи в RPC-сообщениях. Спецификация стандарта XDR приведена в RFC 1014.

Число вызываемых удаленных процедур не регламентируется спецификацией RPC. Поэтому за ними не закрепляются конкретные TCP-

порты. Порты получают сами удаленные процедуры динамическим образом (эфемерные порты). Учет соответствия портов вызываемым процедурам осуществляет специальная программа ***PortMapper*** (регистратор портов). Сам PortMapper доступен по 111 порту и тоже является удаленной процедурой. Процедура PortMapper – это связующее звено между различными компонентами системы. Всякая вызываемая процедура, должна быть зарегистрирована в базе данных PortMapper с помощью специальных служебных функций. Вызывающая сторона (клиентский стаб) с помощью все тех же служебных функций может получить спецификацию вызываемой процедуры.

Развитием технологии RPC для объектно-ориентированного программирования в операционной системе Windows являются технологии ***COM*** и ***DCOM***, которые позволяют создавать удаленные объекты.

Аналогом RPC в Java-технологиях является механизм ***RMI (Remote Method Invocation)***, позволяющий работать с удаленными Java-объектами. Информацию об объекте и его методах вызывающая сторона может получить, обратившись к реестру RMI (аналогу PortMapper).

Сетевая файловая система ***NFS (Network File System)***, повсеместно используемая в качестве службы прозрачного удаленного доступа к файлам основана на механизме RPC Sun Microsystems.

Следует отметить, что кроме Sun-реализации интерфейса RPC, широко применяется и конкурирующий программный продукт, разработанный объединением OSF (Open Software Foundation).

2.8. Основные службы TCP/IP

Программную реализацию Протоколов Прикладного уровня TCP/IP принято называть службами. Работа служб TCP/IP определяется множеством соглашений: спецификациями структур сообщений, поддерживаемых данной службой; регистрацией хорошо известного порта используемого службой; спецификациями программных компонентов, необходимых для работы службы и т.д.

Как правило, служба реализуются в виде сервера, предоставляющего услуги клиентам (другим процессам). В этом случае клиенты использует запросы (определенные спецификациями) и получает соответствующий сервис, предусмотренный данному запросу. Однако ярко выраженной архитектурой клиент-сервер обладают не все службы. В некоторых случаях сами службы могут выступать в роли клиента или осуществлять межсерверный (между однородными серверами) обмен данными, обычно используемый для синхронизации (***репликации***) серверов.

Существуют тысячи служб TCP/IP, спецификации которых изложены в документах RFC. Здесь рассматриваются лишь те, которые принято называть ***традиционными службами TCP/IP***.

2.8.1. Служба и протокол DNS

Служба ***DNS (Domain Name System)*** является одной из важнейших служб TCP/IP, само появление которой в 1980-х годах дало мощный толчок развитию TCP/IP и всемирной сети Internet. Дело в том, что DNS обеспечивает важную возможность преобразования символических доменных имен в соответствующие IP-адреса (***разрешение имен***). Например, для обращения к адресу серверу компании Microsoft, имеющему IP-адрес 207.46.230.229, можно обратиться, используя символическое имя microsoft.com. С одной стороны, это дает более наглядную нотацию, а с другой, появляется возможность не привязывать жестко получение услуг сервера к фиксированному адресу, который при реорганизации сети может измениться.

Службу DNS можно рассматривать, как распределенную иерархическую базу данных, основное назначение которой отвечать на два вида запросов: выдать IP-адрес по символическому имени хоста и наоборот – выдать символическое имя хоста по его IP-адресу. Облуживание этих запросов и поддержку базы данных в актуальном состоянии обеспечивают взаимодействующие глобально рассредоточенные в сети Internet серверы DNS. База данных имеет древовидную структуру, в корне которой нет ничего, а сразу под корнем находятся ***первичные*** сегменты ***(домены)***: **.com, .edu, .gov, …, .ru, .by, .uk, …** Наименование этих первичных доменов отражает деление базы данных DNS по отраслевому (домены, обозначенные трехбуквенным кодом) и национальному признакам (двухбуквенные домены в соответствии со стандартом ISO 3166). ***Доменом*** в терминологии DNS называется любое поддерево дерева базы данных DNS.

DNS-серверы, обеспечивающие работоспособность всей глобальной службы, тоже имеют древовидную структуру подчиненности, которая соответствует структуре распределенной базы данных. По своему функциональному назначению DNS-серверы бывают: первичные серверы (которые являются главными серверами, поддерживающими свою часть базы данных DNS), вторичные серверы (всегда привязан к некоторому первичному серверу и используются для дублирования данных первичного сервера), кэширующие серверы (обеспечивают хранение недавно используемых записей из других доменов и служат для увеличения скорости обработки запросов на разрешение имен).

При обработке запроса на разрешение имени, хост, как правило, обращается к первичному или вторичному DNS-серверу, обслуживающему данный домен сети. В зависимости от сложности запроса, DNS-сервер может сам ответить на запрос или переадресовать к другому серверу DNS. Последней инстанцией в разрешении имен являются пятнадцать (на сегодняшний день) корневых серверов имен, представляющих собой вершину всемирной иерархии DNS.

Разработчик приложения может обратиться за разрешением имени с помощью функций, имеющих, как правило, имена ***gethostbyname*** и ***gethostbyaddr***.

Более подробно с принципами работой службы DNS можно ознакомиться в [6].

2.8.2. Служба и протокол DHCP

DHCP (Dynamic Host Configuration Protocol) – это сетевая служба (и протокол) Прикладного уровня TCP/IP, обеспечивающая выделение и доставку IP-адресов и сопутствующей конфигурационной информации (маска сети, адрес локального шлюза, адреса серверов DNS и т.п.) хостам. Применение DHCP дает возможность отказаться от фиксированных IP-адресов в зоне действия сервера DHCP. Описание протокола DHCP содержится в документах: RFC 1534, 2131, 2132, 2141.

Конструктивно служба DHCP состоит из трех модулей: ***сервера DHCP (DHCP Server)***, ***клиента DHCP (DHCP Client)*** и ***ретранслятора DHCP (DHCP Replay Agent)*** .

DHCP-серверы способны управлять одним или несколькими диапазонами IP-адресов (***адресными пулами***). В пределах одного пула можно всегда выделить адреса, которые не должны распределяться между хостами. DHCP-серверы используют для приема запросов от DHCP-клиентов порт 67. Выделение IP-адресов может быть трех типов: ***ручной***, ***автоматический*** и ***динамический*** [5,6]. Обычно DHCP-серверы устанавливают на компьютерах, исполняющих роль сервера в сети.

DHCP-клиенты представляет собой программный компонент, обычно реализуемый как часть стека протоколов TCP/IP и предназначен для формирования и пересылки запросов к DHCP-серверу на выделение IP-адреса, продления срока аренды IP-адреса и т.п. DHCP- клиенты используют для приема сообщений от DHCP-сервера порт 68.

Логика работы протокола DHCP достаточно проста. При физическом подключении к сети, хост пытается подсоединиться к сети, используя для этого DHCP-клиент. Для обнаружения DHCP-сервера DHCP-клиент выдает в сеть широковещательный запрос (это процесс называется ***DHCP-поиском***). Если в этом домене есть DHCP-сервер, то он окликается, посылая клиенту специальное сообщение, содержащее IP-адрес DHCP-сервера. Если доступны несколько DHCP-серверов, то, как правило, выбирается первый ответивший. Получив адрес сервера, клиент формирует запрос на выделение IP-адреса из пула адресов DHCP-сервера. В ответ на запрос, DHCP-сервер выделяет адрес клиенту на определенный период времени (***аренда адреса***). После получения IP-адреса TCP/IP-стек клиента начинает его использовать. Продолжительность аренды адреса устанавливается специально или по умолчанию (может колебаться от нескольких часов до нескольких недель). После истечения срока аренды DHCP-клиент пытается снова договорится с DHCP-сервером о продлении срока аренды или о выделении нового IP-адреса.

Ретранслятор DHCP используется в том случае, если на первоначальном этапе подключения к сети широковещательные запросы DHCP-клиента не могут быть доставлены (по разным причинам) DHCP-

серверу. Ретранслятор в этом случае играет роль посредника между DHCP-клиентом и DHCP-сервером.

Протокол DHCPv6. Протокол DHCPv6 – это новый протокол, работающий над IPv6. Основные задачи DHCPv6 не сильно отличаются от задач выполняемых его предшественником – протоколом DHCPv4 (выше он назывался просто DHCP). Помимо очевидного отличия в длине и формате IP-адресов, значительное расхождение заключается в том, что IPv6-узлы теперь смогут получить (хотя бы локально функционирующие) IPv6- адреса без помощи DHCPv6. Таким образом, осуществляя поиск DHCPv6-сервера IPv6-узлы уже обладают некоторым IPv6-адресом. По-все видимости, основным назначением DHCPv6-протокола будет выделение глобально уникальных IPv6-адресов для тех интерфейсов, которые не могут их сформировать сами и для пересылки дополнительной конфигурационной информации IPv6-хостам.

2.8.3. NetBIOS over TCP/IP

Система ***NetBIOS (Network Basis Input/Output System)*** была разработана в 1985 году в компании Sytek, а позже была заимствована IBM и Microsoft как средство присвоение имен сетевым ресурсам в небольших одноранговых сетях. Вначале NetBIOS была не протоколом, а программным интерфейсом (API) для обращения к сетевым ресурсам. В качестве транспортного протокола выступал ***NetBEUI (NetBIOS Enhanced User Interface)*** – расширенный пользовательский интерфейс NetBIOS.

Служба NetBIOS, применяющая TCP/IP в качестве транспортного протокола (NetBIOS over TCP/IP) называется ***NetBT*** или ***NBT***. Служба NBT, по отношению к стандартному NetBIOS, претерпела значительные изменения, позволяющие передавать NBT-имена компьютеров и команды NBT по TCP/IP-соединению. Эти решение были опубликованы в 1987 году в документах RFC 1001, 1002.

На сегодняшний момент система NBT используется операционной системой Windows для совместного использования сетевых ресурсов и разрешения имен компьютеров. NBT-имя компьютера, это то имя которое задается при инсталляции Windows или может быть установлено (или скорректировано) с помощью программы MyComputer. При этом существует три способа разрешения NBT-имени в сети Windows: запрос к серверу DNS (или аналогичной службе), запрос к локальному сегменту сети (с помощью широковещания) и поиск в локальном списке хоста (файл hosts). Единственный реальный способ разрешения имен в крупномасштабных сетях является DNS-разрешение. Теоретически компьютер в сети Windows может иметь два имени NBT-имя и DNS-имя (следует, правда оговориться, что в случае с DNS именуется не компьютер, а IP-интерфейс).

Более подробно с протоколом NBT можно ознакомиться в [5,6].

2.8.4. Служба и протокол Telnet

Сетевая служба Telnet как протокол описана в двух документах RFC 854 (собственно протокол Telnet, 1985г.) и RFC 855 (дополнительные возможности Telnet). Служба создавалась для подключения пользователей к удаленному компьютеру для выполнения вычислений, работы с базами данных, подготовки документов и т.д. Кроме того, служба применяется для управления работой удаленного компьютера, для настройки и диагностирования сетевого оборудования.

Служба Telnet имеет архитектуру клиент-сервер. Стандартно для обмена данными между клиентом и сервером используется порт 23, но часто используют и другие порты (это допускается протоколом).

В основу службы Telnet положены три фундаментальные идеи: 1) концепция ***виртуального терминала NVT(Network Virtual Terminal)***; 2) принцип договорных опций (согласование параметров взаимодействия); 3) симметрия связи между терминалом и процессом.

Виртуальный терминал, представляет собой спецификацию, позволяющую клиенту и серверу преобразовать передаваемые данные в стандартную (понятную для обеих сторон) форму, позволяющую вводить, принимать и отображать эти данные. Применение виртуального терминала позволяет унифицировать характеристики различных устройств и этим обеспечить их совместимость. В традиционном смысле виртуальный терминал сети понимается как клавиатура печатающего устройства, принимающая и печатающая байты с другого хоста.

Опции представляют собой параметры или соглашения, применяемые в ходе Telnet-соединения. К примеру, опция Echo определяет, отражает ли хост Telnet символы данных, получаемых по соединению. Некоторые опции требуют обмена дополнительной информацией. До обмена дополнительной информацией оба хоста (клиент и сервер) должны согласится на обсуждение параметров, а затем использовать команду SB для начала обсуждения. Полный список опций Telnet опубликован на сайте http://www.iana.org.

Возможность указания TCP-порта при подключении к хосту, позволяет использовать Telnet для диагностирования других Internet-служб.

Серьезным недостатком протокола Telnet является передача данных в открытом виде. Этот недостаток существенно снижает область применения протокола.

Следует отметить, что существует другие программные продукты, подобные Telnet. Например, протокол ***SSH (Secure Shell)*** или свободно распространяемая программа ***PuTTY*** . При разработке этих продуктов был учтен опыт длительной эксплуатации протокола Telnet и исправлены его основные недостатки.

2.8.5. Служба и протокол FTP

Протокол ***FTP (File Transport Protocol)*** описывает методы передачи файлов между хостами сети TCP/IP с использованием транспортного протокола на основе соединений (TCP). Такое же имя FTP носит служба,

реализующая этот протокол. Описание протокола FTP содержится в документе RFC 959 (1985 г.).

Служба FTP имеет архитектуру клиент-сервер (рисунок 2.8.5.1) и содержит три ключевые компоненты: UI(User Interface), PI (Protocol Interpreter) и DTP (Data Transfer Process).

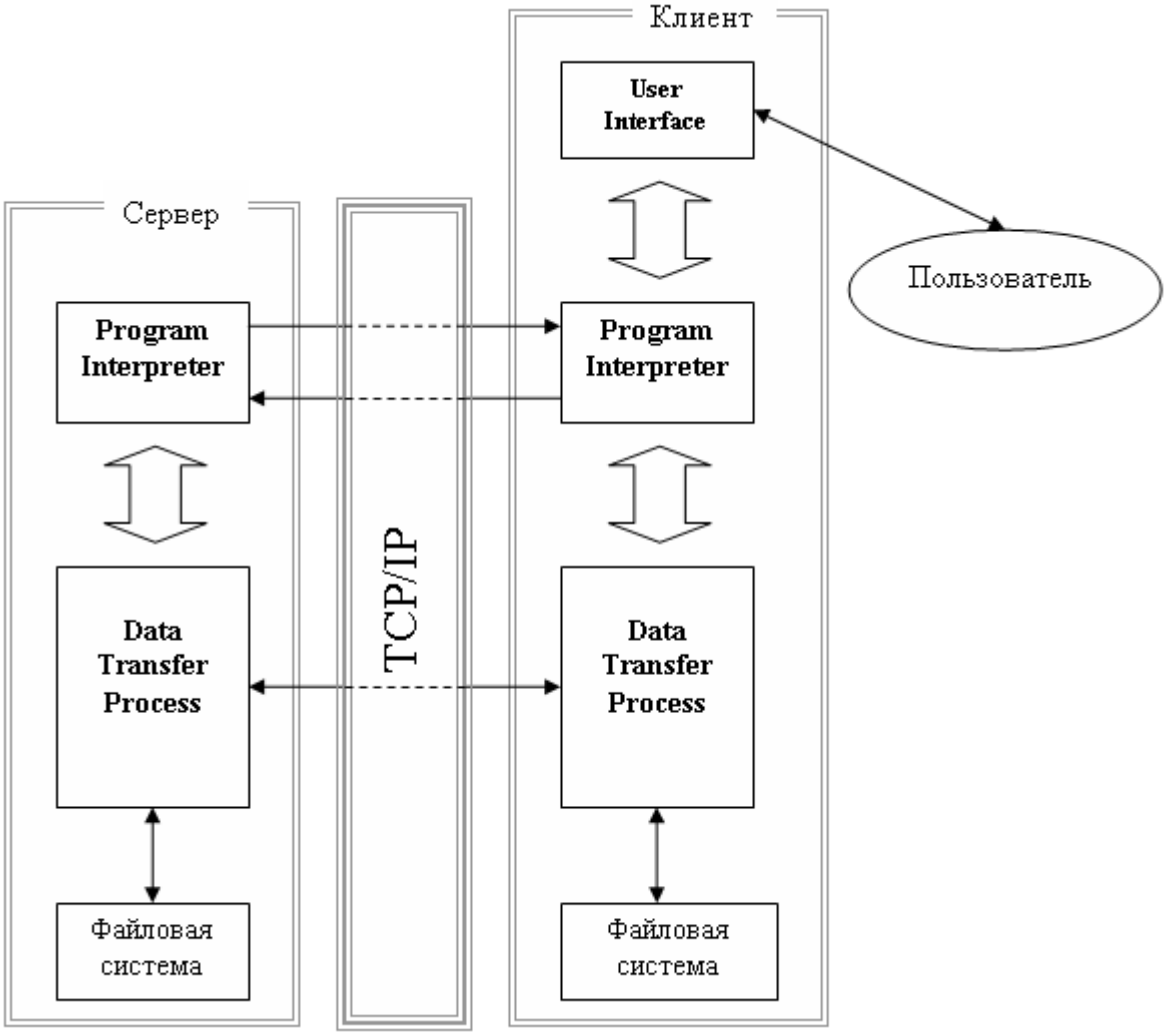

Рисунок 2.8.5.1. Взаимодействие основных компонентов службы FTP

UI – это внешняя оболочка, обеспечивающая интерфейс пользователя. К примеру, клиент FTP операционной системы Windows, обеспечивает интерфейс в виде консоли с командной строкой.

PI – интерпретатор протокола, предназначенный для интерпретации команд пользователя. Кроме того, PI клиента используя эфемерный порт, инициирует соединение с портом 21 PI сервера. Созданный канал (он называется ***каналом управления***) используется для передачи команд пользователя интерпретатору сервера и получения и его откликов. Интерпретатор протокола обрабатывает команды, позволяющие пересылать и удалять файлы, создавать, просматривать и удалять директории и т.п.

DTP – процесс передачи данных, предназначенный для фактического перемещения данных, в соответствии с командами управления, переданными

по каналу управления. Кроме того, на DTP сервера возложена инициатива создания канала передачи данных с DTP клиента. Для этого на стороне клиента используется, как правило, порт 20.

Файловая система на любом конце FTP-соединения может состоять из файлов различного формата: ASCII, EBCDIC, бинарный формат и т.д.

2.8.6. Электронная почта и протоколы SMTP, POP3, IMAP4

Протоколы Прикладного уровня ***SMTP (Simple Mail Transport Protocol)***, ***POP3 (Post Office Protocol)*** и ***IMAP4 (Internet Message Access Protocol)*** являются основой для создания современной электронной почты.

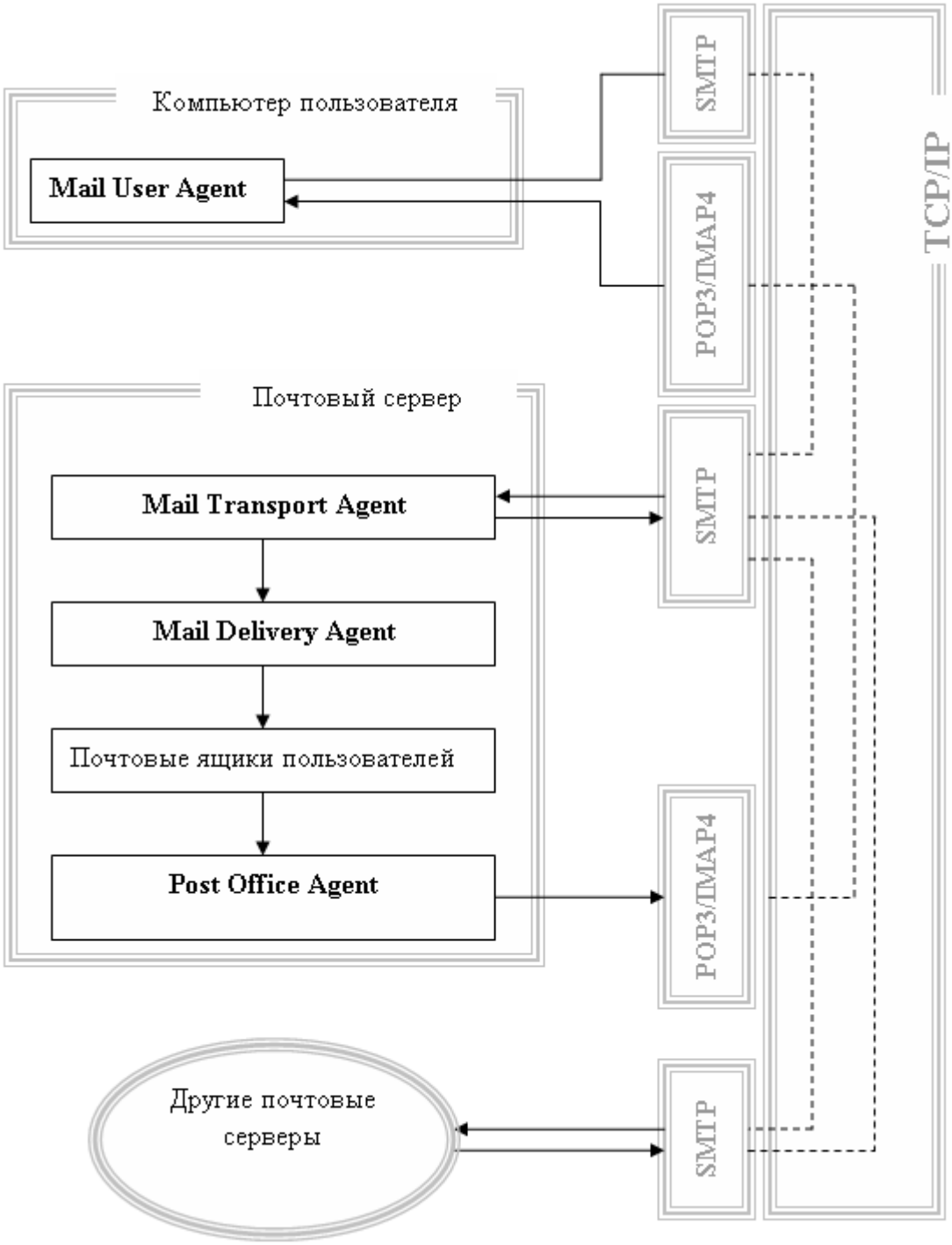

Рисунок 2.8.6.1 Схема взаимодействия компонентов электронной почты

Основными компонентами системы электронной почты являются: ***MTA (Mail Transport Agent)***, ***MDA (Mail Delivery Agent)***, ***POA (Post Office Agent)*** и ***MUA (Mail User Agent)***. На рисунке 2.8.6.1 изображена схема взаимодействия эти компонентов.

MTA – транспортный агент, основное назначение которого: прием почтовых сообщений от пользовательских машин; отправка почтовых сообщений другим MTA (установленных на других почтовых системах); прием сообщений от других MTA; вызов MDA. Это компонент реализован в виде сервера, прослушивающего порт 25 и работающего по протоколу SMTP.

MDA – агент доставки, предназначенный для записи почтового сообщения в почтовый ящик. MDA реализован в виде отдельной программы, которую вызывает MTA по мере необходимости. Обычно, MDA располагают на том же компьютере, что и MTA.

POA – агент почтового отделения, позволяющий пользователю получить почтовое сообщение на свой компьютер. POA реализован в виде сервера, прослушивающего порты 110 и 143. При этом, порт 110 работает по протоколу POP3, порт 143 – IMAP4.

MUA – почтовый агент пользователя позволяет принимать почту по протоколам POP3 и IMAP4 и отправлять почту по протоколу SMTP.

Когда говорят о ***почтовом сервере***, то, обычно подразумевают совокупность серверов MTA, POA, программу MDA, а также систему хранения почтовых сообщений (почтовые ящики) и ряд дополнительных программ, обеспечивающих безопасность и дополнительный сервис, расположенные на отдельном компьютере с TCP/IP-интерфейсом. Наиболее известными являются два почтовых сервера: ***Lotus Notes*** (IBM) и ***Microsoft Exchange Server. Почтовый клиент*** представляет собой программу, устанавливаемую на пользовательском компьютере и взаимодействующую с серверами MTA и POP3, почтового сервера, с помощью TPC/IP – соединения. Например, стандартным клиентом для отправления и организации работы с почтой в ОС Window является программа ***Outlook Express***.

2.8.7. Протокол HTTP и служба WWW

Протокол ***HTTP (Hypertext Transfer Protocol)*** – это протокол Прикладного уровня, доставляющий информацию между различными ***гипермедийными*** системами. Под понятием ***гипермедийной системы*** понимается компьютерное представление системы данных, элементы которой представляются в различных форматах (гипертекст, графические изображения, видеоизображения, звук и т.д.) и обеспечивается автоматическая поддержка смысловых связей между представлениями элементов.

Протокол HTTP применяется в Internet с 1990 года. В настоящее время широкое распространение имеет версия HTTP 1.0, описанная в документе RFC 1945. Разработана новая версия HTTP 1.1 (документ RFC 2616), но пока она находится в стадии предложенного стандарта.

По умолчанию HTTP использует порт 80 и предназначен для построения систем архитектуры клиент-сервер. Запросы клиентов содержат ***URI (Uniform Resource Identifier)*** - универсальный идентификатор ресурса, позволяющий определить у сервера затребованный ресурс. URI представляет собой сочетание ***URL (Uniform Resource Locator)*** и ***URN (Uniform Resource Name)***. URL – унифицированный адресатор ресурсов: предназначен для указания места нахождения ресурса в сети. URN – унифицированное имя ресурса: идентифицирует ресурс, по указанному месту его нахождения (подразумевается, что по данному адресу может быть представлено несколько различных ресурсов). Например, пусть **http://isit301-14:1118/em** – URI, позволяющий вызвать программу Enterprise Manager Oracle Server. Тогда первая часть **http://isit301-14:1118** представляет собой URL (указывает имя хоста и номер порта), а **em** есть URN, идентифицирующее имя ресурса.

Служба ***WWW (World Wide Web)*** предназначена для доступа к ***гипертекстовым документам*** в сети Internet и включает в себя три основных компонента: протокол HTTP, URI-идентификация ресурсов и язык ***HTML (Hyper Text Markup Language)*** .

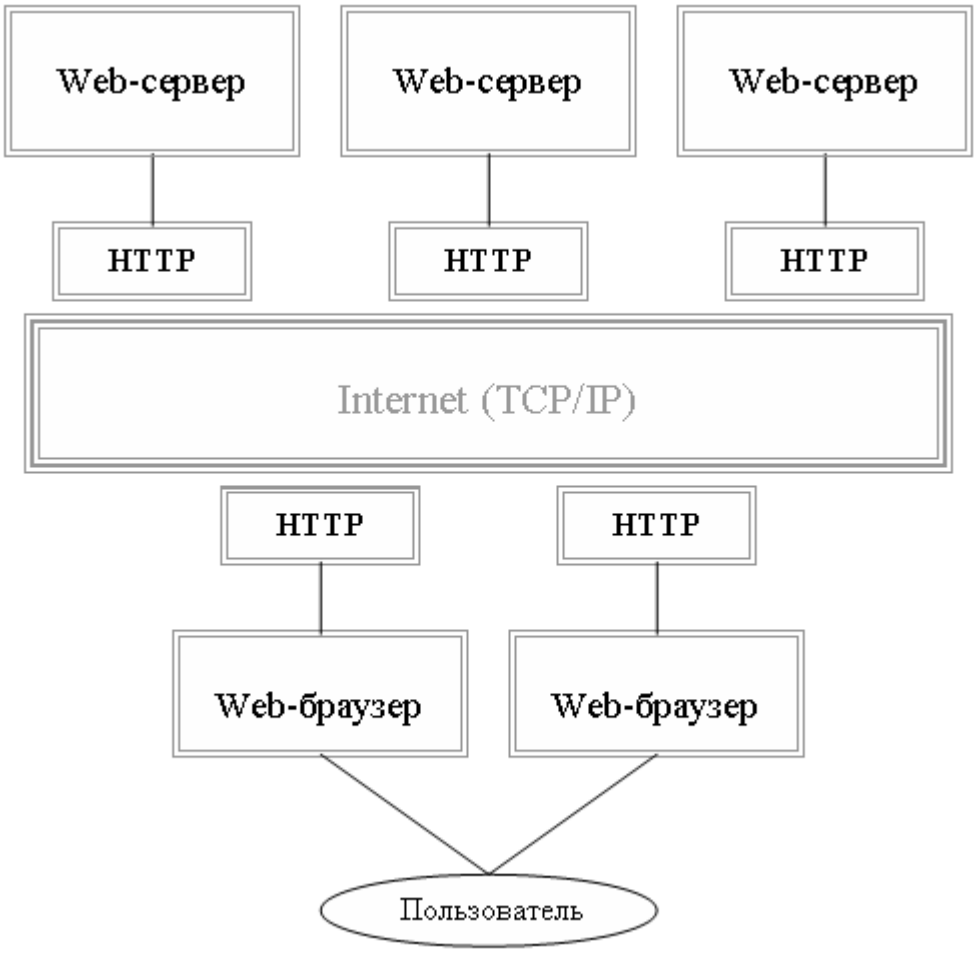

Рисунок 2.8.7.1. Клиент-серверная архитектура службы WWW

HTML – это стандарт оформления гипертекстовых документов. Гипертекстовый документ отличается от любого другого документа тем, что может содержать блоки, физически хранящиеся на разных компьютерах сети Internet. Основной особенностью HTML состоит в том, что форматирование в документе записывается только с помощью ASCII-символов. Одним из ключевых понятий гипертекстового документа является ***гипертекстовая ссылка***. Гипертекстовая ссылка – это объект гипертекстового документа, предназначенный для обозначения URI другого гипертекстового документа.

Как и все службы Internet, служба WWW имеет архитектуру клиент-сервер (рисунок 2.8.7.1). Серверная и клиентская части службы (обычно называемые Web-сервер и Web-браузер) взаимодействуют друг с другом с помощью протокола HTTP. В настоящее время наиболее известными серверными программами являются ***Apache Web Server, Apache Tomcat*** (продукты компании Apache Software Foundation (ASF), распространяемые в соответствии с лицензионным соглашением), ***Microsoft IIS*** (входит в состав дистрибутива последних версий Windows). Наиболее популярными Web-браузерами являются ***Microsoft Internet Explorer***, ***Netscape Navigator***, ***Opera***, ***Mozilla Firebox***.

2.9. Сетевые утилиты

Утилиты представляют собой внешние команды операционной системы и предназначаются для диагностики сети. Сетевые утилиты, поддерживаемые операционной системой Windows, перечислены в таблице 2.9.1. При этом утилиты, наименования которых выделено жирным шрифтом считаются стандартными для протокола TCP/P и присутствуют в большинстве операционных систем.

Таблица 2.9.1

Наименование утилиты	**Назначение утилит**
ping	Проверка соединения с одним или более хостами в сети
tracert	Определение маршрута до пункта назначения
route	Просмотр и модификация таблицы сетевых маршрутов
neststat	Просмотр статистики текущих сетевых TCP/IP-соединений
arp	Просмотр и модификация ARP-таблицы
nslookup	Диагностика DNS-серверов
hostname	Просмотр имени хоста
ipconfig	Просмотр текущей конфигурации сети TCP/IP
nbtstat	Просмотр статистики текущих сетевых NBT-соединений
net	Управление сетью

Утилита ping. Как уже отмечалось раньше, ping в своей работе использует протокол ICMP и предназначена для проверки соединения с удаленным хостом.. Проверка соединения осуществляется путем посылки в

адрес хоста специальных ICMP-пакетов, которые в соответствии с протоколом должны быть возвращены, отправляющему хосту (эхо-пакеты и эхо-ответы).

Для получения справки о параметрах утилиты ping следует выполнить команду ping без параметров. В простейшем случае команда может быть применена с одним параметром:

ping hostname

где hostname – NetBIOS или DNS - имя хоста или его IP-адрес.

Утилит tracert. Как и утилита ping, tracert использует ICMP протокол для определения маршрута до пункта назначения. В результате работы утилиты на консоль выводятся все промежуточные узлы маршрута от исходного хоста до пункта назначения и время их прохождения.

Для получения справки о параметрах утилиты tracert следует выполнить команду tracert без параметров. В простейшем случае команда может быть применена с одним параметром:

tracert hostname

где hostname – NetBIOS или DNS - имя хоста или его IP-адрес.

Утилита route. Утилита route позволяет манипулировать таблицей сетевых маршрутов, которая имеется на каждом компьютере с TCP/IP-интерфейсом. Утилита обеспечивает выполнение четырех команд: print (распечатка таблицы сетевых маршрутов), add (добавить маршрут в таблицу), change (изменение существующего маршрута), delete (удаление маршрута).

Для получения справки о параметрах утилиты route следует выполнить команду route без параметров. В простейшем случае команда может быть использована для распечатки таблицы сетевых маршрутов:

route print

где параметр (команда) print, без уточняющих операндов, указывает на необходимость распечатки всей таблицы.

Утилита netstat. Утилита отражает состояние текущих TCP/IP-соединений хоста, а также статистику работы протоколов. С помощью утилиты netstat можно распечатать номера ожидающих портов всех соединений TCP/IP, имена исполняемых файлов, участвующих в подключениях, идентификаторы соответствующих Windows-процессов и т.д.

Для получения справки о параметрах утилиты netstat, следует выполнить следующую команду.

netstat -?

Активные соединения TCP/IP на компьютере можно просмотреть, набрав на консоли команду nestat с параметром -a.

netstat -a

Утилита arp. Утилита используется для просмотра и модификации ARP-таблицы, используемой для трансляции IP-адресов в адреса протоколов канального уровня (MAC-адреса). С помощью параметров команды можно распечатывать таблицу, удалять и добавлять данные ARP-таблицы. Корректировку ARP-таблицы может осуществлять только пользователь справами администратора.

Для получения справки о параметрах утилиты arp, следует выполнить команду arp без параметров. Получить текущее состояние ARP-таблицы можно с помощью следующей команды.

arp -a

Утилита nslookup. Утилита nslookup предназначена для проверки правильности работы DNS-серверов. С помощью утилиты, пользователь может выполнять запросы к DNS-серверам на получение адреса хоста по его DNS-имени, на получение адресов и имен почтовых серверов, ответственных за доставку почты для отдельных доменов DNS, на получение почтового адреса администратора DNS-сервера и т.д. и т.п. Утилита работает в двух режимах: в режиме однократного выполнения (при запуске в командной строке задается полный набор параметров) и в интерактивном режиме (команды и параметры задаются в режиме диалога). Запуск утилиты в интерактивном режиме осуществляется запуском команды nslookup без параметров.

```
C:\Documents and Settings\Administrator>nslookup
Default Server:  ns1.iptel.by
Address:  80.94.225.5

> www.google.com
Server:  ns1.iptel.by
Address:  80.94.225.5

Non-authoritative answer:
Name:    www.l.google.com
Addresses:  209.85.129.147, 209.85.129.99, 209.85.129.104
Aliases:  www.google.com

> www.oracle.com
Server:  ns1.iptel.by
Address:  80.94.225.5

Non-authoritative answer:
Name:    www.oracle.com
Address:  141.146.8.66

> exit
```

Рисунок 2.9.1. Пример выполнения команды nslookup

На рисунке 2.9.1 демонстрируется работа с утилитой nslookup. Сразу после запуска команды на консоль выводится имя хоста и IP-адрес активного DNS-сервера. После этого в диалоговом режиме были получены IP-адреса хостов с именами www.google.com, www.oracle.com и для завершения работы утилиты была введена команда exit.

Утилита hostname. Утилита предназначена для вывода на консоль имени хоста, на котором выполняется данная команда. Команда hostname не имеет никаких параметров.

Утилита ipconfig. Утилита ipconfig является наиболее востребованной сетевой утилитой. С ее помощью можно определить конфигурацию IP-интерфейса и значения всех сетевых параметров. Особенно эта утилита полезна на компьютерах, работающих с протоколом DHCP: команда позволяет проверить параметры IP-интерфейсов установленные в автоматическом режиме.

Для получения справки о параметрах утилиты следует ввести следующую команду.

```
ipconfig /?
```

Короткий отчет о конфигурации TCP/IP можно получить выдав команду ipconfig без параметров. Для получения полного отчета, можно использовать ключ /all, как это сделано на рисунке 2.9.2.

```
C:\Documents and Settings\Administrator>ipconfig /all

Настройка протокола IP для Windows

        Имя компьютера  . . . . . . . . . : SmeloW
        Основной DNS-суффикс  . . . . . . :
        Тип узла. . . . . . . . . . . . . : гибридный
        IP-маршрутизация включена . . . . : нет
        WINS-прокси включен . . . . . . . : нет

Подключение по локальной сети - Ethernet адаптер:

        DNS-суффикс этого подключения . . :
        Описание  . . . . . . . . . . . . : Broadcom 440x 10/100 Integrated Cont
roller
        Физический адрес. . . . . . . . . : 40-04-29-03-20-24
        Dhcp включен. . . . . . . . . . . : да
        Автонастройка включена  . . . . . : да
        IP-адрес автонастройки. . . . . . : 169.254.152.66
        Маска подсети . . . . . . . . . . : 255.255.0.0
        Основной шлюз . . . . . . . . . . :
        Основной WINS-сервер  . . . . . . : 169.254.152.66
```

Рисунок 2.9.2. Пример выполнения команды ipconfig

Утилита nbtstat. Утилита nbtstat позволяет просматривать статистику текущих соединений, использующих протокол NBT (NetBIOS over TCP/IP). Утилита в чем-то подобна утилите netstat, но применительно к протоколу NBT. Для получения справки о параметрах команды, необходимо ее выполнить без указания параметров.

Утилита net. Утилита net является основным средством управления сетью для сетевого клиента Windows. Команду net часто включают в скрипты регистрации и командные файлы. С помощью этой команды можно зарегистрировать пользователя в рабочей группе Windows, можно осуществить выход из сети, запустить или остановить сетевой сервис, управлять списком имен, пересылать сообщения в сети, синхронизировать время и т.д.

Для вывода списка параметров (команд) утилиты net следует выполнить следующую команду.

```
net help
```

Справка может быть уточнена для каждого отдельного параметра команды. Например, для того, чтобы получить справку для параметра send (пересылка сообщений в сети) следует добавить соответствующий параметр.

```
net help send
```

2.10. Настройка TC/IP в Windows

При установке последних версий операционной системы Windows, поддержка протокола TCP/IP включается автоматически. Работа системного администратора заключается только в настройке параметров сетевого подключения.

Дальнейшее изложение процесса настройки параметров TCP/IP, в основном ориентировано операционную систему Windows XP.

Для настройки параметров в Windows XP, следует использовать окно ***Сетевое подключение*** (Пуск/Панель Управления/Сетевые подключения). Выбрав необходимое сетевое подключение, из предлагаемого списка, получим окно, аналогичное изображенному на рисунке 2.10.1.

Если протокол TCP/IP отсутствует в данном подключении, то установка его может быть выполнена с помощью кнопки **Установить.** Далее будет предложен список компонентов сети для установки.

Если протокол TCP/IP есть, то для его настройки требуется перейти по клавише **Свойства** в окно **Свойства: Internet Protocol (TCP/IP)**. В этом окне требуется ответить на следующие вопросы и заполнить соответчвующие поля.

1. ***Получить IP-адрес автоматически или использовать фиксированный адрес***. Установка признака автоматического получения адреса, подразумевает применение протокола DHCP. Если же адрес фиксируется, то его следует указать в соответствующем поле. Кроме того, в случае фиксированного адреса необходимо установить маску подсети и адрес основного шлюза (шлюз по

умолчанию). В случае автоматического получения IP-адреса, маска подсети и адрес основного шлюза устанавливается по протоколу DHCP.

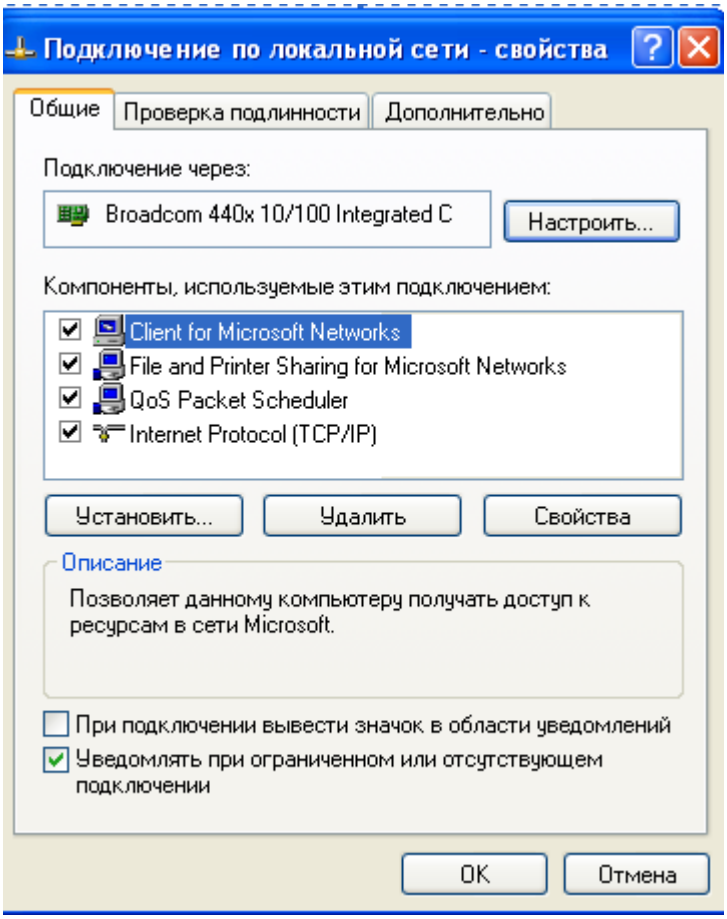

Рисунок 2.10.1 Окно для настройки параметров TCP/IP

2. ***Получить адрес DNS-сервера автоматически или использовать фиксированный адрес.*** Аналогично предыдущему случаю, автоматическая установка подразумевает использования протокола DHCP, который устанавливает этот параметр при подключении компьютера к сети. При установке фиксированного адреса DNS-сервера (и адреса запасного DNS-сервера), для разрешения имен будет использоваться, именно тот DNS-сервер, адрес которого будет указан в соответствующем поле окна.

Более подробно процесс установки и настройки протокола TCP/IP изложен в [7,10,11].

2.11. Итоги главы

1. Стек протоколов TCP/IP является основным стандартом, используемый для взаимодействия распределенных в сети компонентов программных систем. TCP/IP поддерживается большинством современных операционных систем и применяется

практически во всех системных и прикладных программных системах, работающих в сети. Описание всех протоколов TCP/IP содержится в документах именуемых RFC и поддерживаемых группой IETF.

2. Модель TCP/IP является четырехуровневой и в основном согласуется с моделью ISO/OSI. На Уровне доступа к сети используются протоколы, обеспечивающие создание локальных сетей или соединений с глобальными сетями. Наиболее популярными протоколами этого уровня являются Ethernet и PPP. Основным протоколом Межсетевого уровня является IP, но используются и другие протоколы: ICMP (для транспортировки служебной информации и диагностики), ARP (для разрешения MAC-адресов) и т.д. Транспортный уровень обеспечивается протоколами UDP (ненадежный протокол, ориентированный на сообщения) и TCP (надежный протокол ориентированный на соединение). На Прикладной уровне находятся службы TCP/IP и прикладные системы пользователей.
3. Протоколу IP (или точнее IPv4) в семействе протоколов TCP/IP отведена центральная роль. Его основной задачей является доставка дейтаграмм. Протокол считается ненадежным и не поддерживающим соединения. Для доступа к хостам IP использует собственную систему адресации. IP-адресом является последовательность из 32 битов, состоящая из адреса сети и адреса хоста в данной сети. Количество бит отведенных на адрес сети и на адрес хоста зависит от используемой модели адресации.
4. Новая версия протокола IP называется IPv6. Главным отличием протокола IPv6 является применение 128-битных адресов, что позволяет сделать все IP-адреса уникальными в глобальном смысле.
5. Протоколы Транспортного уровня являются пограничными протоколами между прикладной программной системой и стеком протоколов TCP/IP. Любой прикладной процесс протоколы Транспортного уровня идентифицируют с помощью номера порта. Все номера портов разбиты на три группы: хорошо известные номера портов (отводятся для базовых служб TCP/IP), зарегистрированные номера портов (используются известными промышленными системами) и динамически распределяемые или эфемерные порты (выделяются операционной системой динамически, по мере необходимости).
6. Для идентификации прикладного процесса в сети используется понятие сокета. Сокет – это совокупность IP-адреса и номера порта. Кроме того, различают сокеты UDP и сокеты TCP.
7. Большинство реализаций TCP/IP поддерживают интерфейс внутренней петли, позволяющий процессам, находящимся на одном хосте, обмениваться данными по протоколу TCP/IP. При этом дейтаграммы не выходят за пределы TCP/IP-интерфейса хоста. Внутренняя петля применяется, в основном, для отладки распределенных приложений.

С ее помощью на одном компьютере можно смоделировать работу процессов в сети.

8. Для доступа прикладных процессов к процедурам TCP/IP, операционные системы предоставляют специальные API. Наиболее распространенными программными интерфейсами являются API сокетов и интерфейс RPC. API сокетов описан в стандарте POSIX и поддерживается большинством операционных систем. Наиболее распространенной версией RPC является версия RPC Sun Microsystems. Развитием технологии RPC в Windows являются технологии COM и DCOM, а в Java-технологиях механизм RMI.
9. Программную реализацию протоколов Прикладного уровня TCP/IP называют службами. Службы реализуются в виде серверов, предоставляющих услуги другим процессам. Наиболее часто используемыми службами являются: DHCP, DNS, NBT, Telnet, FTP, WWW (протокол HTTP) и служба электронной почты (протоколы SMTP, POP3, IMAP4).
10. Для диагностики TCP/IP и управления сетью используются специальные программы, называемые сетевыми утилитами. Перечень утилит и их параметры зависят от конкретной реализации TCP/IP.
11. Стек протоколов TCP/IP, как правило, устанавливается на компьютер вместе с операционной системой. Работа системного администратора сводится к достаточно простой настройке TCP/IP.

Глава 3. Основы интерфейса Windows Sockets

3.1. Предисловие к главе

Как уже отмечалось раньше, в основе интерфейса Windows Sockets лежит интерфейс сокетов BSD Unix и стандарт POSIX, определяющий взаимодействие прикладных программ с операционной системой.

Интерфейс Windows Sockets акцентирован, прежде всего, на работу в сети TCP/IP, но обеспечивает обмен данными и по некоторым другим протоколам, например, ***IPX/SPX*** (стек протоколов операционной системы NetWare, компании Novell).

Ниже будет рассказано, как составлять сетевые приложения на языке программирования C++ с использованием интерфейса Windows Socket для протокола TCP/IP.

3.2. Версии, структура и состав интерфейса Windows Sockets

Существует две основные версии интерфейса: Windows Sockets 1.1 и Windows Sockets 2. В состав каждой версии входит динамическая библиотека, библиотека экспорта и заголовочный файл, необходимый для работы с библиотеками. Интерфейс версии 1.1 имеет две реализации: для 16-битовых и 32-битовых приложений.

Дальнейшее изложение интерфейса Windows Sockets ориентировано на версию 2, которую далее для краткости будем называть просто Winsock2. Полное описание функций Winsock2 содержится в документации, которая поставляется в составе SDK (Software Developer Kit) для программного интерфейса WIN32 или в MSDN.

Для использования интерфейса Winsock2 в исходный текст программы следует включить следующую последовательность директив компилятора C++.

```
//.....................................................................
#include "Winsock2.h"                    // заголовок  WS2_32.dll
#pragma comment(lib, "WS2_32.lib")       // экспорт  WS2_32.dll
//.....................................................................
```

Динамическая библиотека WS2_32.DLL (которая содержит все функции Winsock2), входит в стандартную поставку Windows, а библиотека экспорта WS2_32.LIB и заголовочный файл Winsock2.h в стандартную поставку Visual C++. С принципами построения и использования динамических библиотек (DLL) можно ознакомиться в [4, 12].

В таблице 3.2.1 приведен список функций интерфейса Windows. Список включает не все функции Winsock2 – здесь перечислены, только функции, которые будут применяться в дальнейших примерах. Кроме того, описания этих функций, не будет полным. Описания предназначены только для решения рассматриваемых в пособии задач. С полным описанием можно ознакомиться, например, на сайте www.microsoft.com.

Таблица 3.2.1

Наименование функции	Назначение
accept	Разрешить подключение к сокету
bind	Связать сокет с параметрами
closesocket	Закрыть существующий сокет
connect	Установить соединение с сокетом
gethostbyaddr	Получить имя хоста по его адресу
gethostbyname	Получить адрес хоста по его имени
gethostname	Получить имя хоста
getsockopt	Получить текущие опции сокета
htonl	Преобразовать u_long в формат TCP/IP
htons	Преобразовать u_short в формат TCP/IP
inet_addr	Преобразовать символьное представление IPv4-адреса в формат TCP/IP
inet_ntoa	Преобразовать сетевое представление IPv4-адреса в символьный формат
ioctlsocket	Установить режим ввода-вывода сокета
listen	Переключить сокет в режим прослушивания
ntohl	Преобразовать в u_long из формата TCP/IP
ntohs	Преобразовать в u_short из формата TCP/IP
recv	Принять данные по установленному каналу
recvfrom	Принять сообщение
send	Отправить данные по установленному каналу
sendto	Отправить сообщение
setsockopt	Установит опции сокета
socket	Создать сокет
TransmitFile	Переслать файл
TransmitPackets	Переслать область памяти
WSACleanup	Завершить использование библиотеки WS2_32.DLL
WSAGetLastError	Получить диагностирующий код ошибки
WSAStartup	Инициализировать библиотеку WS2_32.DLL

3.3. Коды возврата функций интерфейса Windows Sockets

Все функции интерфейса Winsock2 могут завершаться успешно или с ошибкой. При описании каждой функции будет указано, каким образом можно проверить успешность ее завершения. В том случае, если функция завершает свою работу с ошибкой, формируется дополнительный диагностирующий код, позволяющий уточнить причину ошибки.

Диагностирующий код может быть получен с помощью функции WSAGetLastError. Функция WSAGetLastError вызывается, непосредственно сразу после функции Winsock2, завершившейся с ошибкой. Все

диагностирующие коды представлены в таблице 3.3.1. Описание функции приводится на рисунке 3.3.1. На рисунке 3.3.2 приведен пример использования функции WSAGetLastError.

```
// -- Получить диагностирующий код ошибки
// Назначение: функция позволяет определить причину
//             завершения функций Winsock2 с ошибкой

int WSAGetLastError(void);       // прототип функции

// Код возврата: функция возвращает диагностический код
```

Рисунок 3.3.1. Функция WSAGetLastError

```
//.....................................................................
#include "Winsock2.h"
#pragma comment(lib, "WS2_32.lib")
//.....................................................................
string  GetErrorMsgText(int code)     // сформировать текст ошибки
{
  string msgText;
  switch (code)                       // проверка кода возврата
   {
   case WSAEINTR:          msgText = "WSAEINTR";          break;
   case WSAEACCES:         msgText = "WSAEACCES";         break;
   //.........коды WSAGetLastError .........................
   case WSASYSCALLFAILURE: msgText = "WSASYSCALLFAILURE";break;
   default:                msgText = "***ERROR***";       break;
   };
  return msgText;
};
string  SetErrorMsgText(string msgText, int code)
{return  msgText+GetErrorMsgText(code);};

int main(int argc, _TCHAR* argv[])
{
 //.....................................................................
 try
   {
   //.....................................................................
   if ((sS = socket(AF_INET, SOCK_STREAM, NULL))== INVALID_SOCKET)
             throw  SetErrorMsgText("socket:",WSAGetLastError());
   //.....................................................................
   }
 catch (string errorMsgText)
     { cout<< endl << "WSAGetLastError: " << errorMsgText;}

 //.....................................................................
 return 0;
}
```

Рисунок 3.3.2. Пример использования функции WSAGetLastError

В приведенном примере для обработки ошибок используется функция SetErrorMsgText, которая в качестве параметра получает префикс формируемого сообщения об ошибке, код функции WSAGetLastError, а возвращает текст сообщения (используя функцию GetErrorMsgText).

Таблица 3.3.1

Коды возврата функции GetLastError	**Причина ошибки**
WSAEINTR	Работа функции прервана
WSAEACCES	Разрешение отвергнуто
WSAEFAULT	Ошибочный адрес
WSAEINVAL	Ошибка в аргументе
WSAEMFILE	Слишком много файлов открыто
WSAEWOULDBLOCK	Ресурс временно недоступен
WSAEINPROGRESS	Операция в процессе развития
WSAEALREADY	Операция уже выполняется
WSAENOTSOCK	Сокет задан неправильно
WSAEDESTADDRREQ	Требуется адрес расположения
WSAEMSGSIZE	Сообщение слишком длинное
WSAEPROTOTYPE	Неправильный тип протокола для сокета
WSAENOPROTOOPT	Ошибка в опции протокола
WSAEPROTONOSUPPORT	Протокол не поддерживается
WSAESOCKTNOSUPPORT	Тип сокета не поддерживается
WSAEOPNOTSUPP	Операция не поддерживается
WSAEPFNOSUPPORT	Тип протоколов не поддерживается
WSAEAFNOSUPPORT	Тип адресов не поддерживается протоколом
WSAEADDRINUSE	Адрес уже используется
WSAEADDRNOTAVAIL	Запрошенный адрес не может быть использован
WSAENETDOWN	Сеть отключена
WSAENETUNREACH	Сеть не достижима
WSAENETRESET	Сеть разорвала соединение
WSAECONNABORTED	Программный отказ связи
WSAECONNRESET	Связь восстановлена
WSAENOBUFS	Не хватает памяти для буферов
WSAEISCONN	Сокет уже подключен
WSAENOTCONN	Сокет не подключен
WSAESHUTDOWN	Нельзя выполнить send: сокет завершил работу
WSAETIMEDOUT	Закончился отведенный интервал времени
WSAECONNREFUSED	Соединение отклонено
WSAEHOSTDOWN	Хост в неработоспособном состоянии
WSAEHOSTUNREACH	Нет маршрута для хоста
WSAEPROCLIM	Слишком много процессов

Таблица 3.3.1 (продолжение)

Коды возврата функции GetLastError	Причина ошибки
WSASYSNOTREADY	Сеть не доступна
WSAVERNOTSUPPORTED	Данная версия недоступна
WSANOTINITIALISED	Не выполнена инициализация WS2_32.DLL
WSAEDISCON	Выполняется отключение
WSATYPE_NOT_FOUND	Класс не найден
WSAHOST_NOT_FOUND	Хост не найден
WSATRY_AGAIN	Неавторизированный хост не найден
WSANO_RECOVERY	Неопределенная ошибка
WSANO_DATA	Нет записи запрошенного типа
WSA_INVALID_HANDLE	Указанный дескриптор события с ошибкой
WSA_INVALID_PARAMETER	Один или более параметров с ошибкой
WSA_IO_INCOMPLETE	Объект ввода-вывода не в сигнальном состоянии
WSA_IO_PENDING	Операция завершится позже
WSA_NOT_ENOUGH_MEMORY	Не достаточно памяти
WSA_OPERATION_ABORTED	Операция отвергнута
WSAINVALIDPROCTABLE	Ошибочный сервис
WSAINVALIDPROVIDER	Ошибка в версии сервиса
WSAPROVIDERFAILEDINIT	Невозможно инициализировать сервис
WSASYSCALLFAILURE	Аварийное завершение системного вызова

Комментарии с точками в приведенном примере и дальше будут использоваться для обозначения того, что тексты программ не являются законченными и предназначены только для демонстрации использования функций.

3.4. Схемы взаимодействия процессов в распределенном приложении

Существование двух различных протоколов на транспортном уровне TCP/IP, определяет две схемы взаимодействия процессов распределенного приложения: схема, ориентированная на сообщения, и схема, ориентированная на поток.

Принципиальное различие этих схем, заключается в следующем.

В первом между сокетами курсируют UDP-пакеты, и поэтому вся работа, связанная с обеспечением надежности и установкой правильной последовательности передаваемых пакетов возлагается на само приложение. В общем случае, получатель узнает адрес отравителя вместе с пакетом данных.

Во втором случае между сокетами устанавливается TCP-соединение и весь обмен данных осуществляется в рамках этого соединения. Передача по каналу является надежной и данные поступают в порядке их отправления.

В распределенных приложениях архитектуры клиент-сервер, клиенту и серверу отводится разная роль: инициатором обмена является клиент, а

сервер ждет запросы клиента и обслуживает их. Таким образом, предполагается, что к моменту выдачи запроса клиентом, сервер должен быть уже активным, а клиент должен “знать” параметры сокета сервера. На рисунках 3.4.1 и 3.4.2. изображены схемы взаимодействия клиента и сервера, для первого и второго случаев.

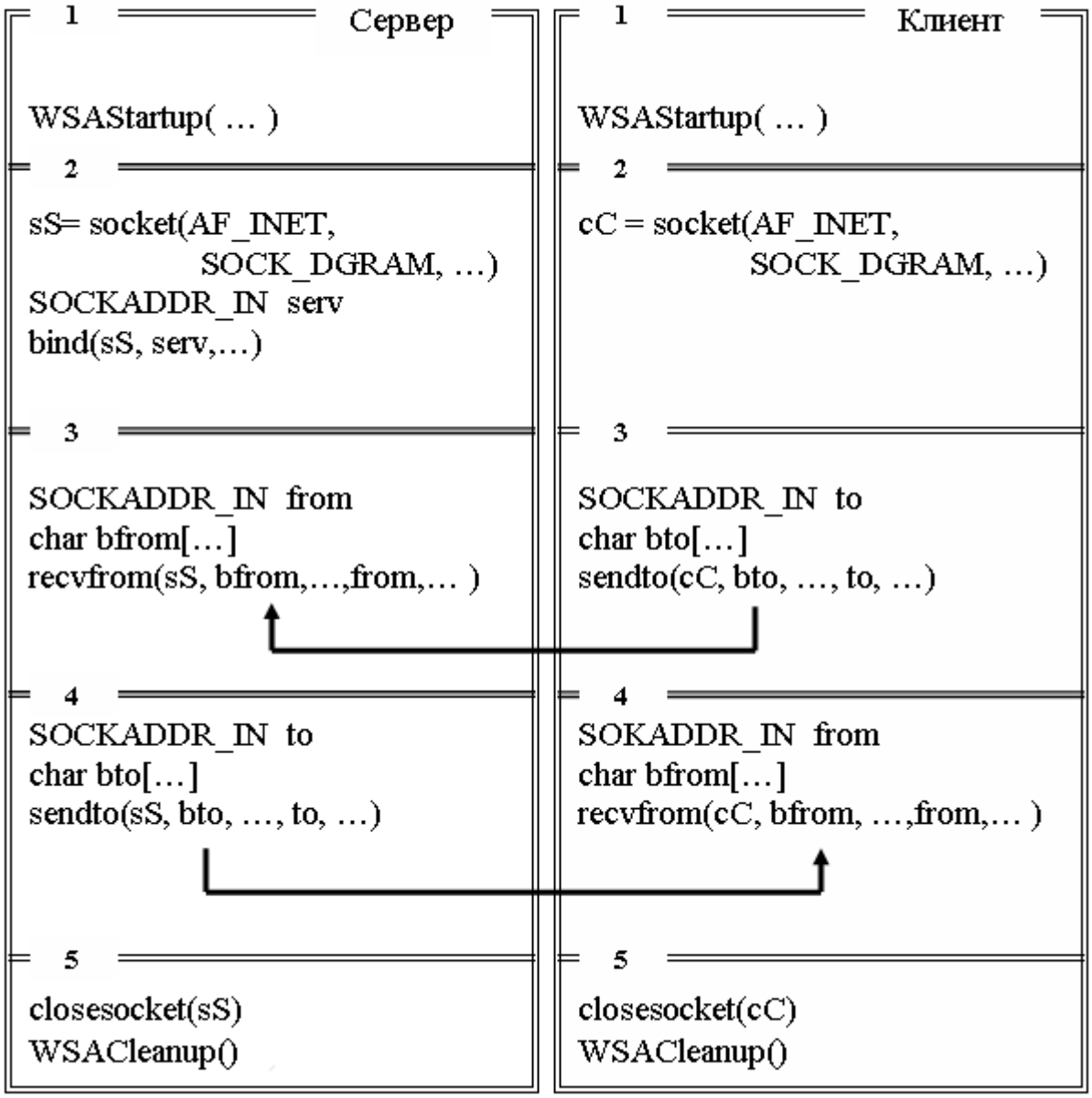

Рисунок 3.4.1. Схема взаимодействия процессов без установки соединения

На рисунке 3.4.1 схематично изображены две программы, реализующие два процесса распределенного приложения. Рассматриваемое приложение имеет архитектуру клиент-сервер (на рисунке сделаны соответствующие обозначения). Обе программы разбиты на пять блоков, а стрелками обозначается движение информации по сети TCP/IP.

Первые блоки обеих программ одинаковые и предназначены для инициализации библиотеки WS2_32.DLL.

Второй блок программы сервера создает сокет (функция socket) и устанавливает параметры этого сокета. Следует обратить внимание на параметр SOCK_DGRAM функции socket, указывающий на тип сокета (в данном случае – сокет, ориентированный на сообщения). Для установки параметров сокета, используется функция bind. При этом говорят, что сокет ***связывают*** с параметрами. Для хранения параметров сокета в Winsock2 предусмотрена специальная структура SOCKADDR_IN (она тоже присутствует на рисунке). Перед выполнением функции bind, которая использует эту структуру в качестве параметра, необходимо ее заполнить данными. Пока скажем только, что в SOCKADDR_IN хранится IP-адрес и номер порта сервера.

В третьем блоке программы сервера выполняется функция recvfrom, которая переводит программу сервера в состояние ожидания, до поступления сообщения от программы клиента (функция sendto). Функция recvfrom тоже использует структуру SOCKADDR_IN – в нее автоматически помещаются параметры сокета клиента, после приема от него сообщения. Данные поступают в буфер, который обеспечивает принимающая сторона (на рисунке символьный массив bfrom). Следует отметить, что в качестве параметра функции recvfrom используется связанный сокет – именно через него осуществляется передача данных.

Четвертый блок программы сервера предназначен для пересылки данных клиенту. Пересылка данных осуществляется с помощью функции sendto. В качестве параметров sendto использует структуру SOCKADDR_IN с параметрами сокета принимающей стороны (в данном случае клиента) и заполненный буфер с данными.

Пятые блоки программ сервера и клиента одинаковые и предназначены для закрытия сокета и завершения работы с библиотекой WS2_32.DLL.

Всем блокам программы клиента, кроме второго, есть аналог в программе сервера. Второй блок, в сравнении с сервером, не использует команду bind. Здесь проявляется основное отличие между сервером и клиентом. Если сервер, должен использовать однозначно определенные параметры (IP-адрес и номер порта), то для клиента это не обязательно – ему Windows выделяет эфемерный порт. Т.к. инициатором связи является клиент, то он должен точно “знать” параметры сокета сервера, а свои параметры клиент получит от Windows и сообщит их вместе с переданным пакетом серверу.

Взаимодействие программ клиента и сервера в случае установки соединения схематично изображено на рисунке 3.4.2. Как и в предыдущем случае обе программы разбиты на блоки. Сплошными направленными линиями обозначается движение данных по сети TCP/IP, прерывистой – синхронизация (ожидание) процессов.

Первые блоки обеих программ идентичны и предназначены для инициализации библиотеки WS2_32.DLL.

Второй блок сервера имеет то же предназначение, что и в предыдущем случае. Единственным отличием является значение SOCK_STREEM параметра функции socket, указывающий, что сокет будет использоваться для соединения (сокет ориентированный на поток).

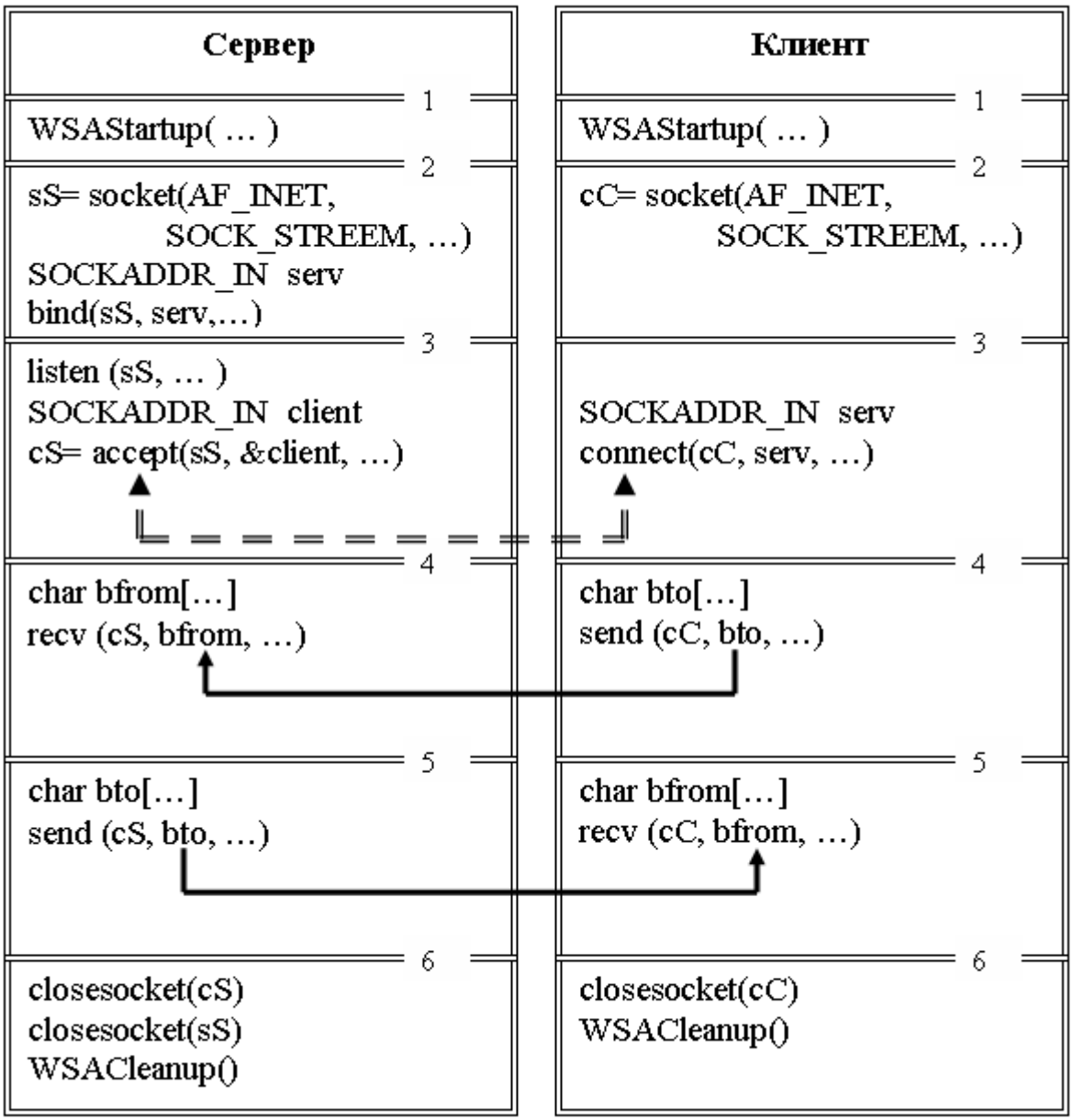

Рисунок 3.4.2. Схема взаимодействия процессов с установкой соединения

В третьем блоке программы сервера выполняются две функции Winsock2: listen и accept. Функция listen переводит сокет, ориентированный на поток, в состояния прослушивания (открывает доступ к сокету) и задает некоторые параметры очереди соединений. Функция accept переводит процесс сервера в состояние ожидания, до момента пока программа клиента не выполнит функцию connect (подключится к сокету). Если на стороне клиента корректно выполнена функция connect, то функция accept

возвращает новый сокет (с эфемерным портом), который предназначен для обмена данными с подключившимся клиентом. Кроме того, автоматически заполняется структура SOCKADDR_IN параметрами сокета клиента.

Четвертый и пятый блоки программы сервера предназначены для обмена данными по созданному соединению. Следует обратить внимание, что, во-первых, используются функции send и recv, а во-вторых, в качестве параметра эти функции используют сокет, созданный командой accept.

В программе клиента осталось пояснить, только работу третьего блока. В этом блоке выполняется функция connect, предназначенная для установки соединения с сокетом сервера. Функция в качестве параметров имеет, созданный в предыдущем блоке, дескриптор сокета (ориентированного на поток) и структуру SOCKADDR_IN с параметрами сокета сервера.

3.5. Инициализация библиотеки Windows Sockets

Для инициализации библиотеки WS2_32.DLL предназначена функция WSAStartup. Описание функции приводится на рисунке 3.5.1.

```
// -- инициализировать библиотеку WS2_32.DLL
// Назначение: функция  позволяет инициализировать
//             динамическую библиотеку, проверить номер
//             версии, получить сведения о конкретной
//             реализации библиотеки. Функция должна быть
//             выполнена до использования любой функции
//             Windows Sockets
//
   int WSAStartup(
          WORD          ver,  //[in]  версия  Windows Sockets
          lpWSAData     wsd   //[out] указатель на WSADATA
                );
// Код возврата: в случае успешного завершения функция
//               возвращает нулевое значение, в случае ошибки
//               возвращается не нулевое  значение
// Примечания: - параметр ver представляет собой два байта,
//               содержащих номер версии Windows Sockets,
//               причем.  старший байт содержит
//               младший номер версии, а младший байт-
//               старший номер версии;
//             - обычно параметр ver задается с помощью макро
//               MAKEWORD;
//             - шаблон структуры WSADATA содержится в
//               Winsock2.h
```

Рисунок 3.5.1. Функция WSAStartup

Как уже отмечалось раньше, функция WSAStartup должна быть выполнена до использования любых функция Winsock2. Пример, использования функции будет приведен ниже.

3.6. Завершение работы с библиотекой Windows Sockets

Для завершения работы с библиотекой WS2_32.DLL используется функция WSACleanup. Описание функции приводится на рисунке 3.6.1.

```
// -- завершить  работу с библиотекой  WS2_32.DLL
// Назначение: функция завершает работу с динамической
//             библиотекой WS2_32.DLL, делает недоступным
//             выполнение функций библиотеки, освобождает
//             ресурсы.
//
   int WSACleanup(void);

// Код возврата: в случае успешного завершения функция
//               возвращает нулевое значение, в случае ошибки
//               возвращается SOCKET_ERROR
```

Рисунок 3.6.1. Функция WSACleanup

На рисунке 3.6.2 приводится пример использования функций WSAStatrup и WSACleanup.

```
//.....................................................
#include "Winsock2.h"
#pragma comment(lib, "WS2_32.lib")

int _tmain(int argc, _TCHAR* argv[])
{
  WSADATA wsaData;
  try
  {
   if (WSAStartup(MAKEWORD(2,0), &wsaData) != 0)
            throw  SetErrorMsgText("Startup:",WSAGetLastError());
    //.....................................................
   if (WSACleanup()== SOCKET_ERROR)
            throw  SetErrorMsgText("Cleanup:",WSAGetLastError());
  }
  catch (string errorMsgText)
  { cout<< endl << errorMsgText;}
return 0;
}
```

Рисунок 3.6.2. Пример использования функций WSAStartup и WSACleanup

3.7. Создание сокета и закрытие сокета

Для создания сокета используется функция socket. Описание функции приводится на рисунке 3.7.1.

```
// -- создать сокет
// Назначение: функция  позволяет создать сокет (точнее
//             дескриптор сокета) и задать его характеристики
//
  SOCKET socket(
            int    af,   //[in]  формат адреса
            int    type, //[in]  тип сокета
            int    prot  //[in]  протокол
               );
// Код возврата: в случае успешного завершения функция
//               возвращает дескриптор сокета, в другом
//               случае возвращается INVALID_SOCKET
// Примечания: - параметр af для стека TCP/IP принимает
//               значение AF_INET;
//             - параметр type может принимать два значения:
//               SOCK_DGRAM – сокет, ориентированный на
//               сообщения(UDP); SOCK_STREEM – сокет
//               ориентированный на поток;
//               старший номер версии;
//             - параметр prot определяет  протокол
//               транспортного уровня: для TCP/IP можно
//               указать NULL
```

Рисунок 3.7.1. Функция socket

После завершения работы с сокетом, обычно, его закрывают (освобождают ресурс). Для закрытия сокета применяется функция closesocket. Описание этой функции приводится на рисунке 3.7.2.

```
// -- закрыть существующий  сокет
// Назначение: переводит сокет в неработоспособное состояние и
//             освобождает все ресурсы связанные с ним
//
  SOCKET closesocket(
             SOCKET s,    //[in] дескриптор сокета
              );
// Код возврата: в случае успешного завершения функция
//               возвращает нуль, в другом случае
//               возвращается SOCKET_ERROR
```

Рисунок 3.7.2. Функция closesocket

На рисунке 3.7.3. приводится пример программы использующей функции socket и closesocket.

```
//.........................................................
#include "Winsock2.h"
#pragma comment(lib, "WS2_32.lib")

int _tmain(int argc, _TCHAR* argv[])
{
  SOCKET  sS;           // дескриптор сокета
  WSADATA wsaData;
  try
  {
   if (WSAStartup(MAKEWORD(2,0), &wsaData) != 0)
          throw  SetErrorMsgText("Startup:",WSAGetLastError());
   if ((sS = socket(AF_INET, SOCK_STREAM, NULL))== INVALID_SOCKET)
          throw  SetErrorMsgText("socket:",WSAGetLastError());
   //.....................................................
   if (closesocket(sS)== SOCKET_ERROR)
          throw  SetErrorMsgText("closesocket:",WSAGetLastError());
   if (WSACleanup()== SOCKET_ERROR)
          throw  SetErrorMsgText("Cleanup:",WSAGetLastError());
  }
  catch (string errorMsgText)
  { cout << endl << errorMsgText;}
return 0;
}
```

Рисунок 3.7.3. Пример использования функций socket и closesocket

3.8. Установка параметров сокета

Для установки параметров существующего сокета используется функция bind. Описание функции приводится на рисунке 3.8.1.

```
// -- связать сокет с параметрами
// Назначение: функция связывает существующий сокет с
//             с параметрами, находящимися в структуре
//             SOCKADDR_IN
//
 int bind(
     SOCKET s,                    //[in] сокет
     cost struct sockaddr_in*  a, //[in] указатель на SOCKADDR_IN
     int    la                    //[in] длина SOCKADDR_IN в байтах
          )
// Код возврата: в случае успешного завершения функция
//               возвращает нуль, в случае ошибки
//               возвращается SOCKET_ERROR
```

Рисунок 3.8.1. Функция bind

Функция связывает дескриптор сокета и структуру SOCKADDR_IN, которая предназначена для хранения параметров сокета. IШаблон структуры SOCKADDR_IN содержится в файле Winsock2.h. Описание структуры SOCKADDR_IN и используемых вместе с ней констант приводится на рисунке 3.8.2. Особое внимание следует обратить внимания на строки, отмеченные тремя знаками плюс. В дальнейшем отмеченные поля и константы будут использоваться в текстах программ. IP-адрес и номер порта в структуре SOCKADDR_IN хранятся в специальном сетевом формате. Этот формат отличается, от формата компьютеров с архитектурой Intel. В составе Winsock2 имеются функции, позволяющие преобразовывать форматы данных.

```
#define INADDR_ANY        (u_long)0x00000000 //любой адрес       +++
#define INADDR_LOOPBACK    0x7f000001        // внутренняя петля +++
#define INADDR_BROADCAST  (u_long)0xffffffff // широковещание    +++
#define INADDR_NONE        0xffffffff        // нет адреса
#define ADDR_ANY           INADDR_ANY        // любой адрес

struct in_addr
{                    // IP-адрес
    union {
     struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
     struct { u_short s_w1,s_w2; }          S_un_w;
     u_long                                 S_addr;
          }                                 S_un;
     #define s_addr  S_un.S_addr    // 32-битный IP-адрес         +++
     #define s_host  S_un.S_un_b.s_b2
     #define s_net   S_un.S_un_b.s_b1
     #define s_imp   S_un.S_un_w.s_w2
     #define s_impno S_un.S_un_b.s_b4
     #define s_lh    S_un.S_un_b.s_b3
}

struct sockaddr_in {
  short   sin_family;       //тип сетевого адреса                  +++
  u_short sin_port;         // номер порта                         +++
  struct  in_addr sin_addr; // IP-адрес                            +++
  char    sin_zero[8];      // резерв
                  };
typedef struct sockaddr_in SOCKADDR_IN;    //                      +++
typedef struct sockaddr_in *PSOCKADDR_IN;
typedef struct sockaddr_in FAR *LPSOCKADDR_IN;
```

Рисунок 3.8.2. Структура SOCKADDR_IN

Для преобразования номера порта в формат TCP/IP следует использовать функцию htons. Описание этой функции приведено на рисунке 3.8.3. Функция ntohs является обратной функцией, предназначена для преобразования двух байтов в формате TCP/IP в формат u_short.

```
// -- преобразовать  u_short в формат TCP/IP
// Назначение: функция преобразовывает два байта данных
//             формата u_short (unsigned  short) в два
//             два байта, сетевого формата
//
   u_short htons (
               u_short hp    //[in] 16 битов данных
                  );
//
// Код возврата: 16  битов в формате TCP/IP
//
```

Рисунок 3.8.3 Функция htons

Полезной является функция inet_addr, предназначенная для преобразования символьного представления IPv4-адреса в формат TCP/IP. Описание функции приведено на рисунке 3.8.4. Функция inet_ntoa предназначена для обратного преобразования из сетевого представления в символьный формат.

```
// -- преобразовать символьное представление IPv4-адреса в формат TCP/IP
// Назначение: функция преобразует общепринятое символьное
//             представление IPv4-адреса (n.n.n.n) в
//             четырехбайтовый IP-адрес в формате TCP/IP
//
  unsigned long inet_addr(
          const char* stra //[in] строка символов, закачивающаяся 0x00
                        );
//
// Код возврата: в случае успешного завершения функция
//               IP-адрес в формате TCP/IP, иначе
//               возвращается INADDR_NONE
```

Рисунок 3.8.4 Функция inet_addr

На рисунке 3.8.5 приведен фрагмент программы сервера. Функция bind связывает сокет с параметрами, заданными в структуре SOCKADDR_IN. Структура содержит три значения (параметры сокета): тип используемого адреса (константа AF_INET используется для обозначения семейства IP-адресов); номер порта (устанавливается значение 2000 с помощью функции htons) и адрес интерфейса . Последний параметр определяет собственный IP-адрес сервера. При этом предполагается, что хост, в общем случае, может иметь несколько IP-интерфейсов. Если требуется использовать определенный IP-интерфейс хоста, то необходимо его здесь указать. Если выбор IP-адреса не является важным или IP-интерфейс один на хосте, то следует указать значение INADDR_ANY (как это сделано в примере). Программа клиента для пересылки сообщений (обратите внимание, что при

создании сокета использовался параметр со значением SOCKET_DGRAM), должна их отправлять именно этому сокету (т.е. указывать его IP-адрес и его номер порта).

```
//.............................................................
SOCKET  sS;                                 // серверный сокет
if ((sS = socket(AF_INET, SOCK_DGRAM, NULL))== INVALID_SOCKET)
              throw  SetErrorMsgText("socket:",WSAGetLastError());

SOCKADDR_IN serv;                           // параметры  сокета sS
serv.sin_family  = AF_INET;                 // используется IP-адресация
serv.sin_port = htons(2000);                // порт 2000
serv.sin_addr.s_addr =  INADDR_ANY;         // любой собственный IP-адрес

if (bind(sS,(LPSOCKADDR)&serv, sizeof(serv))== SOCKET_ERROR)
             throw  SetErrorMsgText("bind:",WSAGetLastError());
//.............................................................
```

Рисунок 3.8.5. Пример использования функции bind

3.9. Переключение сокета в режим прослушивания

После создания сокета и выполнения функции bind сокет остается недоступным для подсоединения клиента. Чтобы сделать доступным уже связанный сокет, необходимо его переключить, в так называемый, прослушивающий режим. Переключение осуществляется с помощью функции listen (рисунок 3.9.1).

```
// -- переключить сокет в режим прослушивания
// Назначение: функция делает сокет доступным для подключений
//             и устанавливает максимальную длину очереди
//             подключений

int listen(
        SOCKET s,     //[in] дескриптор связанного сокета
         int    mcq,  //[in] максимальная длина очереди
          );

// Код возврата: при успешном завершении функция возвращает
//                нуль, иначе возвращается значение
//                SOCKET_ERROR
// Примечания:    для установки значения параметра mcq можно
//                использовать константу SOMAXCONN,позволяющую
//                установить максимально возможное значение
```

Рисунок 3.9.1. Функции listen

После выполнения функции listen клиентские программы могут осуществить подключение к сокету (выполнить функцию connect). Кроме

того, функция listen устанавливает максимальную длину очереди подключений. Если количество одновременно подключающихся клиентов превысит установленное максимальное значение, то последние попытки подключения потерпят неудачу (и функция connect на стороне клиента сформирует соответствующий код ошибки). Следует отметить, что функция listen применяется только для сокетов ориентированных на поток.

3.10. Создание канала связи

Канал связи (или соединение) создается между двумя сокетами, ориентированными на поток. На стороне сервера это должен быть связанный (функция bind) и переключенный в режим прослушивания (функция listen) сокет. На стороне клиента должен быть создан дескриптор ориентированного на поток сокета (функция socket).

Канал связи создается в результате взаимодействия функций accept (на стороне сервера) и connect (на стороне клиента). Алгоритм взаимодействия этих функций зависит от установленного режима ввода-вывода для участвующих в создании канала сокетов.

Winsock2 поддерживает два режима ввода вывода: blocked и nonblocked. Установить или изменить режим можно с помощью функции ioctlsocket. Дальнейшее изложение предполагает, что для сокетов установлен режим blocked (действующий по умолчанию), режим nonblocked будет рассматриваться отдельно.

```
// -- разрешить подключение к сокету
// Назначение: функция используется для создания канала на
//             стороне сервера и создает сокет для обмена
//             данными по  этому каналу

SOKET accept(
       SOCKET s,              // [in]  дескриптор связанного сокета
        struct sockaddr_in* a,//[out] указатель на SOCKADDR_IN
        int* la                //[out] указатель на длину SOCKADDR_IN
             );

// Код возврата: при успешном завершении функция возвращает
//               дескриптор нового сокета, предназначенного
//               для обмена данными по этому каналу, иначе
//               возвращается значение INVALID_SOCKET
// Примечания:   в случае успешного выполнения функции,
//               указатель a содержит адрес структуры
//               SOCKADDR_IN  с параметрами сокета,
//               осуществившего подключение (connect)сокета,
//               а указатель la содержит адрес 4-х байт с
//               длинной (в байтах) структуры SOCKADDR_IN
```

Рисунок 3.10.1 Функция accept

Функция accept (описание на рисунке 3.10.1) приостанавливает выполнение программы сервера до момента срабатывания в программе клиента функции connect (описание на рисунке 3.10.3). В результате работы функции accept создается новый сокет, предназначенный для обмена данными с клиентом. На рисунке 3.10.2 представлен пример использования функции accept.

```
//.....................................................................
try
  {
//...WSAStartup(...),sS = socket(...,SOCKET_STREAM,...),bind(sS,...)

   if (listen(sS,SOMAXCONN)== SOCKET_ERROR)
               throw  SetErrorMsgText("listen:",WSAGetLastError());

   SOCKET cS;                      // сокет для обмена данными с клиентом
   SOCKADDR_IN clnt;               // параметры  сокета клиента
   memset(&clnt,0,sizeof(clnt)); // обнулить память
   int lclnt  = sizeof(clnt);     // размер SOCKADDR_IN

   if ((cS = accept(sS,(sockaddr*)&clnt, &lclnt)) == INVALID_SOCKET)
               throw  SetErrorMsgText("accept:",WSAGetLastError());
//.....................................................................
  }
catch (string errorMsgText)
  {cout << endl << errorMsgText;}
//.....................................................................
```

Рисунок 3.10.2 Фрагмент программы сервера с функцией accept

```
// -- установить соединение с сокетом
// Назначение: функция используется клиентом для создания
//             канала с определенным сокетом сервера

int connect (
       SOCKET s,               // [in] дескриптор связанного сокета
        struct sockaddr_in* a, // [in] указатель на SOCKADDR_IN
        int  la                // [in] длина SOCKADDR_IN в байтах
             );

// Код возврата: при успешном завершении функция возвращает
//               нуль, иначе возвращается значение
//               SOCKET_ERROR
// Примечания: - параметр a является указателем на структуру
//               SOCKADDR_IN; структура должна быть
//               инициализирована параметрами серверного
//               сокета (тип адреса, IP-адрес, порт);
//             - параметр la, должен содержать длину
//               (в байтах) структуры SOCKADDR_IN
```

Рисунок 3.10.3 Функция connect

На стороне клиента создание канала осуществляется с помощью функции connect. Для того, чтобы выполнить функцию connect, достаточно просто предварительно создать сокет (функция socket), ориентированный на поток. Функция connect указывает модулю TCP сокет клиента, который будет использоваться для соединения с сокетом сервера (его параметры указываются через параметры connect). При этом предполагается, что серверный сокет создан (функции socket и bind) и для него уже выполнена функция listen. На рисунке 3.10.4 приведен фрагмент текста программы клиента, в котором используется функция connect.

```
//.............................................................
try
  {
  //....WSAStartup(...).......................................

   SOCKET  cC;                              // серверный сокет
   if ((cC = socket(AF_INET, SOCK_STREAM, NULL))== INVALID_SOCKET)
               throw  SetErrorMsgText("socket:",WSAGetLastError());

   SOCKADDR_IN serv;                        // параметры  сокета сервера
   serv.sin_family = AF_INET;               // используется IP-адресация
   serv.sin_port = htons(2000);                        // TCP-порт 2000
   serv.sin_addr.s_addr = inet_addr("80.1.1.7");  // адрес сервера
   if ((connect(cC,(sockaddr*)&serv, sizeof(serv))) == SOCKET_ERROR)
               throw  SetErrorMsgText("connect:",WSAGetLastError());
  //.............................................................
   }
 catch (string errorMsgText)
  { cout << endl << errorMsgText;}
 //.............................................................
```

Рисунок 3.10.4. Фрагмент программы клиента с функцией connect

3.11. Обмен данными по каналу связи

Обмен данными по каналу связи осуществляется между двумя сокетами и возможен сразу после того, как этот канал создан (выполнена функция accept на стороне сервера и функция connect на стороне клиента). Для пересылки данных по каналу Winsock2 предоставляет функции send и recv (рисунки 3.11.1 и 3.11.2). Функция send пересылает по каналу, указанного сокета, определенное количество байт данных. Функция recv принимает по каналу, указанного сокета, определенное количество байт данных. Для работы обеих функций в программе необходимо выделить память (буфер) для приема или оправления данных. Размер буфера для приема данных и для отправления данных указывается в параметрах функций. Реальное количество пересланных или принятых байт данных возвращается функциями send и recv в виде кода возврата.

Следует иметь в виду, что не всегда количество переданных или оправленных байт совпадает с размерами выходного или входного буферов.

Более того, разрешается выполнять пересылку с нулевым количеством байт. Обычно такую пересылку используют для обозначения конца передачи. Для принимающей стороны операционная система буферизирует принимаемые данные. Если при очередном приеме данных размеры буфера будут исчерпаны, отправляющей стороне будет выдано соответствующий код ошибки.

```
// -- отправить данные по установленному каналу
// Назначение: функция пересылает заданное  количество
//             байт данных по каналу определенного сокета

  int send (
       SOCKET s,         // [in] дескриптор сокета (канал для передачи)
       const char* buf, // [in] указатель буфер данных
       int  lbuf,        // [in] количество байт данных в буфере
       int  flags        // [in] индикатор особого режима маршрутизации
           );

// Код возврата: при успешном завершении функция возвращает
//               количество переданных байт данных, иначе
//               возвращается SOCKET_ERROR
// Примечания:   для параметра flags следует установить
//               значение NULL
```

Рисунок 3.11.1 Функция send

```
// -- принять данные по установленному каналу
// Назначение: функция принимает заданное  количество
//             байт данных по каналу определенного сокета

  int recv (
       SOCKET s,         // [in] дескриптор сокета (канал для приема)
       const char* buf, // [in] указатель буфер данных
       int  lbuf,        // [in] количество байт данных в буфере
       int  flags        // [in] индикатор
           );

// Код возврата: при успешном завершении функция возвращает
//               количество принятых  байт данных, иначе
//               возвращается SOCKET_ERROR
// Примечания:   параметр flags  определяет режим обработки
//               буфера: NULL -  входной буфер очищается
//               после считывания данных (рекомендуется),
//               MSG_PEEK – входной буфер не очищается
```

Рисунок 3.11.2 Функция recv

Работа функций send и recv является синхронной, т.е. до тех пор, пока не будет выполнена пересылка или прием данных выполнение программы приостанавливается. Поэтому, если одной из сторон распределенного будет выдана функция recv для которой нет данных для приема и при этом соединение не разорвано, то это приведет к зависанию программы на некоторое время (называемое time-out) и завершению функции recv с соответствующим кодом ошибки.

```
//.....................................................................
try
{
  //...WSAStartup(...),sS = socket(...,SOCKET_STREAM,...),bind(sS,...)
  //...listen(sS,...),  cS = accept(sS,...).........................

 char ibuf[50],                        //буфер ввода
      obuf[50]= "sever: принято ";     //буфер вывода
 int  libuf = 0,                       //количество принятых байт
      lobuf = 0;                       //количество отправленных байь

 if ((libuf = recv(cS,ibuf,sizeof(ibuf),NULL)) == SOCKET_ERROR)
              throw  SetErrorMsgText("recv:",WSAGetLastError());

 _itoa(lobuf, obuf+sizeof("sever: принято ")-1,10);

  if ((lobuf = send(cS,obuf,strlen(obuf)+1,NULL)) ==  SOCKET_ERROR)
               throw  SetErrorMsgText("send:",WSAGetLastError());
 //.....................................................................
}
catch (string errorMsgText)
{cout << endl << errorMsgText;}
//.....................................................................
```

Рисунок 3.11.3. Пример использования функций send и recv

На рисунке 3.11.3 приведен пример использования функций send и recv в программе сервера. После создания канала сервер выдал функцию recv ожидающую данные от подсоединившегося клиента. После получения данных от клиента, сервер формирует выходной буфер и отправляет его содержимое в адрес клиента с помощью функции send.

3.12. Обмен данными без соединения

Если для передачи данных на транспортном уровне используется протокол UDP, то говорят, об обмене данными без соединения или об обмене данными с помощью сообщений. Для отправки и приема сообщений в Winsock2 используются функции sendto и recvfrom (рисунки 3.12.1 и 3.12.2). При этом предполагается, что сообщения будут курсировать между сокетами ориентированными на сообщения.

Особенностью использования этих функций заключается в том, что протоколом UDP не гарантируется доставка и правильная последовательность приема отправленных сообщений. Весь контроль

надежности доставки и правильной последовательностью поступления сообщений возлагается на разработчика приложения. В связи с этим, обмен данными с помощью сообщений используется, в основном, для широковещательных сообщений или для пересылки коротких сообщений, последовательность получения которых не имеет значения.

```
// -- отправить сообщение
// Назначение: функция предназначена для отправки сообщения
//             без установления соединения
//

  int sendto(
       SOCKET s,               // [in] дескриптор сокета
       const char*  buf,       // [in] буфер для пересылаемых данных
       int          len,       // [in] размер буфера buf
       int          flags,     // [in] индикатор режима маршрутизации
       const struct sockaddr* to, // [in] указатель на SOCKADDR_IN
       int          tolen        // [in] длина структуры to
             );

// Код возврата: при успешном завершении функция возвращает
//               количество пересланных  байт данных, иначе
//               возвращается SOCKET_ERROR
// Примечания: - функция может применяться  только для
//               сокетов, ориентированных  на  сообщения
//               (SOCK_DGRAM))
//             - параметр to указывает на  структуру
//               SOCKADDR_IN с параметрами сокета получателя;
//             - для параметра flags рекомендуется установить
//               значение NULL
```

Рисунок 3.12.1. Функция sendto

Также как и функции send и recv, функции sendto и recvfrom работают в синхронном режиме, т.е. вызвав, например, функцию recvfrom вызывающая программа не получит управления до момента завершения приема данных.

Следует также обратить внимание, что обе функции используют в качестве параметров структуру SOCKADDR_IN. В случае выполнения функции send, структура должна содержать параметры сокета получателя. У функция recv структура SOCKADDR_IN, наоборот, предназначена для получения параметров сокета отправителя.

Часто протокол UDP (и соответственно функции sendto и recvfrom) используется для пересылки сообщений предназначенных для рассылки одного сообщения всем компьютерам сети (широковещательные сообщения). Для этого в параметрах сокета отправляющей стороны используются специальные широковещательные и групповые IP-адреса. Использование и групповых адресов функцией sendto по умолчанию запрещено. Разрешение

```
// -- принять сообщение
// Назначение: функция предназначена для получения сообщения
//             без установления соединения

  int recvfrom(
      SOCKET s,               // [in] дескриптор сокета
      char*  buf,             // [out] буфер для получаемых данных
      int          len,       // [in] размер буфера buf
      int          flags,     // [in] индикатор режима маршрутизации
      struct sockaddr*  to,   // [out] указатель на SOCKADDR_IN
      int*         tolen       //[out] указатель на размер to
             );

// Код возврата: при успешном завершении функция возвращает
//                количество принятых  байт данных, иначе
//                возвращается SOCKET_ERROR
// Примечания: - функция может применяться  только для
//                сокетов, ориентированных  на  сообщения
//                (SOCK_DGRAM));
//             - параметр to  указывает на  структуру
//               SOCKADDR_IN с параметрами сокета отправителя;
//             - параметр tolen содержит адрес четырех байт, с
//               размером структуры SOCADDR_IN
//             - для параметра flags рекомендуется установить
//               значение NULL
```

Рисунок 3.12.2 Функция recvfrom

```
//.....................................................................
try
{  //...WSAStartup(...),sS = socket(...,SOCKET_DGRAM,...)
 SOCKADDR_IN serv;               // параметры  сокета sS
 serv.sin_family  = AF_INET;      // используется IP-адресация
 serv.sin_port = htons(2000);     // порт 2000
 serv.sin_addr.s_addr = INADDR_ANY; // адрес сервера
 if(bind(sS,(LPSOCKADDR)&serv, sizeof(serv))== SOCKET_ERROR)
         throw  SetErrorMsgText("bind:", WSAGetLastError());

 SOCKADDR_IN clnt;                // параметры  сокета клиента
 memset(&clnt,0,sizeof(clnt));    // обнулить память
 int lc = sizeof(clnt);
 char ibuf[50];                   //буфер ввода
 int  lb = 0;                     //количество принятых байт
 if (lb = recvfrom(sS, ibuf, sizeof(ibuf), NULL,
                  (sockaddr*)&clnt, &lc))   ==  SOCKET_ERROR)
        throw  SetErrorMsgText("recv:",WSAGetLastError());
  //.............................................................
}
catch (string errorMsgText)
{cout << endl << errorMsgText;}
```

Рисунок 3.12.3 Пример использования функции recvfrom в программе сервера

на использование широковещательных устанавливается функцией setsockopt.

На рисунках 3.12.3 и 3.12.4 приведены примеры применения функций recvfrom и sendto, которые используются в программах сервера и клиента соответственно.

```
//.....................................................................
try
{//...WSAStartup(...),cC = socket(...,SOCKET_DGRAM,...)

SOCKADDR_IN serv;               // параметры  сокета сервера
serv.sin_family  = AF_INET;     // используется ip-адресация
serv.sin_port = htons(2000);    // порт 2000
serv.sin_addr.s_addr = inet_addr("127.0.0.1"); // адрес сервера
char obuf[50]= "client: I here";   //буфер вывода
int  lobuf = 0;                    //количество отправленных

if ((lobuf = sendto(cC,obuf,strlen(obuf)+1,NULL,
                  (sockaddr*)&serv, sizeof(serv))) == SOCKET_ERROR)
        throw  SetErrorMsgText("recv:",WSAGetLastError());
 //.....................................................
}
catch (string errorMsgText)
{cout << endl << errorMsgText;}
//.....................................................................
```

Рисунок 3.12.3 Пример использования функции sendto в программе клиента

3.13. Пересылка файлов и областей памяти

Интерфейс Winsock2 может быть использован для пересылки файлов и непрерывной области памяти компьютера. Пересылка файлов существляется с помощью функции TransmitFile (рисунок 3.13.1), а пересылку области памяти можно осуществить с помощью функции TransmitPackets (описание этой функции здесь не рассматривается). Следует отметить, что эти функции не поддерживаются интерфейсом сокетов BSD, но активно используются операционной системой Windows для кэширования данных.

Использование функции TransmitFile предполагает существование соединения и наличие доступного и открытого файла данных. Пересылка осуществляется блоками, размер которых указывается в параметрах функции. Прием данных на другой стороне осуществляется с помощью функции recv. Прием данных осуществляется до тех пор пока не разорвется соединение или не поступит пустой блок данных (функция recv вернет нулевое значение).

На рисунке 3.13.2 приведен фрагмент программы использующей функцию TransmitFile. Следует обратить внимание, что пересылаемый файл должен быть открытым с помощью стандартной функции _open. Кроме того, в качестве параметра функция TransmitFile использует системный

```
// -- переслать файл
// Назначение: функция предназначена для пересылки файла
//             по установленному соединению
//

BOOL TransmitFile(
   SOCKET         s,      // [in] дескриптор сокета
   HANDLE         hf,     // [in] HANDLE файла
   DWORD          nw,     // [in[ общее количество пересылаемых байтов
   DWORD          ns,     // [in] размер буфера пересылки
   LPOVERLAPPED   po,     // [in] указатель на структуру OVERLAPPED
   LPTRANSMIT_FILE_BUFFERS pb,  //[in] указатель TRANSMIT_FILE_BUFFERS
   DWORD           flag    // [in] индикатор  режим  сокета
                   );

// Код возврата: в случае успешного завершения возвращается
//               TRUE, иначе функция возвращает значение FALSE
// Примечания: - значение параметра nw может иметь значение
//               NULL, в этом случае пересылается весь файл;
//             - значение параметра ns может иметь значение
//               NULL, в этом случае размер буфера пересылки
//               устанавливается по умолчанию;
//             - структура OVERLAPPED используется для
//               управления вводом/выводом; если параметр po
//               имеет значение NULL, то файл пересылается с
//               текущей позиции файла;
//             - структура  LPTRANSMIT_FILE_BUFFERS
//               используется  для  пересылки информации
//               перед и после пересылаемых данных  файла;
//               если значение параметра pb установлено в
//               NULL, то пересылаются только данные файла;
//             - параметр flag может принимать различные:
//               TF_DISCONNECT - указывает на необходимость
//               разрыва соединения после завершения функции;
//               TF_REUSE_SOCKET - предполагает дальнейшее
//               использование сокета.
```

Рисунок 3.13.1. Функция TransmitFile

```
//.............................................................
int  f;
long  fh;
  if ((f =_open("TransmitFile.txt",O_RDONLY))> 0)  // файл открылся ?
    {
     fh = _get_osfhandle(f);                 // получить os-handel файла
     if (TransmitFile(sS,(HANDLE)fh,0,0,NULL,NULL,TF_DISCONNECT)
                                                          == FALSE)
         throw  SetErrorMsgText("transmit:",WSAGetLastError());
    }
//.............................................................
```

Рисунок 3.13.2. Пример использования функции TransmitFile

дескриптор файла, который в примере получается с помощью другой библиотечной функции _get_osfhandle. Описание стандартных функций библиотеки можно найти в справочниках по языку C++, например в [13, 14].

3.14. Применение интерфейса внутренней петли

При отладке распределенных приложений удобно использовать интерфейс внутренней петли. Применение интерфейса внутренней петли позволяет моделировать обмен данными между процессами распределенного приложения на одном компьютере.

Как уже отмечалось выше, для интерфейса внутренней петли зарезервирован IP-адрес 127.0.0.1. В формате TCP/IP этот адрес можно задать с помощью определенной в Winsock2.h константы INADDR_LOOPBACK. Все приведенные выше примеры, демонстрирующие обмен данными в сети TCP/IP, можно выполнить на одном компьютере с использованием этого адреса.

3.15. Использование широковещательных IP-адресов

До сих пор при разработке распределенного приложения предполагались известными сетевой адрес компьютера, на котором находится программа-сервер и номер порта, прослушиваемый этой программой. В реальности распределенное приложение не должно быть привязано к конкретным параметрам сокетов, т.к. это делает ограниченным его применение.

Для обеспечения независимости приложения от параметров сокета сервера (сетевой адрес и номера порта), как правило, номер порта делают одним из параметров инициализации сервера и хранят в специальных конфигурационных файлах, которые считывается сервером при загрузке (реже номер порта передается в виде параметра в командной строке). Так, например, большинство серверов баз данных в качестве одного из параметров инициализации используют номер порта, а при конфигурации (или инсталляции) клиентских приложений указывается сетевой адрес и порт сервера.

В некоторых случаях удобно возложить поиск сетевого адреса сервера на само клиентское приложение (при условии, что номер порта сервера известен). В этих случаях используются широковещательные сетевые адреса, позволяющие адресовать сообщение о поиске сервера всем компьютерам сети. Предполагается, что сервер (или несколько серверов) должен находиться в состоянии ожидания (прослушивания) на доступном в сети компьютере. При получении сообщения от клиента, сервер определяет параметры сокета клиента и передает клиенту необходимые данные для установки канала связи. В общем случае в сети может находиться несколько серверов, которые откликнутся на запрос клиента. В этом случае алгоритм работы клиента должен предполагать процедуру обработки откликов и выбора подходящего сервера. Сразу следует оговориться, что реально

данный метод можно применять только внутри сегмента локальной сети, т.к. широковещательные пакеты, как правило, не пропускаются маршрутизаторами и шлюзами .

Использование широковещательных адресов возможно только в протоколе UDP. Поэтому при создании дескрипторов сокетов (в программах клиентов и серверов) при вызове функции socket значение параметра type должно быть SOCK_DGRAM**,** а для обмена данными этом случае используются функции sendto и recvfrom**.**

```
// -- установить опции сокета
// Назначение: функция предназначена для установки режимов
//              использования сокета

  int setsockopt (
      SOCKET       s,          // [in] дескриптор сокета
      int          level,      // [in] уровень действия режима
      int          optname,    // [in] режим сокета для установки
      const char*  optval,     // [in] значение режима сокета
      int          fromlen     // [in] длина буфера optval
                );
// Код возврата: в случае успешного завершения возвращается
//               нуль, иначе функция возвращает значение
//               SOCKET_ERROR
// Примечания: - поддерживаются два значения параметра level:
//               SOL_SOCKET и IPPROTO_TCP;
//             - для уровня SOL_SOCKET параметр optval может
//               принимать более десяти различных значений;
//               например, SO_BROADCAST - для разрешения
//               использования широковещательного адреса;
//             - для уровня IPPROTO_TCP поддерживается одно
//               значение параметра level:TCP_NODELAY,
//               которое позволяет устанавливать или отменять
//               использование алгоритма Нейгла (см. TCP/IP);
//             - значение fromlen всегда sizeof(int);
//             - если необходимо установить указанный
//               параметр(optname) в состояние Enabled,
//               то в поле optval долнжо быть не нулевое
//               значение (например, 0x00000001), если же
//               параметр  устанавливается в состояние
//               Disabled, то поле optval должно содержать
//               0x00000000
```

Рисунок 3.15.1. Функция setsockopt

Стандартный широковещательный адрес в формате TCP/IP задается с помощью константы INADDR_BROADCAST, которая определена в Winsock2.h. По умолчанию использование стандартного широковещательного адреса не допускается и для его применения необходимо установить специальный режим использования сокета

SO_BROADCAST с помощью функции setsockopt (рисунок 3.15.1). Проверить установленные для сокета режимы можно с помощью функции getsockopt (описание здесь не приводится).

```
//.....................................................
SOCKET cC;

if ((cC = socket(AF_INET, SOCK_DGRAM, NULL))== INVALID_SOCKET)
          throw  SetErrorMsgText("socket:",WSAGetLastError());

int optval = 1;
if (setsockopt(cC,SOL_SOCKET,SO_BROADCAST,
               (char*)&optval,sizeof(int)) == SOCKET_ERROR)
          throw  SetErrorMsgText("opt:",WSAGetLastError());

SOCKADDR_IN all;                          // параметры  сокета sS
all.sin_family = AF_INET;                 // используется IP-адресация
all.sin_port = htons(2000);               // порт 2000
all.sin_addr.s_addr = INADDR_BROADCAST; // всем
char buf[]  = "answer anyone!";

if ((sendlen = sendto(cC, sendbuf, sizeof(buf), NULL,
                      (sockaddr*)&all, sizeof(all)))== SOCKET_ERROR)
          throw  SetErrorMsgText("sendto:",WSAGetLastError());
//.....................................................
```

Рисунок 3.15.1. Пример применения setsockope

На рисунке 3.15.1 приводится фрагмент программы, использующей стандартный широковещательный адрес. Функция setsockopt используется в этом примере для установки опции сокета SO_BROADCAST, позволяющей использовать адрес INADDR_BROADCAST.

3.16. Применение символических имен компьютеров

В предыдущем разделе разбирался механизм поиска серверного компьютера с помощью использования широковещательных адресов. При наличии специальной службы в сети способной разрешить адрес компьютера по его символическому имени (например, DNS или некоторые протоколы, работающие поверх TCP/IP) проблему можно решить с помощью функции gethostbyname (рисунок 3.16.1). При этом предполагается, что известно символическое имя компьютера, на котором находится программа сервера.

Такое решение достаточно часто применяется разработчиками распределенных систем. Связав набор программ-серверов с определенными стандартными именами компьютеров, распределенное приложение становится не зависимым от адресации в сети. Естественно при этом необходимо позаботиться, чтобы существовала служба, разрешающая адреса

компьютеров по имени. Установка таких служб, как правило, возлагается на системного администратора сети.

Помимо функции gethostbyname в составе Winsock2 имеется функция gethostbyaddr (рисунок 3.16.2), назначение которой противоположно: получение символического имени компьютера по сетевому адресу. Обе функции используют структуру hosten (рисунок 3.16.3), содержащуюся в Winsock2.h.

```
// -- получить адрес хоста по его имени
// Назначение: функция для получения информации о хосте по
//             его символическому  имени

 hostent*  gethostbyname
                (
            const char*   name,   // [in] символическое имя хоста
                );

// Код возврата: в случае успешного завершения функция
//               возвращает указатель на структуру hosten,
//               иначе значение NULL
// Примечание:   допускается в качестве символического имени,
//               указать символическое обозначение адреса
//               хоста  в виде n.n.n.n
```

Рисунок 3.16.1. Функция gethostbyname

```
// -- получить имя  хоста по его адресу
// Назначение: функция для получения информации о хосте по
//             его символическому  имени

 hostent*  gethostbyaddr
                (
        const char*   addr, // [in] адрес в сетевом формате
              int     la,   // [in] длина адреса в байтах
              int     ta    // [in] тип адреса: для TCP/IP AF_INET
                );

// Код возврата: в случае успешного завершения функция
//               возвращает указатель на структуру hosten,
//               иначе возвращается значение NULL
```

Рисунок 3.16.2 Функция gethostbyaddr

```
typedef struct hostent {              // структура hostent
        char FAR* h_name;             // имя хоста
        char FAR  FAR** h_aliases;    // список алиасов
         short h_addrtype;             // тип адресации
        short h_length;               // длина адреса
        char FAR  FAR** h_addr_list; // список адресов
                        } hostent;
```

Рисунок 3.16.3 Структура hostent

Следует отметить, что символическое имя ***localhost*** является зарезервированным именем и предназначено для обозначения собственного имени компьютера. Если с помощью функции gethostbyname получить адрес компьютера с именем localhost, то в будет собственный получен IP-адрес компьютера или адрес INADDR_LOOPBACK.

Кроме того, для получения действительного собственного имени компьютера (NetBIOS-имени или DNS-имени) можно использовать функцию gethostname (рисунок 3.16.4).

```
// -- получить имя хоста
// Назначение: функция для получения собственного имени хоста

 int  gethostname
                 (
            char*  name , // [out] имя хоста
            int    ln     // [in] длина буфера name
                 );

// Код возврата: в случае успешного завершения функция
//                возвращает нуль, иначе возвращается значение
//                SOCKET_ERROR
```

Рисунок 3.16.4 Функция gethostname

3.17. Итоги главы

1. Интерфейс сокетов – это набор специальных функций, входящий в состав операционной системы и предназначенный для доступа прикладных процессов к сетевым ресурсам. Исторически первым интерфейс сокетов был разработан для операционной системы BSD Unix. В настоящее время этот интерфейс поддерживается большинством операционных систем и регулируется стандартом POSIX. Сокетами называют объекты операционной системы, представляющие точки приема или отправления данных в сети.
2. Интерфейс Windows Socket 2 (Winsock2) является реализацией интерфейса сокетов для семейства 32-битовых операционных систем. Winsock2 акцентирован на работу в сети TCP/IP. В состав Winsock2

входит динамическая библиотека WS2_32.DLL, библиотека экспорта WS2_32.LIB и заголовочный файл Winsock2.h.

3. Набор функций Winsock2 включает в себя функции, позволяющие: создавать сокеты, устанавливать параметры и режимы работы сокетов, осуществлять пересылку данных между сокетами, преобразовать форматы данных, обрабатывать возникающие ошибки и другие функции.
4. Winsock2 обеспечивает две схемы взаимодействия прикладных процессов: без установки соединения и с установкой соединения. Схема взаимодействия без установки соединений предполагает использование протокола UDP, а схема с установкой соединения – протокола TCP. Кроме того, обе схемы ориентированы на создание распределенного приложения архитектуры клиент-сервер, т.к. предполагают наличие двух функционально несимметричных сторон: клиента и сервера. Основным отличие между клиентом и сервером заключается в том, что инициирует связь всегда клиент, а сервер ожидает обращение клиента за сервисом.
5. Схема без установки соединения предполагает создание и использование сокетов, ориентированных на сообщения. Обмен данными между сокетами происходит пакетами протокола UDP. При обмене данными интерфейс (на самом деле протокол UDP) не гарантирует доставки сообщений получателю и правильный порядок их получения (порядок получения сообщений правильный, если он совпадает с порядком их отправления). Контроль за доставкой и порядком сообщений возлагается на само приложение. Схема без установки соединения применяется , как правило, для пересылки коротких и (или) широковещательных сообщений.
6. Схема с установкой соединения предполагает использование сокетов, ориентированных на поток. В этом случае интерфейс гарантирует доставку и правильный порядок данных.
7. Для моделирования на одном компьютере работы распределенного приложения в сети, часто применяют интерфейс внутренней петли.
8. Помимо стандартных функций (описанных в стандарте POSIX), интерфейс Winsock2 содержит ряд функции характерных только для Winsock2. Например, функции пересылки файлов и областей памяти.

Глава 4. Интерфейс Named Pipe

4.1. Предисловие к главе

Современные операционные системы имеют встроенные механизмы межпроцессного взаимодействия - ***IPC (InterProcess Communication)*** [15], предназначенные для обмена данными между процессами и для синхронизации процессов.

Разработчики программного обеспечения могут использовать IPC с помощью предоставляемых операционными системами программных интерфейсов (API). С программными интерфейсами IPC операционной системы Windows можно ознакомиться, например, в [4].

В этой главе рассматривается программный интерфейс одного из IPC-механизмов операционной системы Windows, который может быть использован для обмена данными между распределенными в локальной сети процессами, и имеет название ***Named Pipe (именованный канал)***.

4.2. Назначение и состав интерфейса Named Pipe

Именованным каналом называется объект ядра операционной системы, который обеспечивает обмен данными между процессами, выполняющимися на компьютерах в одной локальной сети. Процесс, создающий именованный канал, называется ***сервером именованного канала***. Процессы, которые связываются с именованным каналом, называются ***клиентами именованного канала***. Любой именованный канал идентифицируется своим именем, которое задается при создании канала.

Именованные каналы бывают: ***дуплексные*** (позволяющие передавать данные в обе стороны) и ***полудуплексны***е (позволяющие передавать данные только в одну сторону). Передача данных в именованном канале может осуществляться как потоком, так и сообщениями. Обмен данными в канале может быть ***синхронным*** и ***асинхронным***.

Для использования функций интерфейса Named Pipe в программе на языке C++ необходимо включить в ее текст заголовочный файл Windows.h. Сами функции интерфейса располагаются в библиотеке KERNEL32.DLL ядра операционной системы.

В таблице 4.2.1 перечислены основные функции интерфейса Named Pipe. Следует отметить, что функции CreateFile, ReadFile, WriteFile, которые тоже перечислены в таблице 4.2.1, применяются не только для работы с именованными каналами, но и для работы с файловой системой, с сокетами и т.д. Поэтому эти универсальные функции часто не указывают в составе Named Pipe API.

Все функции Named Pipe API можно разбить на три группы: функции управления каналом (создать канал, соединить сервер с каналом, открыть канал, получить информацию об именованном канале, получить состояние канала, изменить характеристики канала); функции обмена данными (писать в канал, читать из канала, копировать данные канала) и функции для работы с транзакциями.

Таблица 4.2.1

Наименование функции	Назначение
`CallNamedPipe`	Выполнить одну транзакцию
`ConnectNamedPipe`	Соединить сервер с каналом
`CreateFile`	Открыть канал
`CreateNamedPipe`	Создать именованный канал
`DisconnectNamedPipe`	Закончить обмен данными
`GetNamedPipeHandleState`	Получить состояние канала
`GetNamedPipeInfo`	Получить информацию об именованном канале
`PeekNamedPipe`	Копировать данные канала
`ReadFile`	Читать данные из канала
`SetNamedPipeHandleState`	Изменить характеристики канала
`TrasactNamedPipe`	Писать и читать данные канала
`WaitNamedPipe`	Определить доступность канала
`WriteFile`	Писать данные в канал

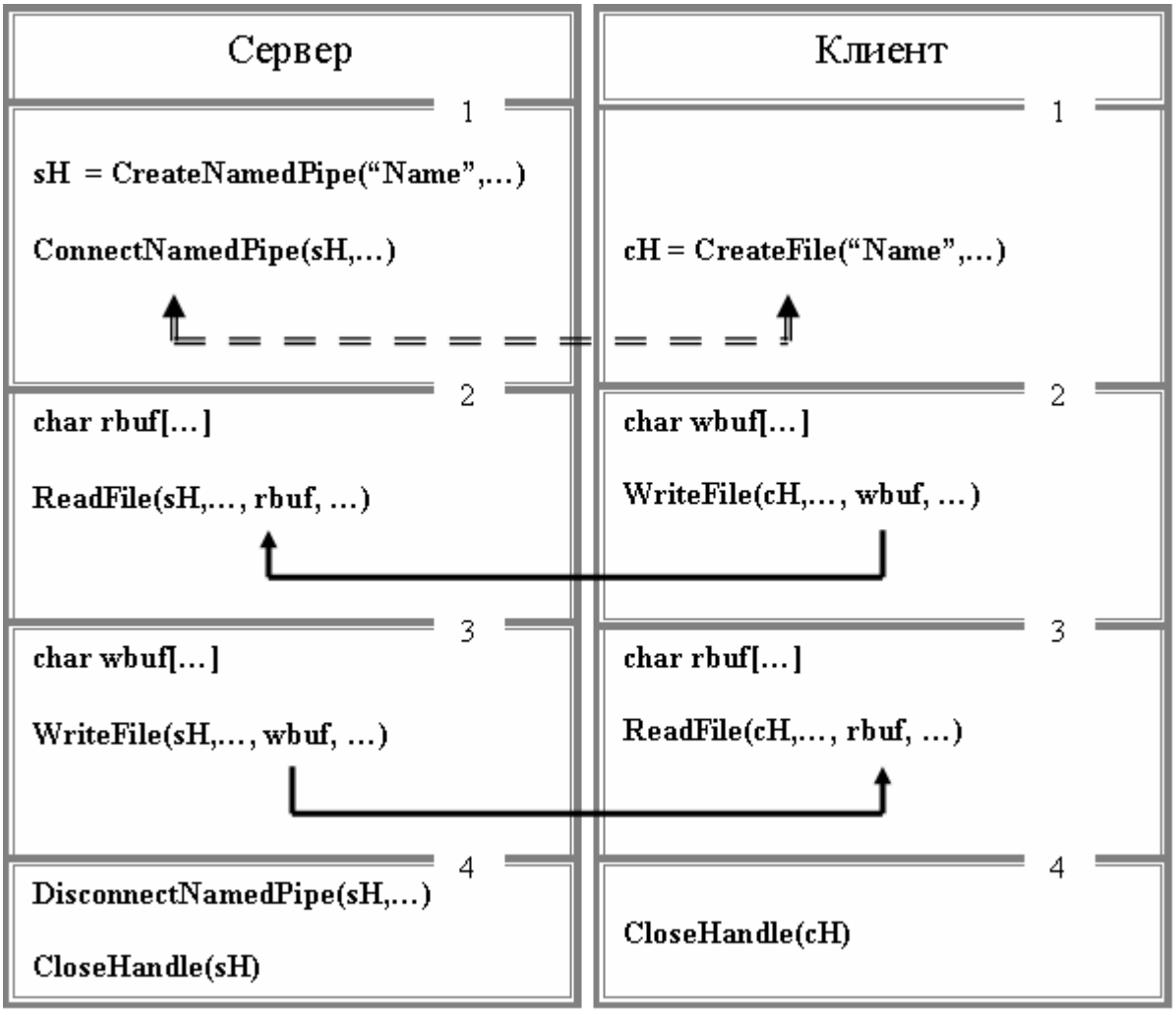

Рисунок 4.2.1. Схема взаимодействия процессов, использующих Named Pipe API

Следует сразу отметить, что при создании именованного канала в программе на языке C++, одновременно создается дескриптор (HANDEL), который потом используется другими функциями Named Pipe API для работы с данным экземпляром именованного канала. По окончании работы с каналом необходимо эти дескрипторы закрыть с помощью функции CloseHandle Win32 API.

На рисунке 4.2.1 изображены две программы, реализующие два процесса распределенного приложения. Каждая из программ разбита на четыре блока. Сплошными направленными линиями обозначается движение данных от одного процесса к другому. Прерывистой линией обозначается синхронизация процессов.

В первом блоке программы сервера выполняются две функции: CreateNamedPipe (создать именованный канал) и ConnectNamedPipe (подсоединить сервер к каналу). Одним из параметров функции CreateNamedPipe является имя канала (строка), а результатом ее работы (возвращаемым значением) является дескриптор (HANDEL) канала. Функция ConnectNamedPipe приостанавливает выполнение программы клиента до момента, пока программа клиента не выполнит функцию CreateFile.

Во втором и третьем блоках программы сервера осуществляется ввод и вывод данных (функции ReadFile и WriteFile) в именованном канале. Следует обратить внимание, что функции осуществляющие ввод и вывод используют в качестве одного из своих параметров дескриптор именованного канала.

В последнем четвертом блоке программы сервер разрывает соединение с помощью функции DisconnectNamedPipe и закрывает дескриптор именованного канала.

Для программы клиента остается пояснить только первый блок, т.к. всем остальным блокам, есть аналогичные в программе сервера. В первом блоке программы клиента выполняется функция CreateFile, одним из параметров которой является строка с именем канала. Если к моменту выполнения функции канал уже создан и сервер подсоединился к каналу, то функция CreateFile возвращает дескриптор именованного канала, который потом используется в других функциях программы клиента. Иногда перед выполнением функции CreateFile, выполняют функцию WaitNamedPipe, позволяющую определить доступность экземпляра канала. Назначение всех функций будет пояснено ниже.

Как уже отмечалось, передача данных может осуществляться как потоком, так и сообщениями. Передача данных потоком возможна в том случае, если сервер и клиент работают на одном компьютере и использует локальные имена канала, в других случаях передача данных осуществляется сообщениями. Схема изображенная на рисунке 4.2.1 является общей для этих двух случаев.

В случае функции Named Pipe API завершения с ошибкой, все функции формируют код системной ошибки Windows, который может быть получен с помощью функции GetLastError.

4.3. Создание именованного канала

Сервером именного канала является процесс, создающий именованный канал. Именованный канал создается с помощью функции CreateNamedPipe (рисунок 4.3.1).

```
// -- создать именованный канал
// Назначение: функция  предназначена для создания
//             именованного канала

 HANDLE CreateNamedPipe
                 (
     LPCTSTR    pname, // [in] символическое имя канала
     DWORD      omode, // [in] атрибуты канала
     DWORD      pmode, // [in] режимы передачи данных
     DWORD      pimax, // [in] макс. к-во экземпляров канала
     DWORD      osize, // [in] размер выходного буфера
     DWORD      isize, // [in] размер входного буфера
     DWORD      timeo, // [in] время ожидания связи с клиентом
     LPSECURITY_ATTRIBUTES sattr // [in] атрибуты безопасности
                 );
// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор именованного канала, иначе
//       возможны следующие значения:
//       INVALID_HANDLE_VALUE -  неудачное завершение;
//       ERROR_INVALID_PARAMETER  - значение параметра pimax
//       превосходит величину  PIPE_UNLMITED_INSTANCES
// Примечание: pname - указывает на строку именем канала в
//       локальном формате;
//       omode - задает флаги направления передачи, например
//       флаг FILE_ACCESS_DUPLEX разрешает чтение и запись в
//       канал; помимо направления здесь могут быть заданы
//       флаги асинхронной передачи, режимы буферизации и
//       безопасности;
//       pmode – задает флаги способов передачи данных,
//       например, флаг PIPE_TYPE_MESSAGE|PIPE_WAIT разрешает
//       запись данных сообщениями в синхронном режиме, а флаг
//       PIPE_READTYPE_MESSAGE|PIPE_WAIT разрешает чтение
//       сообщений в синхронном режиме;
//       pimax – максимальное количество экземпляров канала,
//       значение должно находиться в пределах от 1 до
//       PIPE_UNLMITED_INSTANCES;
//       osize, isize – значения рассматриваются Windows
//       только как пожелания пользователя (рекомендуется 0);
//       timeo – параметр устанавливает время ожидания связи с
//       с  сервером в миллисекундах для функции WaitNamedPipe
//       с параметром  NMWAIT_USE_DEFAULT_WAIT; может быть
//       установлено значение INFINITE (ждать бесконечно);
//       sattr – для установки атрибутов безопасности
//       по умолчанию, следует установить значение NULL
```

Рисунок 4.3.1. Функция CreateNamedPipe

Первый параметр функции CreateNamedPipe – указатель на строку имени канала. В зависимости от контекста в функциях используется два формата имени канала: локальный формат (рисунок 4.3.2) и сетевой формат (рисунок 4.3.3).

```
\\.\pipe\xxxxx
где: точка (.) - обозначает  локальный компьютер;
     pipe      - фиксированное слово;
     xxxxx     - имя канала
```

Рисунок 4.3.2. Локальный формат имени канала

```
\\servname\pipe\xxxxx
где:  servname - имя компьютера – сервера именованного канала;
      pipe     - фиксированное слово;
      xxxxx    - имя канала
```

Рисунок 4.3.3. Сетевой формат имени канала

Для связи сервера по одному именованному каналу с несколькими клиентами, сервер должен создать несколько экземпляров этого канала. Каждый экземпляр создается функцией CreateNamedPipe, которая возвращает дескриптор экземпляра именованного канала. Отметим, что поток создающий экземпляр именованного канала должен иметь право доступа FILE_CREATE_PIPE_INSTANCE. Этим правом по умолчанию обладает поток создавший канал. Подробнее о применении нескольких экземпляров одного канала можно ознакомиться в [4] .

```
// -- соединить сервер с  именованным каналом
// Назначение: функция  предназначена для ожидания сервером
//             подсоединения к  экземпляру именованного канала
//             клиента

 BOOL ConnectNamedPipe
                  (
     HANDLE        hP,  // [in] дескриптор именованного канала
     LPOVERLAPPED ol    // [in,out] используется для асинхр. связи
                  );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание: параметр ol используется только в том случае,
//       если используется асинхронная связь, в случае
//       синхронной  связи можно установить значение NULL
```

Рисунок 4.3.4. Функция ConnectNamedPipe

После того, как сервер создал именованный канал, он должен дождаться соединения клиента с этим сервером. Для этого сервер должен вызвать функцию ConnectNamedPipe (рисунок 4.3.4).

На рисунке 4.3.5 изображен фрагмент программы сервера. С помощью функции CreateNamedPipe создается дуплексный, синхронный канал с именем ConsolePipe, предназначенный для передачи сообщений. Созданный канал может иметь только один экземпляр и, если программа клиента будет использовать для проверки доступности канала ConsolePipe функцию WaitNamedPipe с параметром NMPWAIT_USE_DEFAULT_WAIT, то будет установлен бесконечный интервал ожидания.

```
//.............................................................
HANDLE hPipe; // дескриптор канала
try
{
 if ((hPipe = CreateNamedPipe("\\\\.\\pipe\\ConsolePipe",
                PIPE_ACCESS_DUPLEX,            //дуплексный канал
                PIPE_TYPE_MESSAGE|PIPE_WAIT,   // сообщения|синхронный
                1, NULL, NULL                  // максимум 1 экземпляр
                INFINITE,NULL))  == INVALID_HANDLE_VALUE)
                     throw SetPipeError("create:",GetLastError());
 if (!ConnectNamedPipe(hPipe,NULL))             // ожидать клиента
                      throw SetPipeError("connect:",GetLastError());
//.............................................................
}
catch (string ErrorPipeText)
{cout << endl << ErrorPipeText;}
//.............................................................
```

Рисунок 4.3.5. Фрагмент программы сервера

В программе, приведенной на рисунке 4.3.5, для обработки ошибок используется функция SetPipeError. Эта функция аналогична функции SetErrorMsgText, которая использовалась в примерах третьей главы для обработки ошибок Winsock2 API.

4.4. Соединение клиентов с именованным каналом

Прежде чем соединиться с именованным каналом, клиент может определить: доступен ли какой либо экземпляр этого канала. С этой целью клиент может вызывать функцию WaitNamedPipe (рисунок 4.3.1).

После обнаружения свободного канала, клиент может установить связь с каналом помощью функции CreateFile (рисунок 4.3.2). После успешного выполнения функции клиент и сервер могут обмениваться данными.

Здесь не рассматриваются атрибуты безопасности, которые могут быть определены при вызовах функций CreateNamedPipe и CreateFile. Однако, поясним, что для организации обмена, необходимо, чтобы атрибуты безопасности в этих функция были согласованными.

```
// -- определить доступность канала
// Назначение: функция  предназначена для ожидания клиентом
//             доступного именованного канала

 BOOL WaitNamedPipe
                  (
     LPCTSTR    pn,  // [in] символическое имя канала
     DWORD      to   // [in] интервал ожидания (мс)
                  );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание:- если используется локальный канал, то имя
//       канала задается в локальном формате, если же канал
//       создан на другом компьютере, то имя канала следует
//       задавать в сетевом формате;
//     - параметр to определяет интервал времени
//       ожидания (в миллисекундах) освобождения экземпляра
//       канала; если  для параметра to  установлено значение
//       NMWAIT_USE_DEFAULT_WAIT, то интервал определяется
//       параметром timeo  функции CreateNamedPipe; если
//       установлено значение NMWAIT_WAIT_FOREVER, то время
//       ожидания бесконечно
```

Рисунок 4.4.1. Функция WaitNamedPipe

При установке в функциях Named Pipe API атрибутов безопасности по умолчанию, как это сделано во всех приведенных здесь примерах, подсоединиться каналу удаленный клиент сможет только в том случае, если он запущен от того же имени пользователя и с тем же паролем, что и сервер. С применением атрибутов безопасности в Named Pipe API можно ознакомиться в [4].

Кроме того, следует обратить внимание на правильное использование имени канала. На рисунке 4.3.5 при создании канала с помощью функции CreateNamedPipe использовалось имя канала \\.\pipe\ConsolePipe. При записи строки с именем канала в программе на языке C++, символ обратного слеша, в соответствии с правилами языка, удваиваются.

При использовании форматов имени канала, необходимо помнить, что:

1) при создании канала всегда используется локальный формат имени;
2) если клиент удаленный (на другом компьютере), то он всегда должен использовать сетевой формат имени; при этом обмен данными между клиентом и сервером осуществляется сообщениями;
3) если клиент локальный и использует сетевой формат имени при подсоединении к каналу (функция CreateFile), то обмен данными осуществляется сообщениями;
4) если клиент локальный и использует локальный формат имени канала, то обмен данными осуществляется потоком.

```
// -- открыть канал
// Назначение: функция  предназначена для подключения клиента
//             к  именованному каналу

 HANDLE CreateFile
                  (
     LPCTSTR    pname, // [in] символическое имя канала
     DWORD      accss, // [in] чтение или запись в канал
     DWORD      share, // [in] режим совместного использования
     LPSECURITY_ATTRIBUTES sattr // [in] атрибуты безопасности
     DWORD      oflag, // [in] флаг открытия канала
     DWORD      aflag, // [in] флаги и атрибуты
     HANDLE     exten, // [in] дополнительные атрибуты
                  );

// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор именованного канала, иначе
//       INVALID_HANDLE_VALUE -  неудачное завершение
// Примечание:- параметр pname указывается в локальном или
//       сетевом формате:  в зависимости от способа применения
//     - параметр accss может принимать значения GENERIC_READ
//       (чтение), GENERIC_WRITE (запись) или
//       GENERIC_READ| GENERIC_WRITE (запись,чтение)
//     - параметр share может принимать значения
//        FILE_SHARE_READ (совместное чтение),
//        FILE_SHARE_WRITE (совместная запись),
//       FILE_SHARE_READ| FILE_SHARE_WRITE (чтение и запись);
//     - параметр sattr для установки атрибутов безопасности
//       по умолчанию, следует установить значение NULL;
//     - значение параметра oflag всегда устанавливается  в
//       OPEN_EXISTING(открытие существующего канала);
//     - значение параметров aflag и exten можно установить в
//       NULL, что соответствует значениям по умолчанию
```

Рисунок 4.4.2. Функция CreateFile

Если не существует экземпляров именованного канала с тем именем, которое указано в параметре функции WaitNamedPipe, то эта функция немедленно заканчивается неудачей (FALSE), независимо от установленного в параметре функции значения интервала ожидания. Если же канал создан, но сервер не выполнил функцию ConnectNamedPipe, то функция WaitNamedPipe на стороне клиента все равно вернет FALSE и сформирует диагностический код (функции GetLastError) ERROR_PIPE_CONNECTED. Даже в том случае, если функция WaitNamedPipe обнаружит свободный экземпляр канала (и вернет TRUE), то все равно нет гарантии, что до выполнения функции CreateFile этот канал не будет занят другим клиентом. Все эти замечания, делают применение функции WaitNamedPipe в большинстве случаев нецелесообразным. Фрагмент программы клиента,

демонстрирующий подключение к именованному каналу изображен на рисунке 4.4.3.

```
//.............................................................
HANDLE hPipe; // дескриптор канала
try
{
if ((hPipe = CreateFle(
              "\\\\.\\pipe\\ConsolePipe",
               GENERIC_READ|GENERIC_WRITE,
               FILE_SHARE_READ|FILE_SHARE_WRITE,
               NULL, OPEN_EXISTING, NULL,
               NULL)) == INVALID_HANDLE_VALUE)
                throw  SetPipeError("createfile:",GetLastError());
//.............................................................
}
catch (string ErrorPipeText)
{cout << endl << ErrorPipeText;}
//.............................................................
```

Рисунок 4.4.3. Фрагмент программы клиента

4.5. Обмен данными по именованному каналу

Для обмена данными по именованному каналу используются три функции: ReadFile, WriteFile и PeekNamedPipe (рисунки 4.5.1, 4.5.2, 4.5.3).

```
// -- читать данные из канала
// Назначение: функция  предназначена чтения данных из
//             именованного канала

 BOOL ReadFile
            (
     HANDLE         hP,    // [in] дескриптор  канала
     LPVOID         pb,    // [out] указатель на буфер  ввода
     DWORD          sb,    // [in] количество читаемых байт
     LPDWORD        ps,    // [out] количество прочитанных байт
     LPOVERLAPPED   ol     // [in,out] для асинхронной обработки
             );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание если не используется асинхронная обработка
//       параметр ol рекомендуется установить в NULL
```

Рисунок 4.5.1. Функция ReadFile

```
// -- писать данные в канал
// Назначение: функция  предназначена записи данных в
//             именованный канал

 BOOL WriteFile
           (
     HANDLE         hP,   // [in] дескриптор  канала
     LPVOID         pb,   // [in] указатель на буфер  вывода
     DWORD          sb,   // [in] количество записываемых байт
     LPDWORD        ps,   // [out] количество записанных байт
     LPOVERLAPPED   ol    // [in,out] для асинхронной обработки
            );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание если не используется асинхронная обработка
//       параметр ol рекомендуется установить в NULL
```

Рисунок 4.5.2. Функция WriteFile

Параметры функций ReadFile и WriteFile достаточно просты и не требуют дополнительного пояснения. Функция PeekNamedPipe копирует данные из канала в буфер. При этом данные не извлекаются и их еще можно считать (извлечь) с помощью функции ReadFile.

```
// -- копировать данные канала
// Назначение: функция  предназначена для получения  данных
//             из канала без извлечения

 BOOL PeekNamedPipe
           (
     HANDLE         hP,   // [in] дескриптор  канала
     LPVOID         pb,   // [out] указатель на буфер
     DWORD          sb,   // [in] размер буфера
     LPDWORD        pi,   // [out] количество прочитанных байт
     LPDWORD        pa,   // [out] количество доступных байт
     LPDWORD        pr,   // [out] количество непрочитанных байт
           );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
```

Рисунок 4.5.3. Функция PeekNamedPipe

4.6. Передача транзакций по именованному каналу

Для обмена сообщениями по сети может использоваться функция TansactNamedPipe (рисунок 4.6.1), которая объединяет операции чтения и записи в одну операцию. Такую объединенную операцию называют ***транзакцией*** именованного канала. Функция TansactNamedPipe может быть использована только в том случае, если сервер именованного канала установил флаг PIPE_TYPE_MESSAGE.

Применение TransactNamedPipe целесообразно, если другая сторона канала может обеспечить достаточно быструю реакцию и оправить ответ на пришедшее сообщение.

```
// -- писать и читать  данные канала
// Назначение: функция  предназначена для выполнения записи в
//             канал и чтения из канала  за одну операцию

 BOOL TransactNamedPipe
            (
     HANDLE         hP,    // [in] дескриптор  канала
     LPVOID         pw,    // [in] указатель на буфер для записи
     DWORD          sw,    // [in] размер буфера для записи
     LPVOID         pr,    // [out] указатель на буфер для чтения
     DWORD          sr,    // [in] размер буфера для чтения
     LPDWORD        pr,    // [out] количество прочитанных байт
     LPOVERLAPPED   ol     // [in,out] для асинхронного доступа
            );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание: параметр ol используется для асинхронного
//       доступа к каналу, если асинхронный доступ не
//       не предполагается, то следует указать NULL
```

Рисунок 4.6.1. Функция TransactNamedPipe

Часто взаимодействие сервера и клиента сводится к простому запросу клиента к серверу для получения некоторого сервиса. После выполнения запрошенной клиентом сервисной услуги, сервер информирует клиента о результате своей работы. Т.е. речь идет об одиночных эпизодических транзакциях.

Если требуется передать только одну транзакцию, то используют функцию CallNamedPipe (рисунок 4.6.2), которая работает следующим образом.

Сначала осуществляется установка связи с именованным каналом, имя которого указывается в параметрах функции. При этом именованный канал должен быть открыт в режиме данных сообщениями. После установки связи функция пересылает в канал единственное сообщение и получает одно

сообщение в ответ. После обмена данными осуществляется разрыв связи с именованным каналом.

```
// -- выполнить одну транзакцию
// Назначение: функция  предназначена для  установки связи с
//             именованным каналом, выполнения одной транзакции
//             и разрыва связи

 BOOL CallNamedPipe
            (
     LPCTSTR        nP,    // [in] указатель на имя канала
     LPVOID         pw,    // [in] указатель на буфер для записи
     DWORD          sw,    // [in] размер буфера для записи
     LPVOID         pr,    // [out] указатель на буфер для чтения
     DWORD          sr,    // [in] размер буфера для чтения
     LPDWORD        pr,    // [out] количество прочитанных байт
     DWORD          to     //  [in] интервал ожидания
            );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание: параметр to устанавливает  интервал времени в
//       в миллисекундах; кроме того, здесь  могут быть
//       установлены те же значения, что и в функции
//       WaitNamedPipe
```

Рисунок 4.6.2. Функция CallNamedPipe

4.7. Определение состояния и изменение характеристик именованного канала

Для получения информации о созданном именованном канале можно использовать две функции GetNamedPipeInfo и GetNamedPipeHandleState (рисунки 4.7.1 и 4.7.2).

Функция GetNamedPipeInfo используется для получения информации об атрибутах именованного канала, который являются статическими и не могу быть изменены. Входными параметрами служат дескриптор и тип канала, о котором предполагается получить информацию. При завершении, функция возвращает значения размеров буферов ввода и вывода, а также максимальное количество экземпляров данного именованного канала.

Т.к. для получения информации используется дескриптор, то предполагается, что перед выполнением функции GetNamedPipeInfo, канал уже создан (функция CreateNamedPipe на стороне сервера) или открыт (функция CreateFile на стороне клиента). При этом для ее выполнения необходимо, чтобы было разрешено чтение канала.

```
// -- получить информацию об именованном канале
// Назначение: функция  предназначена для  получения
//             статических  характеристик именованного канала
  BOOL GetNamedPipeInfo
            (
      HANDLE  hP,     // [in] дескриптор именованного канала
      LPDWORD pfg,    // [in] указатель на флаг-тип канала
      LPDWORD psw,    // [out] указатель на размер выходного буфера
      LPDWORD psr,    // [out] указатель на размер входного буфера
      LPDWORD pmi,    // [out] указатель на макс. к-во экземпляров канала
            );
// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание: параметр pfg указывает на переменную типа
//       DWORD, в которой установлен тип именованного канала,
//       атрибуты которого запрашиваются; для установки
//       этой переменной должны использоваться константы:
//       PIPE_CLIENT_END, PIPE_SERVER_END - для обозначения
//       типа используемого в функции дескриптора;
//       PIPE_TYPE_BYTE, PIPE_TYPE_MESSAGE – для установки
//       типа передачи (поток и сообщения)
```

Рисунок 4.7.1. Функция GetNamedPipeInfo

```
// -- получить состояния  именованного канала
// Назначение: функция  предназначена для  получения
//              динамических  характеристик именованного канала
  BOOL GetNamedPipeHandleState
            (
      HANDLE  hP,     // [in] дескриптор именованного канала
      LPDWORD pst,    // [out] указатель на состояние канала
      LPDWORD pci,    // [out] указатель на  к-во экземпляров каналов
      LPDWORD pcc,    // [out] указатель на макс. к-во байт
      LPDWORD pto,    // [out] указатель на  интервал задержки
      LPTSTR  pun,    // [out] указатель на имя владельца канала
      DWORD   lun     // [in]  длина буфера для имени владельца канала
             );
// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
// Примечание: - параметр pst указывает на переменную типа
//       DWORD, в которой установлена комбинация значений:
//       PIPE_NOWAIT -  канал не блокирован;
//       PIPE_READMODE_MESSAGE – канал открыт в режиме
//       передачи сообщениями;
//       - параметр pcc  - указывает на максимальное
//       количество байтов, которые  клиент именного канала
//       должен записать в канал перед передачей серверу;
//       - параметр pto  - указывает на количество миллисекунд
//       которые должно пройти прежде, чем данные будут
//       переданы
```

Рисунок 4.7.2. Функция GetNamedPipeHandleState

Чаще всего функция GetNamedPipeInfo используется на стороне клиента после открытия канала для выяснения размеров буферов, установленных операционной системой при создании канала.

Функция GetNametPipeHandleState используется для получения динамических параметров (которые могут быть изменены) именованного канала. Для изменения некоторых параметров может быть использована функция SetNamedHandleState (рисунок 4.7.3).

```
// -- изменить характеристики канала
// Назначение: функция  предназначена для  изменения
//             динамических характеристик именованного канала

 BOOL SetNamedPipeHandleState
           (
     HANDLE  hP,     // [in] дескриптор именованного канала
     LPDWORD pst,    // [in]  указатель на новое состояние канала
     LPDWORD pcc,    // [in]  указатель на макс. к-во байтов
     LPDWORD pto     // [in]  указатель на  интервал задержки
             );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
```

Рисунок 4.7.2. Функция SetNamedPipeHandleState

Более подробно о применении функций определения и изменения состояния именованного канала описано в [4].

4.8. Итоги главы

1. Современные операционные системы имеют встроенные механизмы межпроцессорного взаимодействия (IPC), позволяющие создавать распределенные в локальной сети приложения. Для работы использования IPC-механизмов операционные системы предоставляют специальные программные интерфейсы.
2. Интерфейс Named Pipe (именованный канал) реализует один из IPC-механизмов операционной системы Windows и позволяет создавать распределенные приложения архитектуры клиент-сервер.
3. Именованный канал представляет собой объект операционной системы Windows, позволяющий создавать между распределенными в локальной TCP/IP-сети процессами дуплексные и полудуплексные каналы, по которым может осуществляться передача данных в синхронном или асинхронном режимах.
4. В состав интерфейса Named Pipe входят функции для управления каналом, функции для обмена данными по каналу и функции работы с транзакциями.

Глава 5. Интерфейс Named Pipe

5.1. Предисловие к главе

В этой главе рассматривается еще один IPC – механизм, поддерживаемый операционной системой Windows и имеющий название ***Mailslots (почтовый ящик)***. Также как и Named Pipe механизм Mailslots может быть использован для обмена данными между распределенными в локальной сети процессами.

5.2. Назначение и состав интерфейса Mailslot

Почтовым ящиком (Mailslot) называется объект ядра операционной системы, который обеспечивает передачу данных от процессов-клиентов к процессам-серверам, выполняющимся на компьютерах в одной локальной сети. Процесс, создающий почтовый ящик называется ***сервером почтового ящика***. Процессы, которые связываются с почтовым ящиком, называются ***клиентами почтового*** ящика.

Каждый почтовый ящик имеет имя, которое определяется сервером при создании и используется клиентами для доступа. Передача может осуществляться только сообщениями и в одном направлении – от клиента к серверу. Обмен данными может происходить в синхронном и асинхронном режимах. Допускается создание нескольких серверов с одинаковым именем почтового ящика – в этом случае все отправляемые клиентом сообщения будут поступать во все почтовые ящики, имеющие имя, указанное клиентом. Однако, следует сказать, что такая рассылка сообщений возможна только в том случае, когда длина оправляемых сообщений не превышает 425 байт.

В том случае, если клиент отправляет сообщение размером меньше, чем 425 байт, то пересылка осуществляется без гарантии доставки. Пересылка сообщения размером более 425 байт возможна только от одного клиента к одному серверу.

Перечень функций интерфейса Mailslot API приводится в таблице 5.2.1. Функции CreateFile, ReadFile, WriteFile являются универсальными и используются также для работы с именованными каналами, файловой системой, сокетами и т.д.

Таблица 5.2.1

Наименование функции	Назначение
`CreateFile`	Открыть почтовый ящик
`CreateMailslot`	Создать почтовый ящик
`GetMailslotInfo`	Получить информацию о почтовом ящике
`ReadFile`	Читать данные из почтового ящика
`SetMailslotInfo`	Изменить время ожидания сообщения
`WriteFile`	Писать данные в почтовый ящик

Как и в случае с именованными каналами, для использования функций Mailslot API в программе на языке C++ достаточно включить в ее текст заголовочный файл Widows.h.

На рисунке 5.2.1 изображена схема взаимодействия процесса-сервера и процесса-клиента в простейшем случае. Каждая программа разбита на три блока. Сплошной направленной линий обозначается движение данных от одного процесса к другому.

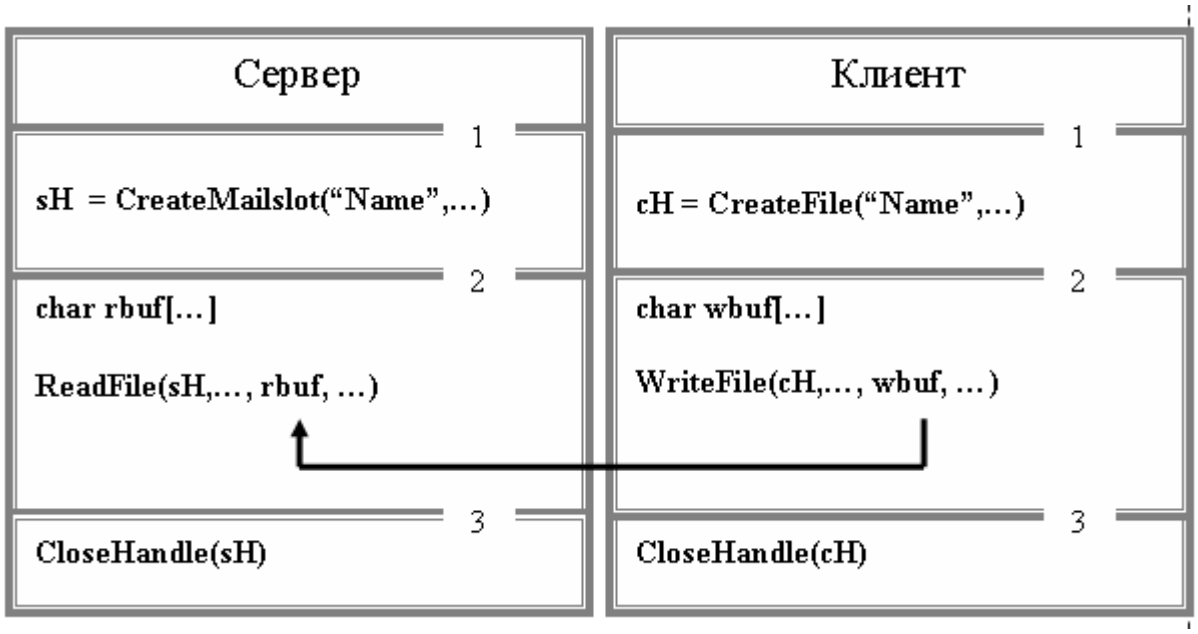

Рисунок 5.2.1. Схема взаимодействия процессов использующих Mailslot API

В первом блоке программы сервера выполняется функция CreateMailslot, создающая почтовый ящик. В случае успешного завершения функция возвращает дескриптор почтового ящика, который будет использоваться дальше. Кроме того, один из параметров функции CreateMailslot определяет время ожидания функцией ReadFile, очередного сообщения от клиента. В простейшем случае можно установить бесконечное время ожидания. Во втором блоке сервера осуществляется считывание данных из почтового ящика. В последнем третьем блоке сервера закрывается дескриптор почтового ящика, что приводит к его уничтожению.

В первом блоке программы клиента осуществляется подсоединение клиента к почтовому ящику с помощью функции CreateFile (открыть почтовый ящик). В случае успешного выполнения, функции формирует дескриптор почтового ящика, который потом используется функцией WriteFile (второй блок клиента) для записи данных в почтовый сервер. При завершении программы, следует закрыть дескриптор почтового ящика с помощью функции CloseHandle.

В принципе, между процессами обмен данными можно организовать в обе стороны. Для этого необходимо в рамках каждого процесса создать свой почтовый ящик, который бы использовался для приема сообщений.

5.3. Создание почтового ящика

Для создания почтового ящика используется функция CreateMailslot (рисунок 5.3.1).

```
// -- создать почтовый ящик
// Назначение: функция  предназначена для создания почтового
//             ящика

 HANDLE CreateMailslot
           (
     LPCTSTR        pname,    // [in] символическое имя ящика
     DWORD          maxms,    // [in] максимальная длина сообщения
     DWORD          timeo,    // [in] интервал ожидания
     LPSECURITY_ATTRIBUTES sattr // [in] атрибуты безопасности
            );

// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор почтового ящика, иначе
//       значение  INVALID_HANDLE_VALUE
// Примечание: pname - указывает на строку именем канала в
//       локальном формате;
//       timeo – параметр устанавливает время ожидания
//       собщения  функцией ReadFile; для задания бесконечного
//       ожидания, следует установить значение
//       MAILSLOT_WAIT_FOREVER;
//       sattr – для установки атрибутов безопасности
//       по умолчанию  следует установить значение NULL
```

Рисунок 5.3.1. Функция CreateMailslot

Параметр функции CreateMailslot, задающий имя почтового ящика, подразумевает, что это имя задано в локальном формате. На рисунках 5.3.2 – 5.3.4 указаны три возможных формата имени почтового ящика.

```
\\.\mailslot\xxxxx
где: точка (.) - обозначает  локальный компьютер;
     mailslot  - фиксированное слово;
     xxxxx     - имя почтового ящика
```

Рисунок 5.3.2. Локальный формат имени почтового ящика

Локальный формат имени почтового ящика используется при создании почтового ящика (ящик всегда создается на локальном для сервера компьютере), а также программой клиентом при открытии ящика, если предполагается использовать для записи все ящики с заданным именем на одном локальном компьютере.

Сетевой формат имени почтового ящика используется программой клиента, для записи сообщений в группу одноименных почтовых ящиков, которые находятся на компьютере, указанном в имени.

```
\\servname\mailslot\xxxxx
где: servname  - имя  компьютера-сервера почтового ящика;
     mailslot  - фиксированное слово;
     xxxxx     - имя почтового ящика
```

Рисунок 5.3.3. Сетевой формат имени почтового ящика

```
\\domain\mailslot\xxxxx
где: domain    - имя  домена компьютеров или *;
     mailslot  - фиксированное слово;
     xxxxx     - имя почтового ящика
```

Рисунок 5.3.4. Доменный формат имени почтового ящика

Доменный формат имени почтового ящика используется программой клиента для записи сообщений в группу одноименных почтовых ящиков, которые находятся на всех компьютерах указанного домена. Если необходимо записать в сообщение в группу почтовых ящиков, которые находятся на компьютерах первичного домена, то вместо имени домена можно указать символ *.

5.4. Соединение клиентов с почтовым ящиком

Для установки связи с почтовым ящиком программа клиента использует функцию CreateFile (рисунок 5.4.1).

Как уже отмечалось, функция CreateFile является универсальной и значения ее параметров практически ничем не отличаются от значений, применяющихся для связи клиента именованного канала с сервером именованного канала. В описании функции и приведенных примерах не рассматриваются никакие параметры, определяющие атрибуты безопасности. Использование атрибутов безопасности установленных по умолчанию, приводит к тому, что связь может быть установлена только между процессами, которые запущены от одного имени и с одним общим паролем. Для знакомства с возможностями интерфейса Mailslots, связанными с системой безопасности операционной системы Windows рекомендуется обратиться к источникам [4] или http://msdn2.miscrosoft.com

Следует обратить внимание на формат имени открываемого почтового ящика. Этот формат определяет пространство поиска почтовых ящиков, с которыми будет установлена связь.

```
// -- открыть почтовый ящик
// Назначение: функция  предназначена для подключения клиента
//             к  почтовому ящику

 HANDLE CreateFile
                  (
     LPCTSTR    mname, // [in] символическое имя почтового ящика
     DWORD      accss, // [in] чтение или запись
     DWORD      share, // [in] режим совместного использования
     LPSECURITY_ATTRIBUTES sattr // [in] атрибуты безопасности
     DWORD      oflag, // [in] флаг открытия почтового ящика
     DWORD      aflag, // [in] флаги и атрибуты
     HANDLE     exten, // [in] дополнительные атрибуты
                  );

// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор именованного канала, иначе
//       INVALID_HANDLE_VALUE -  неудачное завершение
// Примечание:- параметр mname указывается в локальном,
//       сетевом или доменном  формате:  в зависимости от
//       способа применения;
//     - параметр accss  должен принимать значение
//        GENERIC_WRITE
//     - параметр share может принимать значения
//        FILE_SHARE_READ (совместное чтение),
//        FILE_SHARE_WRITE (совместная запись),
//       FILE_SHARE_READ| FILE_SHARE_WRITE (чтение и запись);
//     - параметр sattr для установки атрибутов безопасности
//       по умолчанию, следует установить значение NULL;
//     - значение параметра oflag всегда устанавливается  в
//       OPEN_EXISTING(открытие существующего ящика);
//     - значение параметра aflag можно установить в NULL,
//       что определяет значения флагов и атрибутов  по
//       умолчанию или установить FILE_ATTRIBUTE_NORMAL;
//     - значение параметра exten следует становить в  NULL
```

Рисунок 5.4.1. Функция CreateFile

5.5. Обмен данными через почтовый ящик

Для записи данных в почтовый ящик используется функция WriteFile, а для чтения данных из почтового ящика функция ReadFile. Значения параметров, используемые в этих универсальных функциях при работе с почтовыми ящиками, практически ничем не отличаются от значений, применяемых при работе с именованными каналами. Разница заключается лишь в том, что в одном случае функции используют дескрипторы каналов, а другом – дескрипторы почтовых ящиков. Кроме того, следует помнить, что

функцию ReadFile может выполнять только программа сервера, а WriteFle могут выполнять и сервер (сервер может записывать в свой собственный ящик) и клиент.

```
//.............................................................
HANDLE hM; // дескриптор почтового ящика
DWORD rb;                              // длина почитанного сообщения
char rbuf[100];                        // буфер ввода
try
{
   if ((hM = CreateMailslot("\\\\.\\mailslot\\myslot",
                NULL,
                MAILSLOT_WAIT_FOREVER,       // ждать вечно
                NULL)) == INVALID_HANDLE_VALUE)
                   throw "CreateMailslotError";
  if(!ReadFile(hM,
              rbuf,                          // буфер
              sizeof(rbuf),                  // размер буфера
              &rb,                           // прочитано
              NULL))
                    throw "ReadFileError";
//.............................................................
}
//.............................................................
```

Рисунок 5.5.1. Создание почтового ящика

```
//.............................................................
HANDLE hM; // дескриптор почтового ящика
DWORD wb;                          // длина записанного сообщения
char wbuf[] = "Hello Mailslot";  // буфер вывода
try
{
   if ((hM = CreateFile("\\\\isit301\\mailslot\\myslot",
                 GENERIC_WRITE,    // будем писать в ящик
                 FILE_SHARE_READ,  // разрешаем одновременно читать
                 NULL,
                 OPEN_EXISTING,    //  ящик уже есть
                 NULL, NULL)) == INVALID_HANDLE_VALUE)
                 throw "CreateFileError";
   if(!WriteFile(hM,
                 wbuf,               // буфер
                 sizeof(wbuf),       // размер буфера
                 &wb,                // записано
                 NULL))
                  throw "ReadFileError";

//.............................................................
}
//.............................................................
```

Рисунок 5.5.2. Соединение клиента с почтовым ящиком

На рисунках 5.5.1 и 5.5.2 представлены фрагменты программ сервера и клиента. В программе сервера создается почтовый ящик и читается сообщение из него. В программе клиента осуществляется подключение к почтовому ящику и записывается в него сообщение. Следует обратить внимание, что программы клиента и сервера находятся на разных компьютерах, т.к. символическое имя почтового ящика в функции CreateFile указано в сетевом формате.

5.6. Получение информации о почтовом ящике

Получить информацию о характеристиках почтового ящика можно с помощью функции GetMailslotInfo (рисунок 5.6.1).

```
// -- получить информацию о почтовом ящике
// Назначение: функция  предназначена для получения
//             характеристик созданного почтового ящика

 BOOL GetMailslotInfo
                  (
     HANDLE     hM,    // [in] дескриптор почтового ящика
     LPDWORD    ml,    // [out] максимальная длина сообщения
     LPDWORD    nl,    // [out] длина следующего сообщения
     LPDWORD    nm     // [out] количество сообщений
     LPDWORD    to     // [out] интервал ожидания сообщения
                 );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
```

Рисунок 5.6.1. Функция GetMailslotInfo

Функция GetMailslotInfo может быть использована только на стороне сервера почтового ящика и параметр hM должен быть получен в результате выполнения функции CreateMailslot. Чаще всего функция применяется для выяснения количества непрочитанных сообщений накопившихся в почтовом ящике.

5.7. Изменение интервала ожидания сообщения

Время ожидания функцией ReadFile поступления сообщения в почтовый ящик первоначально устанавливается при создании почтового ящика с помощью функции CreateMailslot. В процессе работы, может оказаться необходимым изменить значение этого интервала или вообще сделать его нулевым. Для этого применяется функция SetMailslotInfo (рисунок 5.6.1). Функция может быть выполнена только в программе сервера и использует в качестве аргумента дескриптор почтового сервера, который был получен при выполнении функции CreateMailslot.

```
// -- изменить время ожидания сообщения
// Назначение: функция  предназначена для изменения
     //             одной характеристики почтового ящика –
//              интервала времени ожидания

 BOOL SetMailslotInfo
                  (
     HANDLE      hM, // [in] дескриптор почтового ящика
     PDWORD      to  // [in] новое значение интервала
                  );

// Код возврата: в случае успешного завершения функция
//       возвращает TRUE, иначе FALSE
```

Рисунок 5.7.1. Функция SetMailslotInfo

5.8. Итоги главы

1. Механизм Mailslots (почтовый ящик) является одним из IPC-механизмов операционной системы Windows, позволяющий создавать распределенные приложения архитектуры клиент-сервер в локальной сети TCP/IP.
2. Почтовый ящик представляет собой объект операционной системы, предоставляющий возможность пересылать данные в одном направлении: от клиента к серверу.
3. Почтовый ящик идентифицируется своим именем. Сервером называется процесс создающий почтовый ящик. Клиентом – процесс, который подключается к почтовому ящику и записывает в него данные.
4. Обмен данными осуществляется сообщениями и может происходить в синхронном и асинхронном режимах. Если клиент и сервер находятся на разных компьютерах, доставка сообщений не гарантируется.
5. Допускается создание нескольких ящиков с одним и тем же именем. Если пересылаемые сообщения не превышают 425 байт, то возможна передача данных одновременно нескольким почтовым ящикам.
6. В состав Mailslots API входят функции создания почтового ящика, подсоединения клиента к почтовому ящику, функции записи и чтения сообщений, а также функции для получения и установки характеристик почтового ящика.

Глава 6. Разработка параллельного сервера

6.1. Предисловие к главе

В предыдущих главах были рассмотрены основные механизмы и интерфейсы операционной системы Windows, позволяющие осуществлять обмен данными между распределенными в сети TCP/IP процессами. Приведенные примеры демонстрировали простейшие распределенные приложения (один сервер – один клиент), созданные на основе этих интерфейсов. Для разработки более сложных распределенных приложений в среде Windows, кроме этого требуются еще специальные методы и приемы программирования.

Основной целью этой главы является изучение методов и приобретение навыков разработки распределенных приложений с архитектурой клиент-сервер, но имеющих более сложную, чем один сервер – один клиент структуру. На рисунке 6.1.1 изображена структура распределенного приложения, на создание которого будет в основном ориентировано дальнейшее изложение материала.

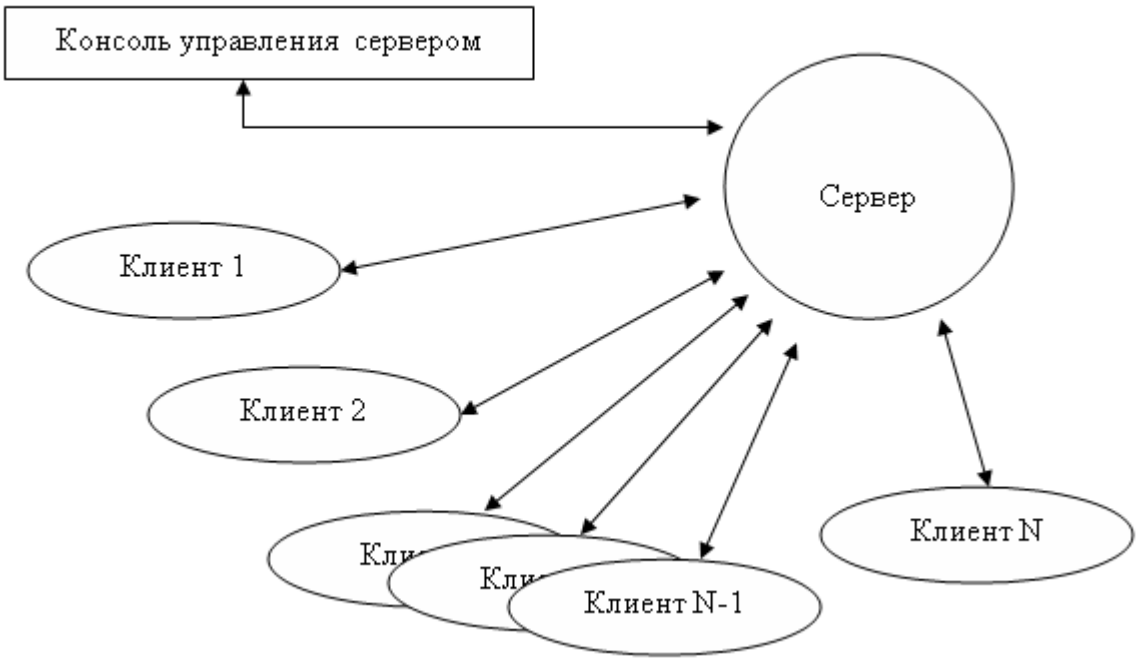

Рисунок 6.1.1. Структура распределенного приложения с сервером, обслуживающим одновременно несколько клиентов

Распределенное приложение, изображенное на рисунке 6.1.1 предполагает наличие одной программы сервера, которая одновременно обслуживает несколько программ клиентов. Управление сервером осуществляется с помощью специальной программы, которую будем называть консолью управления.

Серверы, одновременно обслуживающие несколько клиентов, по методу облуживания подразделяются на ***итеративные*** и ***параллельные*** серверы ***(iterative and concurrent servers)***.

Работа итеративного сервера описывается циклом из четырех шагов: 1) ожидание запроса от клиента; 2) обработка запроса; 3) отправка результата запроса; 4) возврат в ждущее состояние 1. Очевидно, что сервер этого класса может применяться в том случае, если предполагаются короткие запросы от клиентов, не требующие больших затрат на обработку и длинных ответов сервера. Как правило, подобные серверы работают над UDP, когда нет необходимости создавать отдельный канал связи для каждого клиента. Консоль управления в этом случае может быть выполнена в виде специального клиента, запросы которого и есть команды управления сервером.

Параллельные серверы имеют другой цикл работы: 1) ожидание запроса от клиента; 2) запуск нового сервера для обработки текущего запроса; 3) возврат в ждущее состояние 1. Преимущество параллельных серверов заключается в том, что он лишь порождает новые серверы, которые и занимаются обработкой запросов клиентов. Очевидно, что для создания параллельных серверов необходимым условием является мультизадачность операционной среды сервера. По всей видимости, параллельные серверы целесообразно использовать, если предполагается наличие относительно длительного сеанса связи между клиентом и сервером. Как правило, параллельные серверы работают над TCP. Консоль управления может быть создана, как отдельный процесс или поток (в зависимости от возможностей операционной системы) в рамках сервера или так же, как предлагалось для итеративного сервера, выполнить в виде специализированного клиента.

Дальнейшее изложение, в основном, будет посвящено разработке параллельных серверов. При этом мы будем говорить, что в рамках одного сервера работает несколько процессов, работающих параллельно (в одно и тоже время) и выполняющих специфические функции в рамках сервера.

Следует обратить внимание, что определенную путаницу может внести применяемая терминология. Дело в том, что любой процесс сервера может быть реализован, как ***поток*** или как ***процесс*** операционной системы (подразумеваются операционные системы семейства Windows 200x/XP). Поэтому, если в тексте используется понятие “***процесс операционной системы”*** – это будет специально оговариваться. В другом случае, под понятием “***процесс сервера”***, подразумевается часть программы сервера работающая параллельно с другими частями (процессами сервера) независимо о способа реализации.

6. 2. Особенности разработки параллельного сервера

Разработчик параллельного сервера сталкивается с рядом проблем, которые обусловлены необходимостью одновременно обслуживать несколько клиентов. Для этого требуется в рамках программы сервера организовать совместную работу нескольких процессов (еще раз напомним о

терминологическом соглашении), каждый из которых предназначен для обслуживания подключившегося клиента или для выполнения каких-то внутренних задач сервера.

Критическим по времени для параллельного сервера является момент подключения клиента. Сервер не должен тратить много времени на подключение клиента, т.к. в этот момент могут осуществлять подключение другие клиенты, которые из-за занятости сервера могут получить отказ. Поэтому целесообразно в сервере выделить отдельный процесс (и желательно, чтобы он имел наивысший приоритет), который бы был занят только подключением клиентов к серверу.

С другой стороны, может потребоваться управлять процессом подключения. Например, если оператор консоли управления ввел команду, запрещающую подключение новых клиентов к серверу. В этом случае необходимо как-то вмешаться в этот процесс и запретить подключения новых клиентов до получения команды вновь разрешающей подключение к серверу клиентов.

Так как процесс подключения клиента должен выполняться быстро, то загружать его другой работой не целесообразно. Но в работе параллельного сервера необходимо выполнять еще ряд действий, не связанных с подключением клиентов. Например, управление сервером с консоли подразумевает цикл ожидания, ввода и обработки команд оператора. По всей видимости, подобные действия следует выполнять в отдельных процессах.

После подключения клиента к параллельному серверу запускается отдельный процесс, обслуживающий запрос клиента. Для управления обслуживающими процессами и для сбора статистики требуется динамическая структура данных (позволяющая добавлять и удалять элементы структуры), предназначенная для хранения информации о работающих в настоящий момент обслуживающих процессах. Как правило, для хранения подобной информации используют связный список.

Отдельные процессы, работающие в рамках параллельного сервера, могут использовать общие ресурсы. Некоторые ресурсы, могут быть разрушены при совместном использовании несколькими параллельными процессами (например, связный список). Для разрешения этой проблемы следует использовать механизмы синхронизации, позволяющие последовательно использовать такие критические ресурсы.

6.3. Структура параллельного сервера

Структура параллельного сервера, зависит, в конечном счете, от характера решаемой сервером задачи, но все же существуют общие структурные свойства сервера, на которых следует остановиться. Для того, чтобы упростить изложение будем рассматривать далее конкретную реализацию (модель) параллельного сервера с именем ConcurrentServer (рисунок 6.3.1) структура которого, по мнению автора, отражает все требующие внимания моменты.

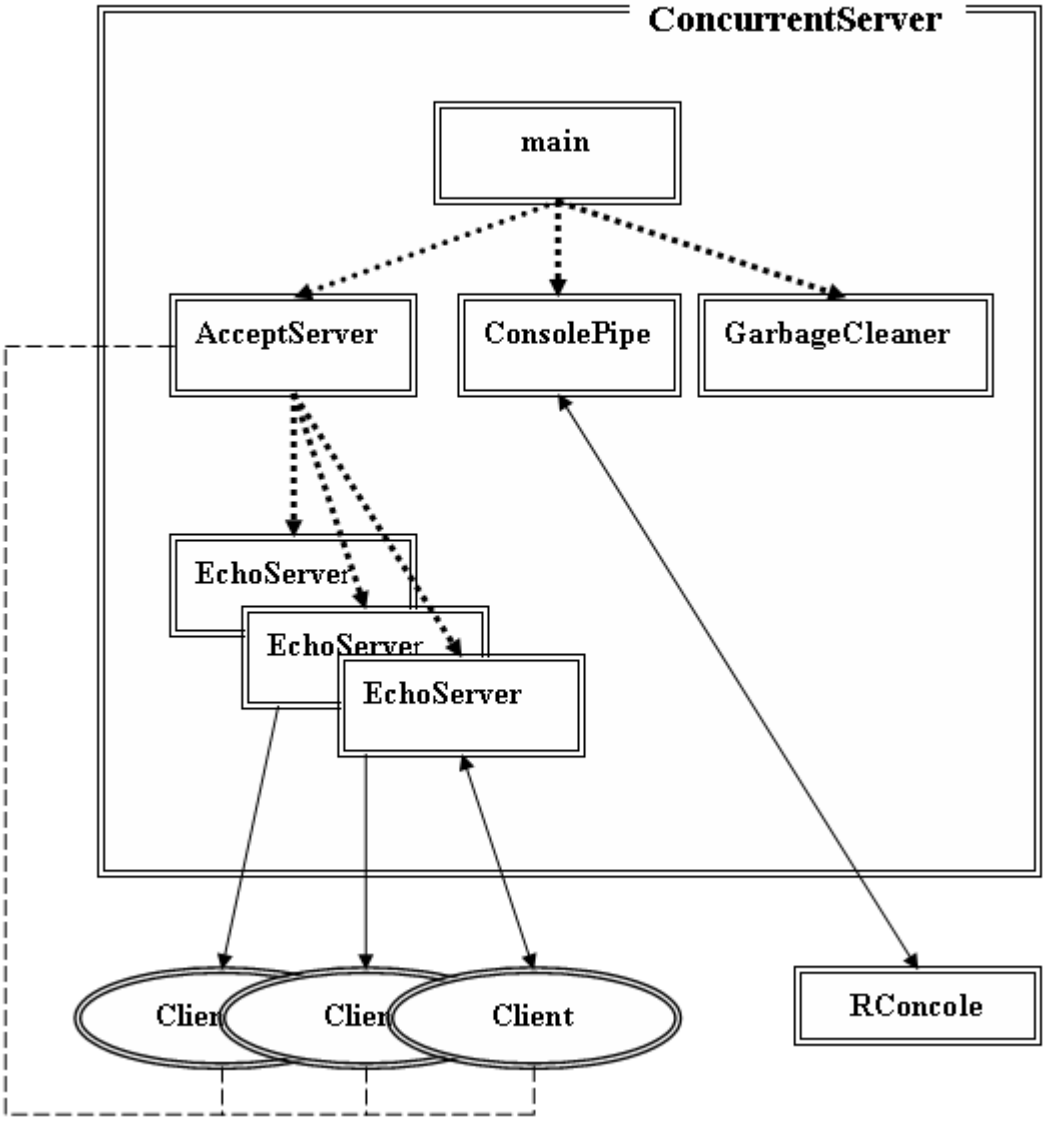

Рисунок 6.3.1. Структура параллельного сервера

На рисунке 6.3.1 изображена структура параллельного сервера, назначением которого является, одновременное обслуживание нескольких клиентских программ. Обслуживание заключается в получении от клиента по установленному TCP-соединению последовательности символов и в возврате (пересылке) этой последовательности обратно. Кроме того, предполагается, что сервер может выполнять команды, введенные с консоли управления, с которой поддерживается связь через именованный канал (Named Pipe).

Все процессы, работающие в рамках сервера, изображены на рисунке прямоугольниками. Пунктирными направленными линиями обозначается создание (запуск) одного процесса другим. Процесс с именем main, является главным процессом сервера, который получает управление от операционной системы. Этот процесс создает и запускает новые процессы AcceptServer, ConsolePipe и GarbageCleaner. Процесс AcceptServer, в свою очередь создает

несколько процессов с именем EchoServer. Сплошными двунаправленными линиями обозначается перемещение данных. Данные перемещаются между EchoServer-процессами сервера и клиентскими программами (с именем Client), обозначенными на рисунке овалами, а также между программой с именем RConsole, реализующей клиентскую часть консоли управления, и процессом ConsolePipe, который реализует серверную часть консоли управления. Штриховой линией, соединяющей изображение клиентских программ и изображение процесса AcceptServer, обозначается процедура создания соединения между клиентом и сервером.

Опишем назначение компонентов изображенного на рисунке 6.3.1 распределенного приложения.

Процесс main. Основным назначением процесса main, является запуск, инициализация и завершение работы сервера. Как уже отмечалось, именно этот процесс первым получает управление от операционной системы. Процесс main запускает основные процессы: AcceptServer, ConsolePipe и RConsole.

Процесс AcceptServer. AcceptServer создается процессом main и предназначен для выполнения процедуры подключения клиентов к серверу, для исполнения команд консоли управления, а также для запуска процессов EchoServer, обслуживающих запросы клиентских программ по созданным соединениям. Кроме того, AcceptServer создает список подключений, который далее будем называть ***ListContanct***. При подключении очередного клиента, процесс AcceptServer добавляет в ListContact элемент, предназначенный для хранения информации о состоянии данного подключения.

Процесс ConsolePipe. ConsolePipe создается процессом main и является сервером именованного канала, по которому осуществляется связь между программой RConsole (консоль управления сервером) и параллельным сервером.

Процесс GarbageCleaner. Основным назначением процесса GarbageCleaner является удаление элемента списка подключений ListContact, после отключения программы клиента. Следует сразу отметить, что ListContact является ресурсом, требующим последовательного использование. Одновременная запись и (или) удаление элементов списка может привести к разрушению списка ListContact.

Процесс EchoServer. Процессы EchoServer создаются процессом AcceptServer по одному для каждого успешного подключения программы клиента. Основным назначением процесса EchoServer является прием данных по созданному процессом AcceptServer подключению и отправка этих же данных без изменения обратно программе клиента. Условием окончания работы сервера является получение от клиента пустого сегмента данных (имеющего нулевую длину).

Программа Client. Программа Client предназначена для пересылки данных серверу и получения ответа от сервера. Программа может работать, как на одном компьютере с сервером (будет использоваться интерфейс

внутренней петли), так и на другом компьютере, соединенным с компьютером сервера сетью TCP/IP. Для окончания работы с сервером программа формирует и отправляет сегмент данных нулевой длины.

Программа RConsole. Программа RConsole предназначена для ввода команд управления сервером и для вывода диагностических сообщений полученных от сервера . RConsole является клиентом именованного канала.

Список подключений ListContact. Список ListContact (не изображен на рисунке) создается основе стандартного класса list и предназначен для хранения информации о каждом подключении. Список создается пустым при инициализации процесса AcceptServer. В рамках этого же процесса осуществляется добавление элементов списка, по одному для каждого подключения. При отключении программы клиента от сервера, соответствующий элемент списка помечается, как неиспользуемый. Удаление неиспользуемого элемента осуществляется процессом GarbageCleaner, который работает в фоновом режиме.

Описанная выше модель распределенного приложения, по мнению автора, является достаточно полной для того, чтобы изложить основные принципы создания параллельного сервера. Дальнейшее изложение материала будет опираться на эту модель.

6.4. Потоки и процессы в Windows

В описанной выше модели параллельного сервера с именем ConcurrentServer, предполагается запуск процессов, работающих параллельно. Сначала процесс main запускает три параллельно работающих процесса (AcceptServer, ConsolePipe и GarbageCleaner), а потом еще процесс AcceptServer осуществляет запуск нескольких процессов (по одному для каждого подключения) EchoServer.

Для организации параллельной работы программ в операционной системе Windows предусмотрены два специальных механизма: ***механизм потоков*** и ***механизм процессов*** .

Понятие потока тесно связано с последовательностью действий процессора во время выполнения программы. Исполняя программу, процессор последовательно выполняет инструкции (машинные коды) программы, иногда осуществляя переходы. Такая последовательность выполнения инструкций называется ***потоком управления (thread – нить)***. Будем говорить, что программа является ***многопоточной***, если в ней существуют одновременно несколько потоков управления. Сами потоки в этом случае называются ***параллельными***. Если в программе может существовать только один поток, то такая программа называется ***однопоточной***.

В рамках потока управления многопоточной программы могут вызываться функции. При вызове одной и той же функции в разных потоках управления, важно чтобы эта функция обладала свойством ***безопасности для потоков***. Безопасная для потоков функции в обладает двумя свойствами: 1) свойством ***реентерабельности***; 2) функция обеспечивает

блокировку доступа к критическим ресурсам, которые она использует.

В общем случае функция называется реентерабельной, если она не изменяет собственный код или собственные статические данные. Другими словами программный код реентерабельной функции должен допускать корректное его использование несколькими потоками одновременно.

Блокировка требуется в том случае, если функцией используется ресурс, доступ к которому может быть только упорядоченным (критический ресурс). Примером критического ресурса может служить изменяемые функцией статические и глобальные переменные.

Каждое приложение, работающее в среде Windows, имеет, по крайней мере, один поток, который называется ***первичным*** или ***главным*** потоком. В консольных приложениях этот поток исполняет функцию ***main***. В приложениях с графическим интерфейсом это поток, который исполняет функцию ***WinMain***.

Поток управления в Windows является объектом ядра операционной системы, которому выделяется процессорное время для выполнения приложения. Каждому потоку принадлежат следующие ресурсы:

- код исполняемой функции;
- набор регистров процессора;
- область оперативной памяти;
- стек для работы приложения;
- стек для работы операционной системы;
- маркер доступа, содержащий информацию для системы безопасности.

Все эти ресурсы образуют так называемый ***контекст потока*** в Windows.
Основные функции для работы с потоками перечислены в таблице 6.4.1.

Таблица 6.4.1

Наименование функции	**Назначение**
`CreateThread`	Создать поток
`ResumeThread`	Возобновить поток
`SuspendThread`	Приостановить поток
`Sleep`	Задержать исполнение
`TermitateThread`	Завершить поток

Для создания потоков в Windows используется функция CreateThread, описание которой приводится на рисунке 6.4.1. Третий параметр функции (pFA) указывает на функцию, которая первая получит управление в созданном потоке. Функция потока принимает только один параметр, значение которого передается с помощью четвертого параметра (pPrm). Функция потока должна завершаться вызовом функции ExitThread, с параметром, устанавливающим код возврата. Пример правильного оформления функции потока и именем AcceptServer приведен на рисунке 6.4.2.

```
// -- создать  поток
// Назначение: функция  предназначена для создания  потока

 HANDLE CreateThread(
      PSECURITY_ATTRIBUTES   pSA,   // [in] атрибуты защиты
      DWORD                  sSt,   // [in] размер стека потока
      LPTHREAD_START_ROUTINE pFA,   // [in] функция потока
      LPVOID                 pPrm,  // [in] указатель на  параметр
      DWORD                  flags, // [in] индикатор запуска
      LPDWORD                pId    // [out] идентификатор потока

                  );
// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор потока; иначе возвращается
//       значение NULL
// Примечания: - значение параметра pSA может принимать
//      значение NULL, в этом случае параметры защиты потока
//      будут установлены по умолчанию;
//       - значение параметра sSt может принимать значение
//      NULL, в этом случае размер стека потока будет
//      установлен по умолчанию (1МБ);
//       - если значение параметра flags установлено NULL, то
//      функция потока начнет выполняться сразу после создания
//      потока;  если же установлено значение CREATE_SUSPENDED,
//      то поток остается в состоянии готовности к исполнению
//      функции до вызова функции ResumeThread;
//      - возвращаемое значение идентификатора потока может
//      быть использовано при вызове  некоторых функций
//      управления потоком; допускается установка  NULL для
//      параметра  pId -  в этом случае идентификатор не
//      возвращается
```

Рисунок 6.4.1. Функция CreateThread

```
DWORD WINAPI AcceptServer (LPVOID pPrm)     // прототип
{
 DWORD rc = 0;     // код возврата

 //.........................................
 ExitThread(rc);  // завершение работы потока
}
```

Рисунок 6.4.2. Структура функции потока

На рисунке 6.4.3 приводится пример использования функции CreateThread. В этом примере главный поток main создает три потока с потоковыми функциями AcceptServer, ConsolePipe, GarbageCleaner. Следует обратить внимание, что после запуска потоков, перед завершением функции (и потока) main, три раза вызывается функция WaitForSingleObject у которой в качестве первого параметра используется дескриптор потока, а

второй параметр имеет значение, определенное константой INFINITE. Такой вызов функции WaitForSingleObject приостанавливает выполнение основного потока main до завершения работы потока, соответствующего указанному в параметре дескриптору. Отсутствие этих функций могло бы привести к завершению потока main, до завершения порожденных им потоков. В этом случае порожденные потоки тоже автоматически завершаются операционной системой. После завершения работы потока следует освободить связанные с потоком ресурсы с помощью функции CloseHandle.

```
 #include <windows.h>     // для функций управления потоками
//.............................................................
HANDLE hAcceptServer,    // дескриптор потока AcceptServer
       hConsolePipe,     // дескриптор потока ConsolePipe
       hGarbageCleaner   // дескриптор потока GarbageCleaner
DWORD WINAPI AcceptServer(LPVOID pPrm);  // прототипы функций
DWORD WINAPI ConsolePipe(LPVOID pPrm);
DWORD WINAPI GarbageCleaner(LPVOID pPrm);

int _tmain(int argc, _TCHAR* argv[])        //главный поток
{
 volatile TalkersCommand  cmd = START;      // команды сервера
 hAcceptServer   = CreateThread(NULL,NULL, AcceptServer,
                                (LPVOID)&cmd, NULL, NULL),
 hConsolePipe    = CreateThread(NULL,NULL, ConsolePipe,
                                (LPVOID)&cmd, NULL, NULL),
 hGarbageCleaner = CreateThread(NULL,NULL,GarbageCleaner,
                                 (LPVOID)NULL, NULL, NULL);

WaitForSingleObject(hAcceptServer,INFINITE);
CloseHandle(hAcceptServer);
WaitForSingleObject(hConsolePipe,INFINITE);
CloseHandle(hConsolePipe);
WaitForSingleObject(hGarbageCleaner,INFINITE);
CloseHandle(hConsolePipe);
return 0;
};
```

Рисунок 6.4.3. Пример использования функции CreateThread

Другой важный момент в приведенном примере, на который следует обратить внимание, это применение квалификатора volatile. Оператор volatile указывает компилятору на необходимость размещения переменной cmd в памяти и не осуществлять относительно этого размещения никакой оптимизации. Дело в том, что область памяти отведенная переменной cmd, используется двумя параллельно работающими потоками AcceptServer и ConsolePipe. Поэтому оптимизация может привести к тому, что потоки будут использовать различные области памяти.

Один поток может завершить другой поток с помощью функции TerminateThread (рисунок 6.4.4). Использовать эту функцию следует только в аварийных ситуациях, т.к. завершение потока подобным образом не освобождает распределенные операционной системой ресурсы.

```
// -- завершить  поток
// Назначение: функция  предназначена для завершения  потока
//             без освобождения ресурсов

 BOOL TerminateThread(
      HANDLE   hT,   // [in] дескриптор потока
      DWORD    rc    // [in] код завершения потока
                    )
// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение; иначе возвращается
//       значение NULL
```

Рисунок 6.4.4. Функция TerminateThread

Исполнение каждого потока может быть приостановлено с помощью функции SuspendThread (рисунок 6.4.5). С каждым созданным потоком связан специальный счетчик, показывающий сколько раз была выполнена приостановка потока функцией SuspendThread. Максимальное значение счетчика приостановок равно MAXIMUM_SUSPEND_COUNT. Поток исполняется только в том случае если значение счетчика приостановок рано нулю.

```
// -- приостановить  поток
// Назначение: функция  предназначена для временной
//             приостановке исполнения  потока

 DWORD SuspendThread(
      HANDLE   hT,   // [in] дескриптор потока
                    )
// Код возврата: в случае успешного завершения функция
//       возвращает текущее значение счетчика приостановок;
//       иначе возвращается значение -1
```

Рисунок 6.4.5. Функция SuspendThread

Для уменьшения текущего значения счетчика приостановок потока используется функция ResumeThread (рисунок 6.5.6).

```
// -- возобновить поток
// Назначение: функция  предназначена для уменьшения счетчика
//             приостановки потока на единицу

 DWORD ResumeThread(
      HANDLE   hT,   // [in] дескриптор потока
                   )
// Код возврата: в случае успешного завершения функция
//       возвращает текущее значение счетчика приостановок;
//       иначе возвращается значение -1
// Примечание: если значение счетчика приостановок после
//       выполнения функции станет равным нулю, то поток
//       возобновляет свою работу с того места, где он был
//       приостановлен функцией SuspendThread
```

Рисунок 6.4.6. Функция ResumeThread

Полезной, особенно на этапе отладки, является функция Sleep (рисунок 6.4.7), с помощью которой поток может задержать свое исполнение на заданный интервал времени.

```
// -- задержать поток
// Назначение: функция  предназначена для задержки исполнения
//             потока на заданный интервал времени

 VOID  Sleep(
      DWORD   ms    // [in]  интервал времени в миллисекундах
            );
```

Рисунок 6.4.7. Функция Sleep

Под ***процессом*** операционной системы Window понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением операционной системы. Выполнение процесса начинается с первичного потока main. Во время выполнения, процесс может создавать новые потоки и порождать новые процессы. С точки зрения техники программирования, работа с процессами очень напоминает работу с потоками с одним существенным отличием. Отличие заключается в том, что потоки, работающие в рамках одного процесса, разделяют общее пространство памяти, а каждый процесс имеет свое собственное пространство. Поэтому для взаимодействия между различными процессами операционной системы (обмен данными, синхронизация и т.п.) надо использовать средства из разряда IPC, о которых уже говорилось раньше. Создание и запуск отдельного процесса по затратам ресурсов значительно превосходит затраты на создание потока в рамках существующего процесса. Кроме того, более затратными оказывается процедура переключения с одного процесса на другой. Все эти особенности приводят к значительной

потере производительности параллельного сервера, использующего механизм процессов операционной системы, по сравнению с сервером, сделанного с применением механизма потоков.

При разработке параллельных серверов использовать механизмы управления процессами целесообразно с тех случаях, когда в рамках сервера используются такие компоненты сервера (чаще всего это обслуживающие процессы), которые могут разрушить работу всего параллельного сервера. В этом случае, действительно, целесообразно реализовать эти компоненты сервера как отдельные процессы операционной системы Windows.

Дальнейшее изложение ориентировано на разработку сервера с применением механизма потоков операционной системы Windows. С использованием механизма процессов Windows можно ознакомиться в [4, 14].

6.5. Синхронизация потоков параллельного сервера

С некоторыми механизмами синхронизации используемыми в Windows мы уже познакомились, когда говорили о функции WaitSingleObject, которая использовалась для ожидания окончания работы потока. Еще один рассмотренный способ синхронизации – приостановка потока с помощью функции SuspendThread.

Критическим ресурсом, который можно использовать только последовательно, в нашей модели параллельного сервера является список подключений ListContact. На рисунке 6.5.1 предлагается пример реализации такого списка с помощью стандартного класса list [13].

Рассмотрим структуру элемента списка подключений (структура Contact). Два поля type и sthread предназначены для описания состояния соединения с клиентом на разных этапах: type – на этапе подключения; sthread – на этапе обслуживания. Поля s, prms, lprms используются для хранения параметров соединения. Для хранения дескриптора обсуживающего потока (в нашей модели потока EchoServer) используется поле hthread. Дескриптор htimer, может быть использован, для организации ожидающего таймера, позволяющего ограничить время работы обслуживающего процесса. Поля msg, srvname могут использоваться обслуживающими потоками для записи диагностирующего сообщения и символических имен обрабатывающих потоков.

Синхронизация будет осуществляться между двумя потоками: AcceptServer и GarbageCleaner. Как уже описывалось выше, синхронизация необходима в связи с тем, что одновременное добавление элемента в список ListContact, выполняемое потоком AcceptServer, и удаление элемента списка, выполняемое потоком GarbageCleaner может привести к непредсказуемым последствиям.

```
#include <list>
#include "Winsock2.h"
//.....................................................
using namespace std;

struct Contact         // элемент списка подключений
{
 enum TE{              // состояние  сервера подключения
    EMPTY,             // пустой элемент списка подключений
    ACCEPT,            // подключен (accept), но не обслуживается
    CONTACT            // передан обслуживающему серверу
         }     type;   // тип элемента списка подключений
 enum ST{              // состояние обслуживающего сервера
    WORK,              // идет обмен данными с клиентом
    ABORT,             // обслуживающий сервер завершился не нормально
    TIMEOUT,           // обслуживающий сервер завершился по времени
    FINISH             // обслуживающий сервер завершился  нормально
       }       sthread; // состояние  обслуживающего сервера (потока)

 SOCKET      s;         // сокет для обмена данными с клиентом
 SOCKADDR_IN prms;      // параметры  сокета
 int         lprms;     // длина prms
 HANDLE      hthread;   // handle потока (или процесса)
 HANDLE      htimer;    // handle таймера

 char msg[50];          // сообщение
 char srvname[15];      //  наименование обслуживающего сервера

 Contact( TE t = EMPTY, const char* namesrv = "") // конструктор
     {memset(&prms,0,sizeof(SOCKADDR_IN));
      lprms = sizeof(SOCKADDR_IN);
      type = t;
      strcpy(srvname,namesrv);
      msg[0] = 0;};

 void SetST(ST sth, const char* m = "" )
      {sthread = sth;
       strcpy(msg,m);}
   };
typedef list<Contact> ListContact;          // список подключений
```

Рисунок 6.5.1. Пример реализации списка подключений ListContact

Операционная система Windows обладает широким спектром механизмов синхронизации потоков и процессов: критические секции, мьютексы, события, семафоры, ожидающий таймер и т.д. Теоретические основы механизмов синхронизации потоков и процессов подробно изложены в [4, 15]. Здесь будет рассматриваться только механизм критических секций. С остальными способами синхронизации процессов и потоков операционной системы Windows можно ознакомиться в [4, 14].

Таблица 6.5.1

Наименование функции	Назначение
`DeleteCriticalSection`	Разрушить критическую секцию
`EnterCriticalSection`	Войти в критическую секцию
`InitializeCriticalSection`	Инициализировать критическую секцию
`LeaveCriticalSection`	Покинуть критическую секцию
`TryEnterCriticalSection`	Пытаться войти в критическую секцию

Критические секции, является одним из самых простых механизмов синхронизации и в нашей модели могут быть использованы для исключения совместного использования списка ListContact потоками AcceptServer и GarbageCleaner. Критическая секция является объектом операционной системы типа CRITICAL_SECTION. Для работы с этим объектом используются функции, которые перечислены в таблице 6.5.1.

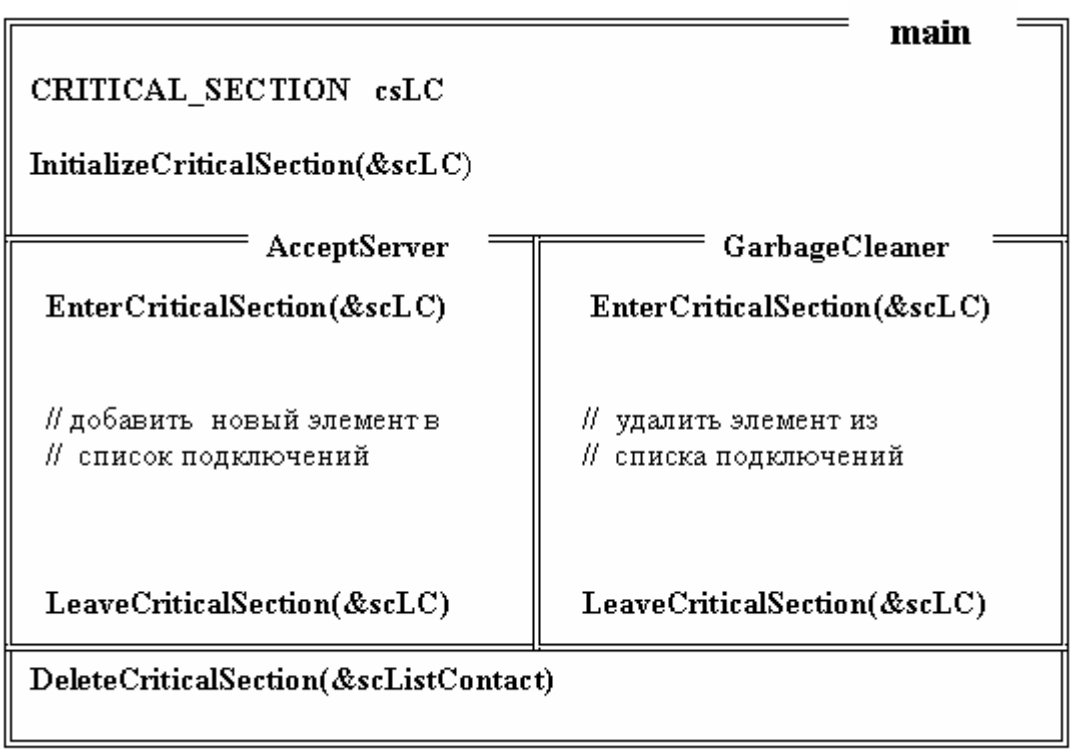

Рисунок 6.5.2. Схема применения механизма критических секций

Схема использования механизма критических секций изображена на рисунке 6.5.2.

```
// -- инициализировать критическую  секцию

// Назначение: функция  предназначена для  инициализации
//             критической секции
 VOID InitializeCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                                )

// -- войти в критическую  секцию
// Назначение: функция  предназначена для входа в критическую
//             секцию
 VOID EnterCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                               )
// Примечание: в том случае если в критической секции не
//       находится ни один из потоков, функция завершает
//       свою работу, пропуская поток внутрь критической
//       секции (занимая секцию); если же секция занята
//       другим потоком, то функция приостанавливает
//       выполнение потока до того момента освобождения секции

//-- покинуть критическую  секцию
// Назначение: функция  предназначена для выхода из
//             критической  секции

VOID LeaveCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                              )

//-- пытаться войти в критическую  секцию
// Назначение: функция  предназначена для условного входа в
//             критическую  секцию и

 BOOL TryEnterCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                              )
// Код возврата: функция возвращает  ненулевое значение в
//         в том случае если секция не занята или поток
//         уже находится в критической секции; иначе
//         возвращается значение NULL

//-- разрушить критическую секцию
// Назначение: функция  предназначена для разрушения
//             критической секции
   VOID DeleteCriticalSection (
       LPCRITICAL_SECTION pCS, // указатель на критическую секцию
                               )
```

Рисунок 6.5.3. Функции реализующие механизм критических секций

На рисунке изображены 6.5.2 два параллельно работающих потока AcceptServer и GarbageCleaner. При подключении очередного клиента в рамках потока AcceptServer в список подключений добавляется элемент, содержащий информацию об этом подключении. Поток GarbageCleaner просматривает последовательно элементы списка подключений и удаляет неиспользуемые элементы. Для того, чтобы добавление и удаление элементов списка не осуществлялось одновременно, эти операции помещают внутри критической секции соответствующего потока. Каждая критическая секция начинается функцией EnterCriticalSection, а заканчивается функцией LeaveCriticalSection. Следует обратить внимание, что эти функции используют в качестве параметра общий объект синхронизации.

6.6. Асинхронный вызов процедур

Может возникнуть ситуация, когда одному из потоков параллельного сервера потребуется выполнить в рамках другого или нескольких других потоков процедуру. Причем старт процедуры должен быть согласованным с исполняющим потоком. Для решения такой задачи может быть применен механизм асинхронного вызова процедур.

Асинхронной процедурой называется функция, которая выполняется асинхронно в контексте какого-нибудь потока. Для исполнения асинхронной процедуры необходимо определить асинхронную процедуру, указать поток, в контексте которого она будет выполняться, и дать разрешение на выполнение асинхронной процедуры.

Если перейти к описанной выше модели сервера, то можно предложить следующий пример использования асинхронных процедур. При подключении очередного клиента, функция AcceptServer запускает обслуживающий поток EchoServer. С этого момента AcceptServer теряет связь с потоком EchoServer и, в принципе, даже не знает о моменте окончания его работы. Будем предполагать, что в начале и при окончании работы потока EchoServer, поток AcceptServer должен выдавать на свою консоль сообщение о завершении работы обслуженного клиента. В этом случае поток EchoServer может поставить функцию (асинхронную процедуру) в специальную очередь к потоку AcceptServer. Момент выполнения асинхронной процедуры определяется внутри функции потока AcceptServer.

Основные функции необходимые для асинхронного вызова процедур перечислены в таблице 6.6.1.

Таблица 6.6.1

Наименование функции	Назначение
`QueueUserAPC`	Поставить асинхронную процедуру в очередь
`SleepEx`	Приостановит поток для выполнения асинхронных процедур

На рисунке 6.6.1 изображена схема использования механизма асинхронного вызова процедур для параллельного сервера ConcurrentServer.

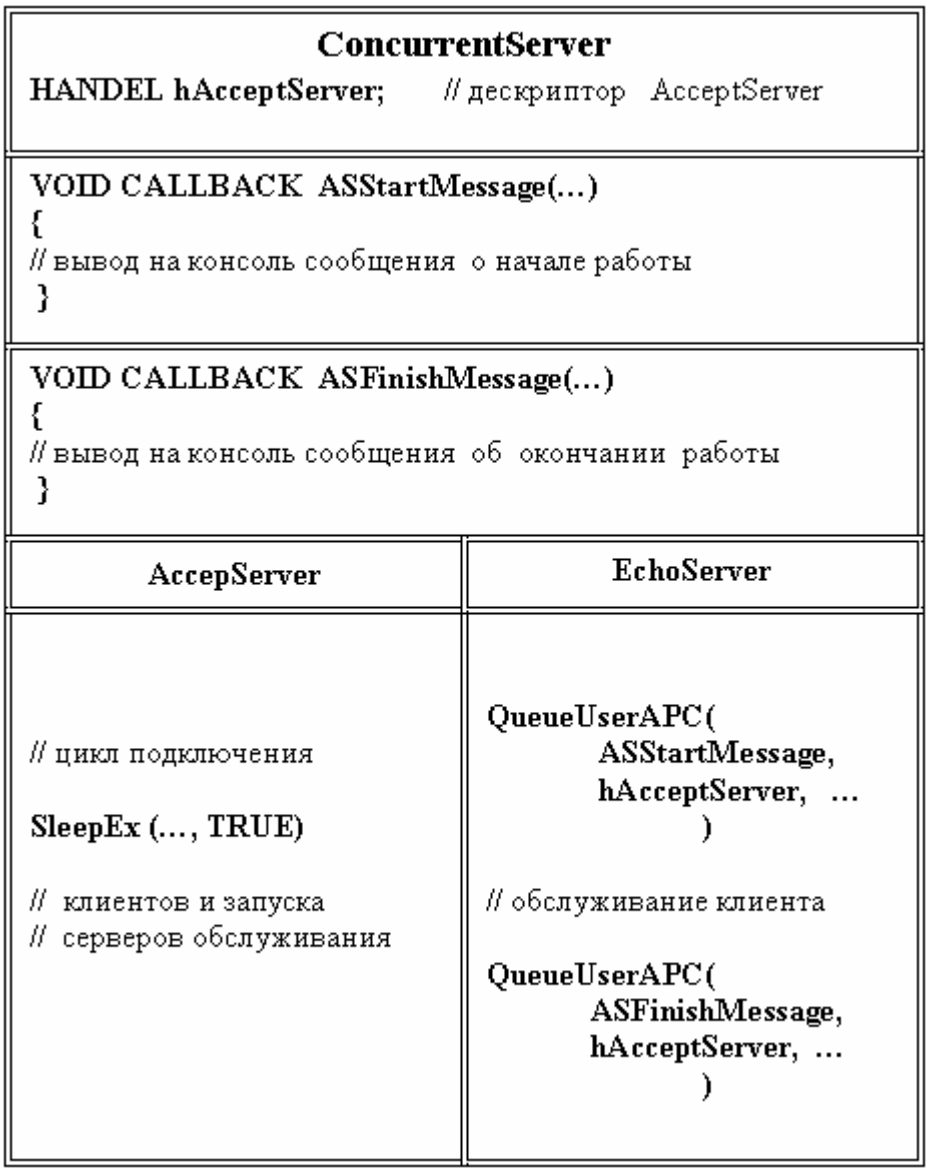

Рисунок 6.6.1. Схема использования механизма асинхронного вызова процедур

На схеме изображен параллельный сервер ConcurrentServer, в рамках которого определены две асинхронные процедуры ASStartMessage и ASFinishMessage, а также два параллельно работающих потока AcceptServer и EchoSever.

В начале своей работы функция EchoServer выполняет функцию QueueUserAPC (рисунок 6.6.2), которая помещает асинхронную процедуру ASStartMessage в очередь к потоку AcceptServer. Эта очередь обеспечивается операционной системой Windows и работает по алгоритму FIFO (First Input First Output). После отключения программы клиента, перед самым завершением своей работы, функция EchoServer вновь исполняет функцию

QueueUserAPC, но уже для постановки в очередь асинхронной процедуры ASFinishMessage.

Выполнение всех асинхронных процедур, находящихся к этому моменту в очереди потока, осуществляется последовательно в потоке AcceptServer, после того, как в рамках этого потока будет выполнена функция SleepEx (рисунок 6.6.3).

```
// -- поставить асинхронную процедуру в очередь
// Назначение: функция  предназначена для постановки  в
//             FIFO-очередь к потоку асинхронную  процедуру

 DWORD QueueUserAPC(
          PAPCFUNC  fn,   // [in] имя функции асинхронной процедуры
          HANDLE    hT,   // [in] дескриптор исполняющего потока
          DWORD     pm    // [in] передаваемый параметр
                   )
// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение; иначе возвращается
//       значение NULL
// Примечание:  в случае неудачного выполнения устанавливается
//       системный код возврата, который может быть получен
//       с помощью функции GetLastError
```

Рисунок 6.6.2. Функция QueueUserAPC

```
// -- приостановить поток для выполнения асинхронных процедур
// Назначение: функция позволяет приостановить поток для
//             ожидания и  последовательного выполнения в
//             контексте данного потока асинхронных процедур,
//             находящихся к этому моменту в очереди

 DWORD SleepEx(
             DWORD  ms,   // [in] интервал времени в миллисекундах
             BOOL   rg    // [in] режим
              )
// Код возврата: в случае истечения заданного интервала
//       времени  функция возвращает значение NULL, иначе
//       возвращает ненулевое значение
// Примечание:- если  для параметра rg установлено значение
//        TRUE, то при наличии асинхронных процедур в
//        очереди потока они начинают немедленно выполняться;
//        если же асинхронных процедур в очереди нет, то
//        ожидается их появление заданный в параметре ms
//        интервал времени;
//        если для параметра rg установлено значение FALSE,
//        то поток просто приостанавливается на заданный
//        параметром ms интервал времени
```

Рисунок 6.6.3. Функция SleepEx

Асинхронные процедуры не могут возвращать никакого значения и принимают только один параметр. Прототип асинхронной процедуры изображен на схеме использования механизма асинхронных процедур (рисунок 6.6.1).

6.7. Использование ожидающего таймера

Очевидно, что производительность параллельного сервера в значительной степени зависит от количества одновременно подключившихся клиентов: с ростом подключившихся клиентов, производительность убывает. Обслуживание каждого клиента связано с выделением ему определенных ресурсов: процессорного времени, оперативной памяти, сетевого трафика и т.п. В связи с ограниченностью ресурсов возникает необходимость управлять процессом обслуживания.

Одной из задач управления сервером является выявление слишком продолжительных подключений. Подключения, которые удерживаются клиентом сверх разумного времени, могут возникнуть по самым разным причинам: особенности алгоритма, зацикливание или зависание программы клиента и т.п. Очевидным решением проблемы является введение ограничения на продолжительность соединения. Для реализации такого решения может быть использован механизм ожидающего таймера.

Опишем еще один очевидный случай, когда может быть востребован ожидающий таймер. Речь идет о поддержке некоторого расписания работ в рамках сервера. Например, предполагается, что некоторые услуги сервера поддерживаются только в определенные интервалы времени суток, или существует поток (или процесс) периодически запускаемый по определенному расписанию, для выполнения внутренних функций самого сервера (например, для вывода собранной статистики в предназначенный для этого файл).

Ожидающим таймером в Windows, называется объект синхронизации, который переходит в ***сигнальное состояние*** при наступлении заданного момента времени. Если ожидающий таймер ждет момента перехода в сигнальное состояние, то говорят, что он находится в ***активном состоянии***. Другое состояние ожидающего таймера ***пассивное*** – из этого состояния он не может перейти в сигнальное состояние.

По способу перехода из сигнального стояния в несигнальное, ожидающие таймеры разделяются на ***таймеры с ручным сбросом*** и таймеры с ***автоматическим сбросом***, иначе называемые ***таймерами синхронизации***.

По способу перехода из несигнального состояния в сигнальное, ожидающие таймеры бывают ***периодические*** и ***непериодические***. Периодические таймеры работают по циклу: активное состояние – сигнальное состояние – активное состояние. Непериодические таймеры могут только один раз перейти из активного состояния в сигнальное.

Основные функции, необходимые для работы с ожидающим таймером перечислены в таблице 6.7.1.

Таблица 6.7.1

Наименование функции	Назначение
CancelWaitableTimer	Отменить ожидающий таймер
CreateWaitableTimer	Создать ожидающий таймер
OpenWaitableTimer	Открыть существующий ожидающий таймер
SetWaitableTimer	Установить ожидающий таймер
WaitForSingleObject	Ждать сигнального состояния ожидающего таймера

```
// -- создать ожидающий таймер
// Назначение: функция предназначена для создания дескриптора
//             ожидающего таймера и установки его статических
//             параметров

 HANDLE CreateWaitableTimer(
       LPSECURITY_ATTRIBUTES sattr, // [in] атрибуты безопасности
       BOOL                  reset, // [in] тип сброса таймера
       LPCTSTR               tname  // [in] имя таймера
                           );
// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор вновь созданного или уже
//       существующего с таким именем ожидающий таймер; в
//       последнем случае функция GetLastError  вернет
//       значение ERROR_ALREADY_EXISTS
// Примечание:- если  для параметра sattr установлено значение
//        NULL, то атрибуты безопасности устанавливаются по
//        умолчанию  и дескриптор не наследуется дочерними
//        процессами;
//        - если  значение параметра reset равно TRUE, то  это
//        таймер с ручным сбросом, если FALSE – то это таймер
//        синхронизации;
//        - если  значение параметра tname равно NULL, то
//        создается безымянный таймер, иначе этот параметр
//        указывает на строку с именем ожидающего таймера.
```

Рисунок 6.7.1. Функция CreateWaitableTimer

Создание ожидающего таймера и установка его статических параметров осуществляется функцией CreateWaitableTimer (рисунок 6.7.1).

Для перевода ожидающего таймера в активное состояние и его установки динамических параметров предназначена функция SetWaitableTimer (рисунок 6.7.2).

```
// -- установить ожидающий таймер
// Назначение: функция предназначена для перевода таймера в
//             в активное состояние и установки его параметров

    BOOL SetWaitableTimer(
    HANDLE                hWTimer, //[in] дескриптор таймера
    const LARGE_INTEGER   DueTime, // [in] время срабатывания
    LONG                  lPeriod, // [in] период времени
    PTIMERAPCROUTINE      apcFunc, // [in] процедура завершения
    LPVOID                prmFunc, // [in] параметр процедуры
    BOOL                  ofPower  // [in] управление питанием
                          );

// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение, иначе возвращается
//       значение FALSE
// Примечание:-  параметр DueTimer  содержит адрес структуры
//        типа LARGE_INTEGER (целое число размером  64 бита),
//        если это число положительное, то его значение
//        указывает абсолютное время перехода таймера в
//        сигнальное состояние; если число отрицательное,
//        считается, что задан интервал времени от текущего
//        системного времени; время задается в  единицах
//        равных 100 наносекунд (10^(-7)сек).
//        - значение lPeriod  определяет, является ли таймер
//        периодическим: если его значение равно нулю, то
//        таймер  не периодический; если значение больше нуля
//        то таймер периодический и lPeriod задает период в
//        миллисекундах;
//        - параметр apcFunc должен указывать на функцию
//        завершения, которая устанавливается в очередь
//        асинхронных  процедур после перехода  таймера в
//        сигнальное состояние; если  для параметра apcFunc
//        установлено значение NULL, то функция завершения
//        не используется;
//        - параметр prmFunc может содержать единственный
//        аргумент, который может быть передан функции
//        завершения;
//        - параметр ofPower устанавливает режим управления
//        питанием: если установлено значение TRUE, то после
//        после перехода таймера в сигнальное состояние
//        компьютер переключается в режим экономии
//        электроэнергии; если установлено FLASE, то
//        переключения не происходит
```

Рисунок 6.7.2. Функция SetWaitableTimer

Если создан поименованный таймер, то он может быть использован в контексте другого процесса с помощью функции OpenWaitableTimer (рисунок 6.7.3). Для перевода таймера в неактивное состояние применяется

функция CancelWaitableTimer (рисунок 6.7.4). Для ожидания сигнального состояния таймера используется универсальная функция WaitForSingleObject (рисунок 6.7.5), о которой уже упоминалось выше, в связи с ожиданием завершения потока.

```
// -- открыть существующий ожидающий таймер
// Назначение: функция предназначена для создания дескриптора
//             существующего ожидающего таймера

    HANDLE OpenWaitableTimer(
          DWORD          raccs,  //[in] режимы доступа
          BOOL           rinht,  //[in] режим наследования
          LPCTSTR        tname   //[in] имя таймера
                          );

// Код возврата: в случае успешного завершения функция
//       возвращает дескриптор ожидающего таймера, иначе
//       возвращается значение NULL
// Примечание:-  параметр raccs  может принимать любую
//       комбинацию следующих флагов:
//       TIMER_ALL_ACCESS – произвольный доступ к таймеру,
//       TIMER_MODIFY_STATE – можно только изменять состояние,
//       SYNCHRONIZE – можно использовать только в функциях
//       ожидания;
//       - если параметр rinht установлен в значение FALSE, то
//       дескриптор таймера не наследуется дочерними
//       процессами, если установлено значение TRUE, то
//       осуществляется наследование.
```

Рисунок 6.7.3. Функция OpenWaitableTimer

```
// -- отменить ожидающий таймер
// Назначение: функция предназначена для перевода таймера
//             в пассивное состояние

 BOOL CancelWaitableTimer(
               HANDLE  hTimer  //[in] дескриптор ожидающего таймера
                          );

// Код возврата: в случае успешного завершения функция
//       возвращает ненулевое значение, иначе возвращается
//       значение NULL
```

Рисунок 6.7.4. Функция CancelWaitableTimer

Следует отметить, что для ожидания сигнального состояния ожидающего таймера, можно использовать и другие функции. Например, если используется несколько ожидающих таймеров (или других объектов

синхронизации), то применяется функция WaitForMultipleeObjects [4, 14]. В параметрах этой функции можно указать массив дескрипторов объектов синхронизации и различные режимы ожидания и проверки сигнального состояния. Если кроме того, требуется исполнять асинхронные процедуры, то можно использовать функции WaitForSingleObjectEx и WaitForMultipleeObjectsEx.

```
// -- отменить ожидающий таймер
// Назначение: функция предназначена для перевода таймера
//             в пассивное состояние

 DWORD WaitForSingleObject(
             HANDLE  hTimer //[in] дескриптор ожидающего таймера
             DWORD   mstout //[in] временной интервал в миллисекундах
                          );

// Код возврата: в случае успешного завершения функция
//     возвращает одно из следующих значений:
//     WAIT_OBJECT_0 – таймер в сигнальном состоянии;
//     WAIT_TIME_OUT – истек интервал времени, таймер не
//     в сигнальном состоянии;
// Примечание: если значение параметра mstout равно нулю, то
//     не осуществляется приостановка потока, а только
//     осуществляется проверка сигнального состояния таймера;
//     если значение mstout больше нуля, то осуществляется
//     приостановка потока пока не будет исчерпан заданный
//     интервал времени или таймер не перейдет в сигнальное
//     состояние; если значение mstout  равно INFINITE, то
//     сигнальное состояние ожидается бесконечно долго.
```

Рисунок 6.7.5. Функция WaitForSingleObject

С помощью функции SetWaitableTimer может быть установлена функция завершения, которая устанавливается в очередь асинхронных процедур после перехода ожидающего таймера в сигнальное состояние. Функция завершения должна иметь такой же прототип, как и у асинхронных процедур. Прототип и механизм вызова асинхронных процедур уже рассматривался выше

Если перейти к рассмотрению модели ConcurrentServer, то, по всей видимости, ожидающий таймер будет создаваться потоком AcceptServer для каждого запущенного потока EchoSever. Сразу же после перехода одного из ожидающих таймеров, в очередь асинхронных процедур потока AcceptServer будет поставлена соответствующая процедура завершения.

На рисунке 6.7.6 изображена схема использования ожидающего таймера в модели параллельного сервера ConcurrentServer. Следует обратить внимание на новое применение асинхронной процедуры

ASFinishMessage, назначение которой рассматривалось выше. Теперь, в рамках этой асинхронной функции отменяется ожидающий таймер.

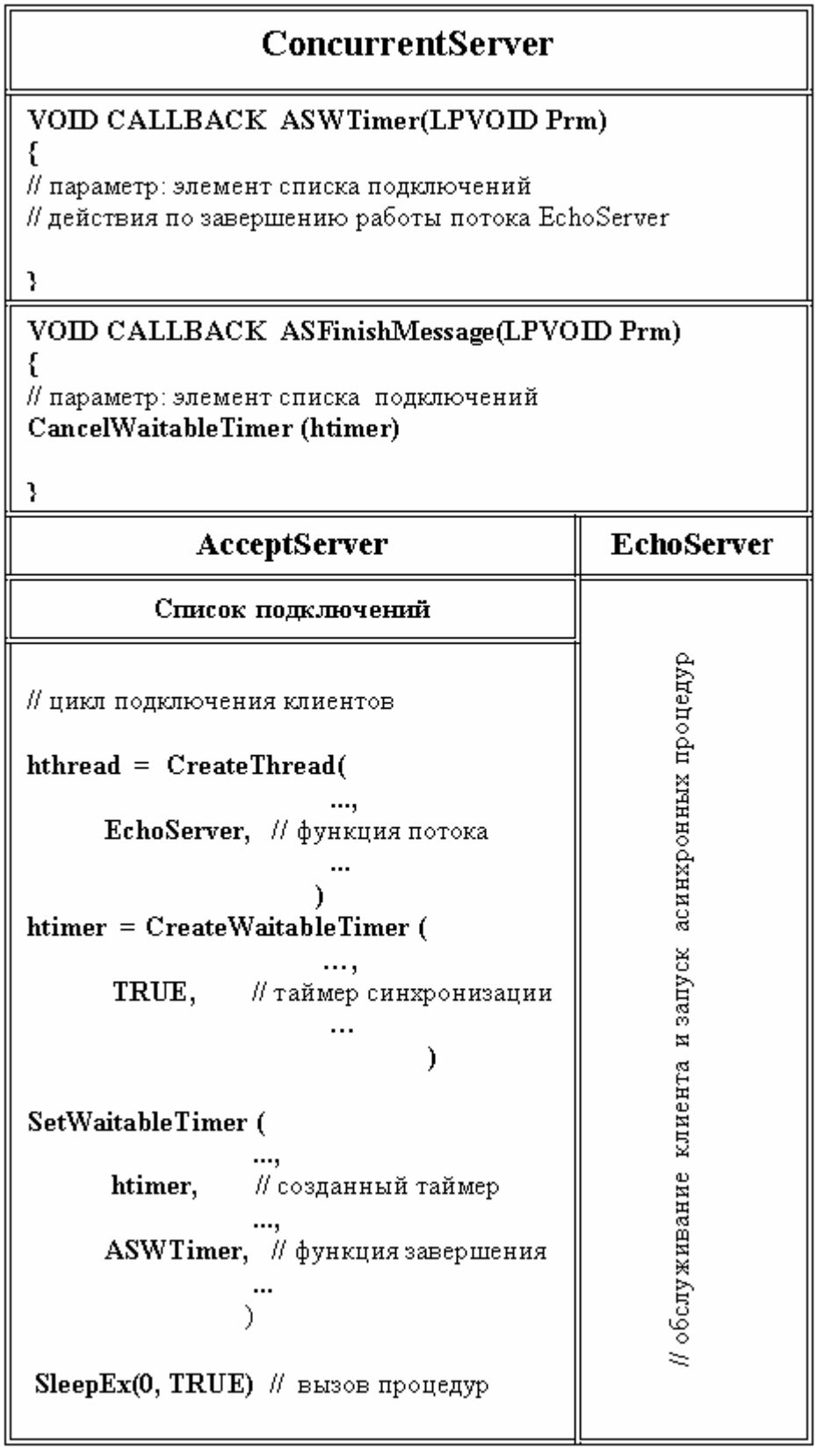

Рисунок 6.7.6. Схема использования ожидающего таймера в модели ConcurrentServer

На рисунке 6.7.6 обозначен список подключений. Структура элементов этого списка рассматривалась выше. Основное предназначение списка – это хранение информации о каждом подключении. В модели сервера ConcurrentServer предполагается хранить дескрипторы обрабатывающего потока и ожидающего таймера в элементах списка подключений. Так как асинхронные процедуры в модели всегда связаны с конкретным подключением, то удобно передавать в качестве параметра этим процедурам соответствующий элемент списка подключений.

В заключении следует отметить, что как и все дескрипторы (HANDEL) операционной системы Windows, неиспользуемые дескрипторы ожидающего таймера должны закрываться с помощью функции CloseHandle.

6.8. Применение атомарных операций

Иногда параллельным потокам необходимо выполнять некоторые несложные действия над общими переменными, исключая совместный доступ к этим переменным. Если в этом случае использовать критические секции или другие механизмы синхронизации, то может оказаться, что затраты на синхронизацию потоков значительно превысят затраты на выполнение самих операций. В таких случаях применяют блокирующие функции. Блокирующие функции выполняют несколько элементарных операций, которые объединяются в одну неделимую операцию, называемую ***атомарной операцией***.

В таблице 6.8.1 приведены четыре блокирующие функции, используемые в операционных системах семейства Windows.

Таблица 6.8.1

Наименование Функции	Назначение
`InterlockedCompareExchange`	Сравнить и заменить значение
`InterlockedDecrement`	Уменьшить значение на единицу
`InterlockedExchange`	Заменить значение
`InterlockedExchangeAdd`	Изменить значение
`InterlockedIncrement`	Увеличить значение на единицу

Все перечисленные функции требуют, чтобы адреса переменных были выровнены на границу слова, т.е. были кратны четырем. Для такого выравнивания достаточно, чтобы переменная была объявлена в программе со спецификаторами типов long, unsigned long или LONG, ULONG, DWORD.

Функция InterlockedExchange (рисунок 6.8.1) позволят установить (заменить) значение переменной. Если требуется заменить конкретное известное значение, то можно использовать функцию InterlockedCompareExchange (рисунок 6.8.2). Функции InterlockedIncrement и InterlockedDecrement (рисунки 6.8.3 и 6.8.4) используются для увеличения

или уменьшения значения переменной на единицу. Функция InterlockedExchangeAdd (рисунок 6.8.5) позволяет увеличить или уменьшить значение переменной на заданную величину.

```
// -- заменить значение
// Назначение: функция предназначена выполнения атомарной
//             операции замены переменной

 LONG InterlockedExchange(
             LPLONG  pt,    // [in] адрес заменяемой переменной
              LONG    vl     // [in] новое значение переменной
                         );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.1. Функция InterlockedExchange

```
// -- сравнить и заменить значение
// Назначение: функция предназначена выполнения атомарной
//             операции сравнения и замены переменной в случае
//             удачного сравнения

 PVOID InterlockedCompareExchange(
                 PVOID   pt,    // [in] адрес заменяемой переменной
                 PVOID   vl     // [in] новое значение переменной
                 PVOID   cp     // [in] значение для сравнения
                                   );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.2. Функция InterlockedCompareExchange

```
// -- увеличить значение на единицу
// Назначение: функция предназначена выполнения атомарной
//             операции инкремента

 LONG InterlockedIncrement(
                 LPLONG  pt,    // [in] адрес изменяемой переменной
                          );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.3. Функция InterlockedIncrement

```
// -- уменьшить значение на единицу
// Назначение: функция предназначена выполнения атомарной
//             операции декремента

 LONG InterlockedDecrement(
                 LPLONG  pt,    // [in] адрес изменяемой переменной
                          );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.4. Функция InterlockedDecrement

```
// -- изменить значение
// Назначение: функция предназначена выполнения атомарной
//             операции изменения значения переменной на
//             заданную величину

 LONG InterlockedExchangeAdd(
                 LPLONG  pt,    // [in] адрес изменяемой переменной
                 LONG    vl      // [in] прибавляемое значение
                             );

// Код возврата: функция возвращает старое значение переменной
```

Рисунок 6.8.4. Функция InterlockedExchangeAdd

В модели параллельного сервера ConcurrentServer атомарные операции могут быть использованы, например, для управления количеством одновременно обслуживаемых клиентов.

Предположим, что число одновременно обслуживаемых клиентов хранится в переменной с именем ClientServiceNumber, а максимальное количество одновременно работающих клиентов в глобальной переменной MaxClientServiceNumber. Тогда удобно в асинхронных процедурах ASStartMessage и ASFinishMessage выполнять увеличение и уменьшение значения переменной ClientServiceNumber. Перед очередным подключением клиента функция AcceptServer сравнивает значение переменных ClientServiceNumber и MaxClientServiceNumber и, если число подключенных клиентов достигло максимального значения, оказывает клиенту в подключении. При такой схеме работы нет параллельных потоков одновременно изменяющих переменную ClientServiceNumber, т.к. функции AcceptServer, ASStartMessage, ASFinishMessage работают в одном потоке последовательно. Более того, можно даже предусмотреть команду консоли ConsolePipe управления сервером, позволяющую уменьшать или увеличивать значение MaxClientServiceNumber, что позволяет регулировать нагрузку на весь сервер.

Предположим теперь, что в фоновом режиме работает еще один поток с именем AdvisorServer, который определенным образом оценивает нагрузку на параллельный сервер и в необходимые моменты уменьшает или увеличивает значение MaxClientServiceNumber. В этом случае параллельная работа с консолью уже не возможна, т.к. возможно несогласованное изменение переменной MaxClientServiceNumber.

Проблема возникнет и в том случае, если еще усложнить работу сервера и добавить еще один поток AcceptServer (предположим прослушивающего другой порт) или разрешить подключение более одной консоли (увеличив количество экземпляров именованного канала). В этом случае значения переменных ClientServiceNumer и MaxClientServiceNumer станут непредсказуемыми.

6.9. Не блокирующий режим работы сокета

В описанной выше модели параллельного сервера ConcurrentServer на функцию потока AcceptServer возложена обязанность подключения клиентских приложений, запуска обслуживающего потока (и функции) EchoServer, а также исполнение некоторых команд управления сервером. Остановимся подробнее на процессе обработки команд управления.

Для подключения клиентских программ AcceptServer использует функцию Winsock2 accept, которая приостанавливает работу потока AcceptServer до момента подключения клиента. После того, как клиентская программа выполнит функцию connect, функция accept сервера сформирует новый сокет для обмена данными между сервером и клиентом, а также возобновит исполнение потока AcceptServer.

Для ввода команд сервера (в том числе и исполняемых потоком AcceptServer) в модели ConcurrentServer предназначается функция потока ConsolePipe, которая поддерживает именованный канал с клиентом консоли RConsole.

Предположим, что с консоли управления введена команда stop, предназначенная для приостановки сервером подключения новых клиентских программ до выдачи команды start, которая должна возобновить возможность подключения клиентов. Если в момент ввода команды stop, поток AcceptServer был приостановлен функцией accept, то действие этой команды наступит только после подключения следующего клиента, т.к. нет возможности проверить наличие, введенной и переданной функцией ConsolePipe команды. Введенная следующая команда может “затереть” предыдущую и команда stop не выполнена вообще.

Возникшую проблему можно решить несколькими способами. Например, можно выполнить команду connect из функции ConsolePipe и этим самым заставить AcceptServer сделать цикл и проверить наличие команды управления. В этом случае будет необходим механизм, позволяющий различать подключение от ConsolePipe и других клиентов. Можно было бы реализовать подключение консоли не через именованный канал, а также как клиентов – через сокет. В последнем случае, также как и в

предыдущем случае требуется различать подключение консоли и остальных клиентов.

Рассмотрим еще один способ решения проблемы управления процессом подключения клиентов в потоке AcceptServer. Этот способ основан на специальном режиме работы сокета.

Алгоритм работы функции accept, который рассматривался выше, был обусловлен ***режимом блокировки (blocking mode)***, установленным (по умолчанию) для сокета. Переключение сокета в ***режим без блокировки (nonblocking mode)***, позволяет избежать приостановки программы. В режиме без блокировки выполнение accept, не приостанавливает выполнение потока, как это было прежде, а возвращает значение нового сокета, если обнаружен запрос на создание канала (функция connect, выполненная клиентом), или значение INVALID_SOCKET, если запроса на создание канала нет в очереди запросов или возникла ошибка. Для того, чтобы отличить ошибку от отсутствия запроса, используется уже рассмотренная выше функция WSAGetLastError, которая в последнем случае возвращает значение WSAEWOULDBLOCK.

Для переключения режимов сокета применяется функция ioctlsocket, входящая в состав Winsock2 (рисунок 6.9.1).

```
// -- переключить режим работы сокета
// Назначение: функция предназначена для управления режимом
//             ввода/вывода сокета

 int ioctlsocket(
            SOCKET  sock, // [in] дескриптор сокета
            long    scmd, // [in] команда, применяемая к сокету
             u_long*  pprm  // [in,out] указатель на параметр команды
                );
// Код возврата: в случае успешного выполнения функция
//        возвращает нулевое значение, иначе возвращается
//        значение SOCKET_ERROR
// Примечание: параметр scmd может принимать следующие
//        значения: FIONBIO, FIONREAD, SIOCATMARK; команда
//        FIONBIO применяется для переключения режимов:
//        blocking/nonblocking; для установки режима
//        nonblocking значение параметра  *pprm должно быть
//        отличным от нуля.
```

Рисунок 6.9.1. Функция ioctlsocket

На рисунке 6.9.2 представлен фрагмент функции потока AcceptServer в котором используется функция ioctlsocket для переключения сокета в режим без блокировки. После успешного выполнения функции ioctlsocket вызывается функция CommandsCycle, фрагменты которой приведены на рисунке 6.9.3. Предполагается, что команды консоли поток AcceptServer выбирает из области памяти, адрес которой передается в качестве параметра

функции AcceptServer. Тип TalkersCommand, к которому преобразуется параметр функции потока, является перечислением (тип enum) команд применяемых для управления сервером.

```
DWORD WINAPI AcceptServer(LPVOID  pPrm)  //  сервер ожидания запроса
{
//.............................................................
try
 {
  //... WSAStartup(...),sS = socket(...),bind(sS, ...),listen(sS,...),..

  u_long nonblk;
  if (ioctlsocket( sS, FIONBIO, &(nonblk = 1)) == SOCKET_ERROR)
          throw SetErrorMsgText("ioctlsocket:",WSAGetLastError());
  CommandsCycle(*((TalkersCommand*)pPrm));

  //...closesocket(sS),WSACleanup()...................................
 }
 //.............................................................
 ExitThread(*(DWORD*)pPrm);                 // завершение потока
 }
```

Рисунок 6.9.2. Пример применения функции ioctlsocket

```
void CommandsCycle(TalkersCommand& cmd)      // цикл обработки команд
{
int  squirt = 0;
while(cmd  != EXIT)     // цикл обработки команд консоли и подключений
{
 switch (cmd)
 {
 //.....................................................
 case START: cmd = GETCOMMAND; // возобновить подключение клиентов
             squirt =  AS_SQUIRT; //#define AS_SQUIRT 10
             break;
 case STOP:  cmd = GETCOMMAND; // остановить подключение клиентов
             squirt = 0;
             break;
 //.....................................................
 };
 if (AcceptCycle(squirt))   //цикл  запрос/подключение (accept)
   {
    cmd = GETCOMMAND;
    //.... запуск потока EchoServer.....................
    //.... установка ожидающего таймера для процесса EchoServer ...
   }
 else SleepEx(0, TRUE);    // выполнить асинхронные процедуры
};
};
```

Рисунок 6.9.3. Пример обработки команд stop, start и exit

Функция CommandCycle (рисунок 6.9.3) предназначена для обработки команд управления сервером и подключения клиентов. Подключение клиентов осуществляется функцией AcceptCycle (рисунок 6.9.4). Изображенный фрагмент функции CommandCycle содержит цикл обработки команд stop (остановить подключение клиентов), start (возобновить подключение клиентов) и exit (завершение работы потока AcceptServer), а также функцию AcceptCylcle, предназначенную для подключения клиентов. Если команда принята функцией на обработку, то устанавливается новое значение команды getcommand, обозначающее, что функция AcceptCycle готова к приему новой команды управления.

В случае успешного подключения клиента функция AcceptCylcle возвращает значение true. Значение squirt, передаваемое в качестве параметра функции AcceptCylcle является максимальным количеством итераций выполнения функции accept (в режиме без блокировки) для подключения клиента за один вызов функции AcceptCylcle. Если клиент подключился, то для него запускается поток EchoServer, устанавливается ожидающий таймер, проверяется и увеличивается счетчик подключений и т.п. Если клиент не подключился, то осуществляется запуск асинхронных процедур с помощью функции SleepEx.

```
bool AcceptCycle(int squirt)
{
bool rc = false;
Contact c(Contact::ACCEPT,"EchoServer");

while(squirt-- > 0 && rc == false)
{
if ((c.s = accept(sS,
                  (sockaddr*)&c.prms, &c.lprms)) == INVALID_SOCKET)
 {
  if (WSAGetLastError() != WSAEWOULDBLOCK)
          throw  SetErrorMsgText("accept:",WSAGetLastError());
 }
else
 {
  rc = true;                // подключился
  EnterCriticalSection(&scListContact);
  contacts.push_front(c);
  LeaveCriticalSection(&scListContact);
  }
}
return rc;
};
```

Рисунок 6.9.4. Пример применения функции accept в режиме без блокировки сокета

После каждого вызова функции accept для сокета в режиме без блокировки (рисунок 6.9.4), следует проверять код возврата на равенство

значению WSAEWOULDBLOK. Это значение возвращается в том случае, если очередь подключений пуста. В функции AcceptCycle организован цикл проверки очереди подключений повторяющийся squirt раз.

6.10. Использование приоритетов

Производительность параллельного сервера в значительной степени зависит от правильного распределения ресурсов между конкурирующими потоками. Важнейшим ресурсом при этом является процессорное время.

По принципу обслуживания потоков операционная система Windows относится к классу систем с вытесняющей многозадачностью [15]. Каждому потоку операционной системы периодически выделяется квант процессорного времени. Величина кванта, выделяемая потоку, зависит от типа операционной системы Windows, типа процессора и приблизительно равна двадцати миллисекундам. По истечении кванта времени исполнение текущего потока прерывается, контекст потока сохраняется и процессор передается потоку с высшим приоритетом.

Приоритеты потоков в Windows определяются относительно приоритета процесса, в рамках которого они исполняются. Приоритеты потоков изменяются от 0 (низший приоритет) до 31 (высший приоритет). Основные функции необходимые для работы с приоритетами приведены в таблице 6.10.1.

Таблица 6.10.1

Наименование функции	Назначение
`GetPriorityClass`	Получить приоритет процесса
`GetThreadPriority`	Получить приоритет потока
`GetProcessPriorityBoost`	Определить состояние режима процесса
`GetThreadPriorityBoost`	Определить состояние режима потока
`SetPriorityClass`	Изменить приоритет процесса
`SetThreadPriority`	Изменить приоритет потока
`SetProcessPriorityBoost`	Установить или отменить динамический режим потоков процесса
`SetThreadPriorityBoost`	Установить или отменить динамический режим потока

Приоритеты процессов устанавливаются при их создании функцией CreateProcess [14]. Операционная система Windows различает четыре типа процессов в соответствии с их приоритетами: фоновые процессы, процессы с нормальным приоритетом, процессы с высоким приоритетом и процессы реального времени.

Фоновые процессы выполняют свою работу, когда нет активных пользовательских процессов. Обычно эти процессы следят за состоянием системы.

Процессы с нормальным приоритетом – это обычные пользовательские процессы. Это приоритет присваивается пользовательским процессам по умолчанию. В рамках этого класса допускается небольшое понижение или повышение приоритетов процесса.

```
// -- получить приоритет процесса
// Назначение: функция предназначена для определения
//             приоритетного класса процесса

 DWORD GetPriorityClass(
                   HANDLE hP    // [in] дескриптор процесса
                        );
// Код возврата: в случае  успешного  выполнения   функция
//    возвращает  одно из следующих значений, обозначающих
//    приоритетный класс процесса:
//    IDLE_PRIORITY_CLASS – фоновый процесс;
//    BELOW_NORMAL_PRIORITY_CLASS – ниже нормального;
//    NORMAL_PRIORITY_CLASS – нормальный процесс;
//    ABOVE_NORMAL_PRIORITY_CLASS – выше нормального;
//    HIGH_NORMAL_PRIORITY_CLASS – высокоприоритетный процесс;
//    REAL_NORMAL_PRIORITY_CLASS – процесс реального времени;
//    иначе возвращается значение NULL.
```

Рисунок 6.10.1. Функция GetPriorityClass

```
// -- изменить приоритет процесса
// Назначение: функция предназначена для изменения приоритета
//             процесса

 BOOL SetPriorityClass(
                  HANDLE hP,    // [in] дескриптор процесса
                  DWORD  py    // [in] новый приоритет процесса
                        );
// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое  значение,  иначе  возвращается
//    значение FALSE.
// Примечание: параметр py может принимать одно из следующих
//    значений:
//    IDLE_PRIORITY_CLASS – фоновый процесс;
//    BELOW_NORMAL_PRIORITY_CLASS – ниже нормального;
//    NORMAL_PRIORITY_CLASS – нормальный процесс;
//    ABOVE_NORMAL_PRIORITY_CLASS – выше нормального;
//    HIGH_NORMAL_PRIORITY_CLASS – высокоприоритетный процесс;
//    REAL_NORMAL_PRIORITY_CLASS – процесс реального времени.
```

Рисунок 6.10.2. Функция SetPriorityClass

Процессы с высоким приоритетом это тоже пользовательские процессы, от которых требуется более быстрая реакция на некоторые события. Обычно такие приоритеты имеют программные системы, работающие на платформе операционной системы Windows.

Процессы реального времени обычно работают непосредственно с аппаратурой компьютера и продолжительность их работы ограниченно отведенным временем реакции на внешние события.

Узнать приоритет процесса можно с помощью функции GetPriorityClass (рисунок 6.10.1), а изменить с помощью функции SetPriorityClass (рисунок 6.10.2). Эти функции используют в качестве параметра дескриптор или псевдодескриптор процесса. Псевдодескриптор текущего процесса может быть получен с помощью функции GetCurrentProcess (рисунок 6.10.3).

```
// -- получить псевдодескриптор процесса
// Назначение: функция предназначена для получения
//             псевдодескриптора текущего процесса

 HANDLE GetCurrentProcess(VOID);

// Код возврата: в случае успешного выполнения функция
//    возвращает псевдодескриптор текущего процесса, иначе
//    возвращается значение FALSE.
```

Рисунок 6.10.3. Функция GetCurrentProcess

При диспетчеризации процессов и потоков квант времени выделяется потоку. Приоритет потока, который учитывается операционной системой при выделении процессорного времени, называется ***базовым*** или ***основным приоритетом потока***. Всего существует 32 базовых приоритетов – от 0 до 31. Для каждого базового приоритета существует очередь потоков. При диспетчеризации квант процессорного времени выделяется потоку, который стоит первым в очереди с наивысшим приоритетом. Базовый приоритет потока определятся исходя из приоритета процесса и уровня приоритета потока. Уровень приоритета потока может быть: низший, ниже нормального, нормальный, выше нормального, высший приоритет, приоритет фонового потока и приоритет потока реального времени. Значения базовых приоритетов потоков можно определить, воспользовавшись таблицей 6.10.2.

При создании потока его базовый приоритет устанавливается как сумма приоритета процесса, в контексте которого этот поток выполняется, и значения THREAD_PRIORITY_NORMAL (0), соответствующего нормальному уровню приоритета потока. Значение уровня приоритета потока может быть изменено с помощью функции SetThreadPriority

(рисунок 6.10.4), а узнать текущий уровень приоритета потока – с помощью функции GetThreadPriority (рисунок 6.10.5).

```
// -- изменить приоритет потока
// Назначение: функция предназначена для изменения приоритета
//             потока

 BOOL SetThreadPriority(
                     HANDLE hT,    // [in] дескриптор потока
                     DWORD  py     // [in] новый приоритет потока
                       );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое  значение,  иначе  возвращается
//    значение FALSE.
// Примечание: параметр py может принимать одно из следующих
//    значений:
//    THREAD_PRIORITY_LOWEST – низший приоритет;
//    THREAD_PRIORITY_BELOW_NORMAL – ниже среднего;
//    THREAD_PRIORITY_NORMAL – нормальный;
//    THREAD_PRIORITY_ABOVE_NORMAL – выше нормального;
//    THREAD_PRIORITY_HIGHEST – высший приоритет;
//    THREAD_PRIORITY_IDLE – фоновый поток;
//    THREAD_PRIORITY_TIME_CRITICAL – поток реального времени.
```

Рисунок 6.10.4. Функция SetThreadPriority

```
// -- получить приоритет потока
// Назначение: функция предназначена для получения приоритета
//             потока

 DWORD GetThreadPriority(
               HANDLE hT,    // [in] дескриптор потока
                       );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает одно из следующих значений:
//    THREAD_PRIORITY_LOWEST – низший приоритет;
//    THREAD_PRIORITY_BELOW_NORMAL – ниже среднего;
//    THREAD_PRIORITY_NORMAL – нормальный;
//    THREAD_PRIORITY_ABOVE_NORMAL – выше нормального;
//    THREAD_PRIORITY_HIGHEST – высший приоритет;
//    THREAD_PRIORITY_IDLE – фоновый поток;
//    THREAD_PRIORITY_TIME_CRITICAL – поток реального времени;
//    иначе возвращается   значение FALSE.
```

Рисунок 6.10.5. Функция GetThreadPriority

Таблица 6.10.2

Приоритет потока	Приоритет процесса					
	реального времени	высокий	выше норм.	нормальный.	ниже норм.	фоновый.
реального времени	31	15	15	15	15	15
высший	26	15	12	10	8	6
выше норм.	25	14	11	9	7	5
нормальный	24	13	10	8	6	4
ниже норм.	23	12	9	7	5	3
низший	22	11	8	6	4	2
фоновый	16	1	1	1	1	1

Если базовый приоритет потока находится в пределах от 0 до 15, то он может изменяться динамически операционной системой. При получении потоком сообщения Windows или при переходе его в состояние готовности система повышает его приоритет на 2. В процессе выполнения базовый приоритет такого потока понижается на единицу после каждого отработанного кванта, но не ниже базового значения.

```
// -- установить или отменить динамический режим всех потоков
//    процессов
// Назначение: функция предназначена для установки или отмены
//             динамического изменения базового приоритета
//             всех потоков процесса

 BOOL SetProcessPriorityBoost(
                        HANDLE hP, // [in] дескриптор процесса
                        BOOL   de  // [in] состояние режима
                               );

// Код возврата: в случае успешного выполнения функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потоков процесса запрещается; если значение de
//    установлено в FALSE – режим разрешается.
```

Рисунок 6.10.6. Функция SetProcessPriorityBoost

Для управления режимом динамического изменения базового приоритета потока могут быть использованы функции SetProcessPriorityBoost и SetThreadPriorityBoost (рисунки 6.10.5 и 6.10.6). С помощью функции SetProcessPriorityBost осуществляется отмена или возобновление режима динамического изменения базового приоритета всех

потоков, работающих в контексте общего процесса. Для изменения режима только для одного потока применяется функция SetThreadPriorityBoost.

```
// -- установить или отменить динамический режим потока
// Назначение: функция предназначена для установки или отмены
//             динамического изменения базового приоритета
//             потока

 BOOL SetThreadPriorityBoost(
                        HANDLE hP,  // [in] дескриптор потока
                        BOOL   de   // [in] состояние режима
                             );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потока запрещается; если значение de    установлено
//    в FALSE – режим разрешается.
```

Рисунок 6.10.7. Функция SetThreadPriorityBoost

Определить состояние режима динамического изменения приоритетов потока можно с помощью функций GetProcessPriorityBoost (рисунок 6.10.8) и GetThreadPriorityBoost (рисунок 6.10.9).

```
// -- определить состояние  режима процесса
// Назначение: функция предназначена для определения состояния
//             режима динамического изменения базового
//             приоритета   всех потоков процесса

 BOOL GetProcessPriorityBoost(
                        HANDLE hP, // [in] дескриптор процесса
                        PBOOL  de  // [out] состояние режима
                              );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потоков процесса запрещен; если значение de
//    установлено в FALSE – режим разрешен.
```

Рисунок 6.10.8. Функция GetProcessPriorityBoost

```
// -- определить состояние  режима потока
// Назначение: функция предназначена для определения состояния
//             режима динамического изменения базового
//             приоритета  потока

 BOOL GetThreadPriorityBoost(
                        HANDLE hT, // [in] дескриптор потока
                        PBOOL  de  // [out] состояние режима
                              );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе возвращается FALSE.
// Примечание: если значение параметра de установлено в TRUE,
//    то режим динамического изменения базового приоритета
//    для потока запрещен; если значение de  установлено
//    в FALSE – режим разрешен.
```

Рисунок 6.10.9. Функция GetThreadPriorityBoost

В модели параллельного сервера ConcurrentServer основная конкуренция за процессорное время происходит между потоком AcceptServer, обрабатывающими потоками (потоки EchoServer), и потоками GarbageCleaner и ConsolePipe. Принцип распределения приоритетов между этими потоками, зависит от стратегии, преследуемую разработчиком. Очевидным является только то, что с пониженным приоритетом (в фоновом режиме) должен работать поток GarbageCleaner, который моделирует внутренний процесс сервера по сборке мусора. С повышением приоритета потока AcceptServer, по-видимому, более активным станет подключение клиентов, с повышением приоритетов обрабатывающих потоков клиенты будут быстрее обслуживаться и отключаться от сервера.

Можно говорить, о системе управлении приоритетами, которая бы периодически оценивала статистику работы сервера и самостоятельно перестраивала распределение приоритетов. Например, если в очередь подключений растет, то может быть следует увеличить приоритет потока AcceptServer, или, наоборот, если подключения редки, но работает много обслуживающих потоков, то следует повысить приоритет последних и понизить приоритет всех остальных. И, наконец, можно просто назначать приоритеты с консоли управления

6.11. Обработка запросов клиента

До сих пор предполагалось, что параллельный сервер исполняет однотипные запросы клиентов. При подключении клиента, запускался определенный (известный) поток, который обслуживал данное соединение. В реальности часто бывает, что заранее не известно, какого рода услуга будет запрошена клиентом у сервера.

Обычно сервер должен распознавать некоторое количество команд (запросов), которые клиент может направить в его адрес. Каждая команда идентифицируется своим кодом и может содержать определенный набор операндов. Например, часто первая команда, которую обрабатывает сервер, является команда **connect (**или, что-то подобное**),** в которой указывается запрашиваемый клиентом сервис и другие дополнительные параметры.

Поступающая на вход сервера команда должна пройти лексический, синтаксический и (если необходимо) семантический анализ, и только после этого может быть исполнена сервером. С методами построения лексических, синтаксических и семантических анализаторов можно ознакомиться в [16, 17]. Здесь эти вопросы рассматриваться не будут.

Рассмотрим принципы построения параллельного сервера, обрабатывающего несколько различных запросов клиента на примере, уже рассмотренной выше, модели ConcurrentServer.

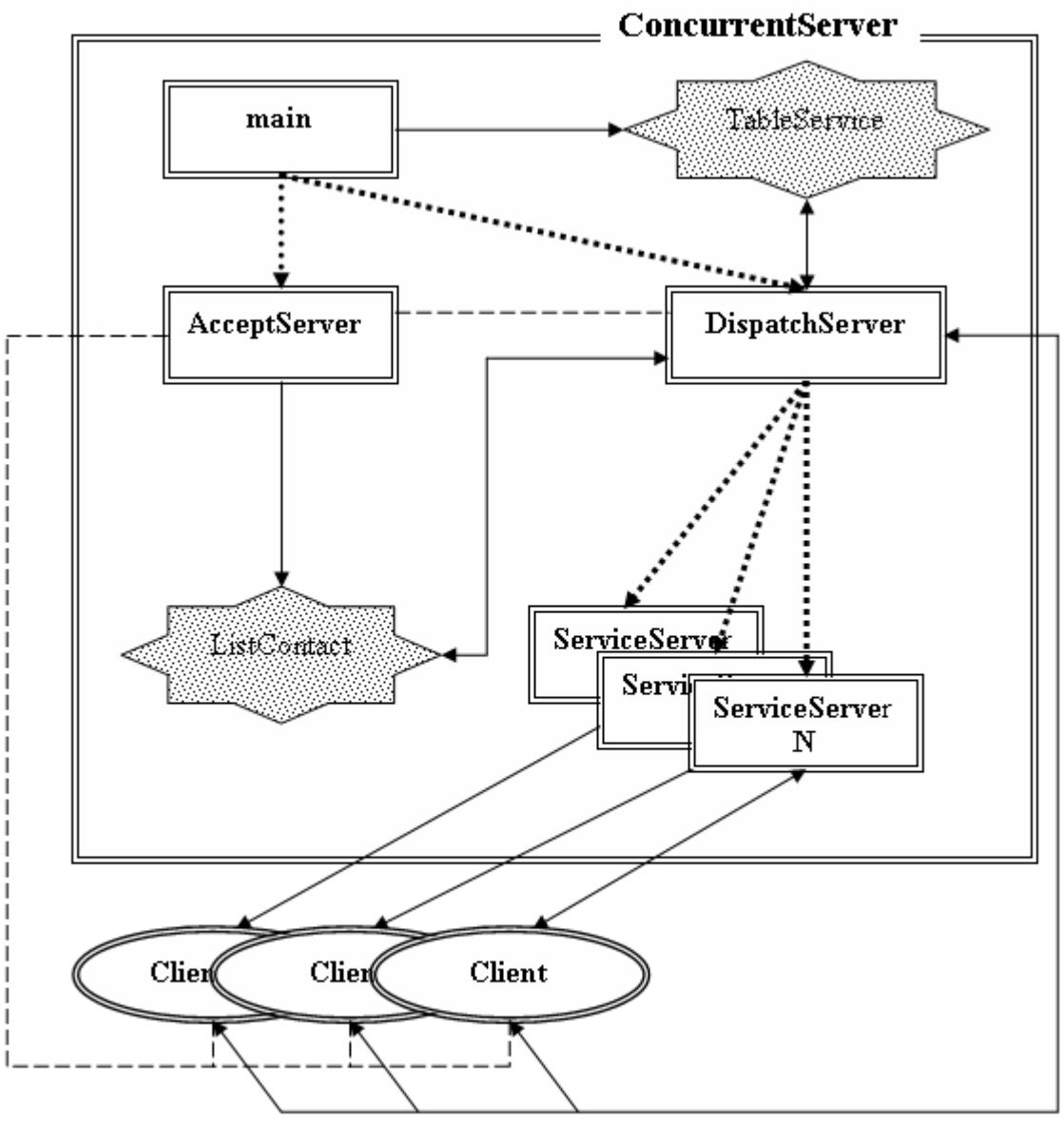

Рисунок 6.11.1. Структура параллельного сервера с диспетчером запросов

На рисунке 6.11.1 изображена структура параллельного сервера ConcurrentServer, обслуживающего разнотипные запросы. Для простоты здесь не изображены некоторые потоки, которые уже рассматривались ранее. Как и прежде сплошными направленными линиями изображается движение информации, пунктирными запуск потока, а штриховой − процедуры подключения и синхронизации.

Основным отличием новой структуры, является промежуточный поток DispatchServer между AcceptServer и обслуживающими потоками ServiceServer (раньше это был единственный поток называемый EchoServer). Теперь предполагается, что сначала программа клиента осуществляет процедуру подключения (для этого используется поток AcceptServer), потом поток DispatchServer принимает от клиента запрос (команду) на обслуживание и после этого уже запускается соответствующий поток ServiceServer, который исполняет команду и в случае необходимости обменивается данными с клиентом.

Введение промежуточного звена, обуславливается тем, что после этапа подключения, клиент (в общем случае) может достаточно долго не запрашивать у сервера услугу (не выполнять функцию send, пересылающую команду сервера). Ожидать поступление команды в потоке AcceptServer не целесообразно, т.к. его основное назначение − подключение клиентов.

На поток DispatchServer возлагается прием первой команды от клиента после подключения, содержащей идентификатор, запрашиваемого сервиса. Соответствие идентификатора и адреса потоковой функции содержится в специальной таблице TableService, который формируется при инициализации сервера (например, на основе конфигурационного файла).

Поток DispatchServer получает информацию (сокет и его параметры) о новом подключении через список ListContact (элементы списка создаются и заполняются в потоке AcceptServer). Поиск в списке ListContact нового подключения DispatchServer осуществляет после того, как поток AcceptServer сигнализирует потоку DispatchServer (на рисунке сигнал обозначен штриховой линией). Сигнал о наличии нового подключения можно выполнить с помощью, уже рассмотренного выше, механизма асинхронных процедур или с помощью механизма событий, который будет рассматриваться ниже. В любом случае поток DispatchServer должен отследить интервал времени (с помощью ожидающего таймера) от момента подключения до получения первой команды, проанализировать команду на правильность, а также запустить поток соответствующий запрошенному сервису (и) или оправить диагностирующее сообщение.

По всей видимости, увеличение функциональности сервера, потребует некоторого переосмысления процесса управления его работой. Очевидным является необходимость динамически запрещать или разрешать поддержку определенных запросов, классифицировать запросы по приоритетности, устанавливать различные интервалы.

6.12. Применение механизма событий

Выше уже упоминался механизм событий, позволяющий оповестить поток о некотором выполненном действии, произошедшем за пределами потока. Саму ***задачу оповещения*** часто называют ***задачей условной синхронизации***.

В операционных системах семейства Windows события описываются объектами ядра ***Events***. Различают два типа событий: с ручным сбросом и с автоматическим сбросом. Различие между этими типами заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только с помощью функции ResetEvent, а событие с автоматическим сбросом переходит в несигнальное состояние как с помощью функции ResetEvent, так и при помощи функции ожидания.

Функции необходимые для работы с событиями приведены в таблице 6.12.1.

Наименование функции	Назначение
`CreateEvent`	Создать событие
`OpenEvent`	Открыть событие
`PulseEvent`	Освободить ожидающие потоки
`ResetEvent`	Перевести событие в несигнальное состояние
`SetEvent`	Перевести событие в сигнальное состояние

Для создания события используется функция CreateEvent (рисунок 6.12.1). С помощью функции OpenEvent (рисунок 6.12.2) можно получить дескриптор уже созданного события и установить некоторые его характеристики. С помощью функции SetEvent (рисунок 6.12.3) любое событие может быть переведено в сигнальное состояние. Для перевода любого события в несигнальное состояние применяется функция ResetEvent (рисунок 6.12.4). Следует отметить, что если событие с автоматическим сбросом, ждут несколько потоков, то переходе события в сигнальное состояние освобождается только один поток.

Функция PulseEvent (рисунок 6.12.5) используется для событий с ручным сбросом. Если одно и тоже событие ожидается несколькими потоками, то выполнение функции PulseEvent приводит к тому, что все эти потоки выводятся из состояния ожидания, а само событие сразу же переходит в несигнальное состояние.

В таблице не приводиться, уже рассмотренная выше, универсальная функция WaitForSingleObject, которая используется для перевода потока в состояние ожидания до получения сигнала.

Следует обратить внимание, что при создании события можно указать состояние (сигнальное или не сигнальное) в котором находится данное событие. Кроме того, задав имя события, можно обеспечить к нему доступ из другого процесса.

```
// -- создать событие
// Назначение: функция предназначена для создания  события и
//             установки его параметров

 HANDLE  CreateEvent(
      LPSECURITY_ATTRIBUTES sattr, // [in] атрибуты безопасности
      BOOL                  etype, // [in] тип события
      BOOL                  state, // [in] начальное состояние
      LPCTSTR               ename  // [in] имя события
                  );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает дескриптор события  иначе  NULL.
// Примечания:- если значение параметра sattr  установлено в
//    NULL, то значения атрибутов безопасности будут
//    установлены по умолчанию;
//    - если значение параметра etype установлено в TRUE, то
//    создается событие с ручным сбросом, иначе с -
//    автоматическим;
//    - если значение параметра state установлено в TRUE, то
//    начальное состояние события является сигнальным, иначе
//    состояние не сигнальное;
//    - параметр ename задает имя события, что позволяет
//    работать разным процессам работать с общим событием;
//    если не предполагается использовать имя события, то для
//    этого параметра может быть установлено значение NULL.
```

Рисунок 6.12.1. Функция CreateEvent

```
// -- открыть событие
// Назначение: функция предназначена для создания дескриптора
//             уже существующего поименованного события

 HANDLE  OpenEvent(
        DWORD   accss, // [in] флаги доступа
        BOOL    rinrt, // [in] режим наследования, TRUE - разрешено
        LPCTSTR ename  // [in] имя события
                  );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает дескриптор события  иначе  NULL.
// Примечания:- параметра accss  может принимать любую
//   логическую комбинацию следующих флагов:
//   EVENT_ALL_ACCESS – полный доступ к событию;
//   EVENT_MODIFY_STATE – допускается изменение состояния;
//   SYNHRONIZE – можно использовать в функциях ожидания.
```

Рисунок 6.12.2. Функция OpenEvent

```
// -- перевести событие в сигнальное состояние
// Назначение: функция предназначена перевода существующего
//             события  в сигнальное состояние

 BOOL  SetEvent(
             HANDLE   hE  // [in] дескриптор события
               );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе  NULL.
```

Рисунок 6.12.3. Функция SetEvent

```
// -- перевести событие в  несигнальное состояние
// Назначение: функция предназначена перевода существующего
//             события  в несигнальное состояние

 BOOL  ResetEvent(
             HANDLE   hE  // [in] дескриптор события
               );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе  NULL.
```

Рисунок 6.12.4. Функция ResetEvent

```
// -- освободить ожидающие потоки
// Назначение: функция предназначена вывода всех ожидающих
//             сигнала потоков  из состояния ожидания и
//             перевода события в несигнальное состояние

 BOOL  PulseEvent(
             HANDLE   hE  // [in] дескриптор события
                 );

// Код возврата: в случае  успешного  выполнения   функция
//    возвращает ненулевое значение, иначе  NULL.
// Примечание: функция используется только для событий с
//    ручным сбросом
```

Рисунок 6.12.5. Функция PulseEvent

6.13. Использование динамически подключаемых библиотек

При разработке параллельного сервера часто бывает полезным разделить процедуры управления сервером и обслуживания клиентов. Имеется в виду, что поддерживаемый сервером сервис может меняться при неизменной логике управления сервером. Одним из способов такого

разделения является размещения функций обслуживающих потоков в динамически подключаемой библиотеке (dll-библиотеке).

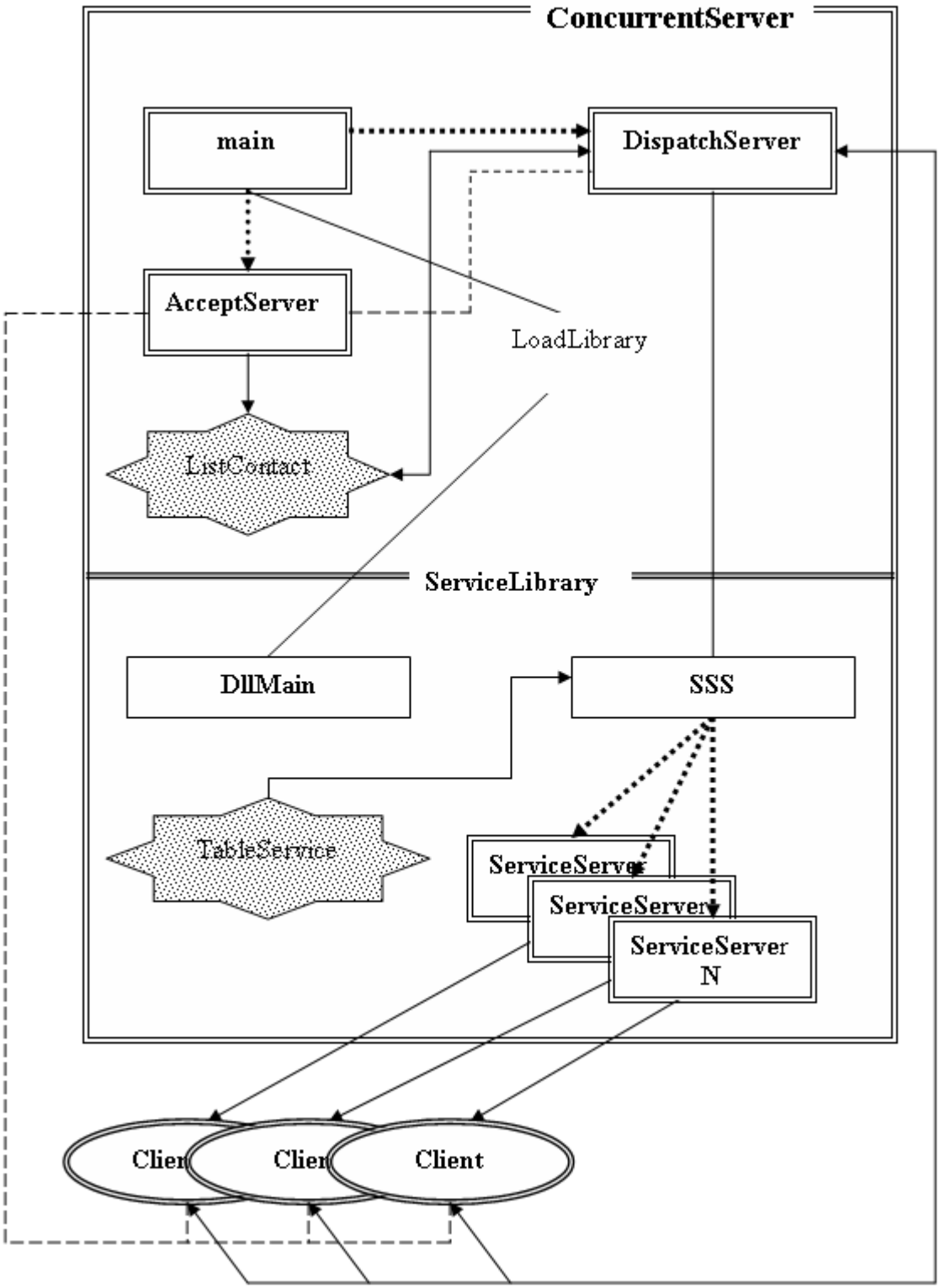

Рисунок 6.13.1. Структура параллельного сервера с динамической библиотекой

Рассмотренный выше обслуживающий разнотипные запросы сервер использовал таблицу TableService для того, чтобы сопоставить запросу

клиента функцию обслуживающего потока. Удобно поместить эту таблицу вместе с функциями обслуживающих потоков в dll-библиотеке. В этом случае сервер может динамически загружать dll-библиотеку.

Таким образом, теперь параллельный сервер состоит из двух частей: управляющий сервер и dll-библиотека функций потоков обслуживания. При такой структуре, набор сервисных услуг предоставляемых сервером зависит от версии динамической библиотеки. Местонахождение библиотеки может быть одним из параметров инициализации сервера.

На рисунке 6.13.1 изображена структура параллельного сервера, использующего динамическую библиотеку ServiceLibrary, для хранения таблицы TableService и функций обслуживающих потоков. Библиотека загружается в память при инициализации сервера. В состав библиотеки входит стандартная функция DllMain (рисунок 6.13.2), экспортируемая функция SSS для запуска потока по заданному клиентом коду команды, а также функции обслуживающих потоков.

```
// -- главная функция динамически подключаемой библиотеки
// Назначение: функция обозначает точку входа в программный
//             модуль динамически подключаемой библиотеки,
//             функции получает управление от операционной
//             системы в момент загрузки.

 BOOL  WINAPI DllMain(
             HINSTANCE hinst,    //[in] дескриптор dll
             DWORD     rcall,    //[in] причина вызова
             LPVOID    wresv     // резерв Windows
                    );

// Код возврата: в случае  успешного  завершения  функция
//    должна вернуть значение  TRUE, иначе  FALSE.
// Примечание: - в параметре hinst операционная система
//      передает дескриптор, который фактически равен
//      виртуальному адреса загруженной dll;
//    - параметр rcall может принимать одно из следующих
//      значений:
//      DLL_PROCESS_ATTACH – dll загружена в адресное
//      пространство процесса;
//      DLL_THREAD_ATTACH – dll вызывается в контексте
//      потока, созданного в рамках процесса;
//      DLL_THREAD_DETACH – завершился поток в контексте
//      которого загружена dll;
//      DLL_PROCESS_DETACH – dll – выгружается из
//      адресного пространства процесса;
//    - если параметр rcall имеет значение DLL_PROCESS_ATTACH,
//      а параметр wresv равен NULL, то библиотека загружена
//      динамически, другое значение wresv, говорит о
//      статической загрузке библиотеки.
```

Рисунок 6.13.2. Функция DllMain

Динамические библиотеки представляют собой программный модуль, который может быть загружен в виртуальную память процесса как статически, во время создания исполняемого модуля процесса, так и динамически во время исполнения процесса операционной системой. Дальнейшее изложение, в основном, будет касаться динамически загружаемых библиотек. Принципы использования статически загружаемых библиотек изложены в [12].

Для создания dll-библиотеки в среде Visual Studio необходимо выбрать проект типа Win32 Dynamic-Link Library. Как и любая программа на языке C++, динамически подключаемая библиотека имеет главную функцию, которая отмечает точку входа программы при ее исполнении операционной системой. Главная функция dll-библиотеки называется DllMain – ее шаблон автоматически создается Visual Studio.

Функции необходимые для работы с динамически подключаемыми библиотеками приведены таблице 6.13.1. Следует отметить в таблице 6.13.1 содержатся не все функции Win32 API, применяемые для работы с библиотеками. С полным перечнем функций можно ознакомиться в [4, 12, 14].

Таблица 6.13.1

Наименование функции	**Назначение**
`FreeLibrary`	Отключить dll-библиотеку от процесса
`GetProcAddress`	Импортировать функцию
`LoadLibrary`	Загрузить dll-библиотеку

Функция LoadLibrary (рисунок 6.13.3) применяется для загрузки динамической библиотеки в память компьютера. Для выгрузки библиотеки используется функция FreeLibrary (рисунок 6.13.4). Следует отметить, что если при загрузке библиотеки обнаруживается, что она уже загружена (даже другим процессом), то повторной загрузки не осуществляется. В то же время следует помнить, что для dll-библиотек используется механизм проецирования dll-библиотек, предоставляющий каждому ***клиенту библиотеки*** (так принято называть приложения использующие функции dll-библиотеки) полную независимость. Для создания общей памяти для нескольких процессов в рамках одной dll-библиотеки приходится предпринимать дополнительные усилия [12]. Выгрузка (точнее освобождение ресурсов) dll-библиотеки осуществляется после того, как последний процесс, использовавший библиотеку выполнит функцию FreeLibrary. О функциях dll-библиотеки которые предназначены для вызова из вне говорят, что они ***экспортируются*** библиотекой. Когда рассматривают эти же функции со стороны клиента dll-библиотеки, то говорят об ***импорте функций*** из dll-библиотеки. Для импорта функций dll-библиотеки применяется функция GetProcAddress (рисунок 6.13.4). Следует отметить, что импортировать можно не только функции, но некоторые переменные.

Импорт переменных рассмотрен в [4], а дальнейшее изложение касается только импорта функций.

```
// -- загрузить dll-библиотеку
// Назначение: если dll-библиотека еще не находится в памяти
//             компьютера,  то осуществляется загрузка
//             библиотеки, настройка адресов и проецирование
//             dll-библиотеки в адресное пространство
//             процесса; если же  dll-библиотека к моменту
//             вызова функции уже загружена, то происходит
//             только проецирование; в любом случае управление
//             получит функция DllMain.

 HMODULE LoadLibrary(
             LPCTSTR  fname,    //[in]   имя файла dll-библиотеки
                    );

// Код возврата: в случае  успешного  завершения  функция
//     возвращает дескриптор загруженного модуля, иначе NULL
// Примечание: при поиске файла используется следующая
//      последовательность:
//      1) каталог, из которого запущено приложение;
//      2) текущий каталог;
//      3) системный каталог Windows;
//      4) каталоги, указанные в переменной окружения PATH.
```

Рисунок 6.13.3. Функция LoadLibrary

```
// -- отключить dll-библиотеку от процесса
// Назначение: функция предназначена для освобождения ресурсов
//             процесса, занимаемых dll-библиотекой; если
//             нет больше процессов, которые используют
//             библиотеку, то осуществляется ее выгрузка
 BOOL FreeLibrary(
            HMODULE hDll,  //[in]   дескриптор dll-библиотеки
                  );

// Код возврата: в случае  успешного  завершения  функция
//     возвращает ненулевое значение, иначе NULL
```

Рисунок 6.13.4. Функция FreeLibrary

Экспортируемая dll-библиотекой функция должна быть специальным образом оформлены, а именно иметь прототип изображенный на рисунке 6.13.6. Модификатор extern “C” используется для указания компилятору на то, что эта функция имеет имя в стиле языка C. Квалификатор __declspec(dllexport) применяется для обозначения экспортируемых dll-библиотекой функций.

```
// -- импортировать функцию
// Назначение: функция предназначена извлечения (импорта)
//             адреса функции dll-библиотеки

 FARPROC  GetProcAddress(
            HMODULE hDll,  //[in]   дескриптор dll-библиотеки
            LPCTSTR name   //[in]   имя импортируемой функции
                  );
// Код возврата: в случае  успешного  завершения  функция
//     возвращает адрес функции, иначе NULL
```

Рисунок 6.13.5. Функция GetProcAddress

```
// -- прототип экспортируемой dll-библиотекой функции

extern "C" __declspec(dllexport) Funcname (.....);
```

Рисунок 6.13.6. Прототип экспортируемой функции

```
#include "stdafx.h"
#include "Windows.h"
#include "DefineTableService.h" // макро для TableService
#include "PrototypeService.h"  // прототипы  обслуживающих потоков

BEGIN_TABLESERVICE              // таблица
  ENTRYSERVICE("Echo", EchoServer),
  ENTRYSERVICE("Time", TimeServer),
  ENTRYSERVICE("0001", ServiceServer01)
END_TABLESERVICE;

extern "C" __declspec(dllexport) HANDLE  SSS (char* id, LPVOID prm)
{
 HANDLE rc = NULL;
 int  i = 0;
 while(i < SIZETS && strcmp(TABLESERVICE_ID(i), id) != 0)i++;
 if (i < SIZETS)
      rc =  CreateThread(NULL,NULL,
                         TABLESERVICE_FN(i), prm ,NULL,NULL);
 return rc;
};

BOOL APIENTRY DllMain( HANDLE hinst, DWORD  rcall, LPVOID wres)
{
 return TRUE;
}
```

Рисунок 6.13.7. Пример построения TableService и функции SSS

На рисунке 16.13.7 приводится текст программы простейшего dll-модуля. Главная функция DllMain имеет только одно назначение – она

возвращает значение TRUE. За время использования сервером dll-библиотеки функция DllMain вызываться несколько раз: при загрузке и освобождении библиотеки, при запуске и остановке потоковой функции. Таблица TableService создается с помощью достаточно простых макроопределений хранящихся в заголовочном файле DefineTableService.h (рисунок 6.13.8).

```
struct __TableEntry {
                     char __Id[9];
                     DWORD  ((WINAPI* __Fn)) (LPVOID);
                    };
#define BEGIN_TABLESERVICE   __TableEntry  __TableService[] = {
#define ENTRYSERVICE(s,t)  {s,t}
#define END_TABLESERVICE    };
#define TABLESERVICE_ID(i)   __TableService[i].__Id
#define TABLESERVICE_FN(i)   __TableService[i].__Fn
#define SIZETS sizeof(__TableService)/sizeof(__TableEntry)
```

Рисунок 6.13.8. Пример макроопределений для построения TableService

Заголовочный файл PrototypeService.h должен содержать все прототипы потоковых функций, которые определяются в таблице TableService.

Следует обратить внимание, что экспортируемой является только одна функция SSS, которая запускает обслуживающий поток по заданному коду команды. Пример импорта функции SSS программой клиента dll-библиотеки приводится в фрагменте программы на рисунке 6.13.9.

```
//.............................................................
HANDLE (*ts)(char*, LPVOID);
HMODULE st  = LoadLibrary("ServiceTable");
//.............................................................

ts = (HANDLE (*)(char*, LPVOID))GetProcAddress(st, "SSS");

//.............................................................

contacts.begin()->hthread = ts("Echo",(LPVOID)&(*contacts.begin()));

//.............................................................
//.............................................................

FreeLibrary(st);
//.............................................................
```

Рисунок 6.13.9. Пример импорта функции SSS

6.14. Принципы разработки системы безопасности сервера

Система безопасности сервера является достаточно объемным понятием, которое может включать организационные мероприятия, специализированное аппаратное обеспечение, специальное программное обеспечение и т.д. и т.п. Здесь будет говориться только о вопросах предотвращения несанкционированного доступа к ресурсам (услугам) сервера. Причем в той части, которая касается программной реализации сервера.

В зависимости от функциональной нагрузки сервера возможны различные подходы к системе безопасности. Нет смысла создавать систему безопасности для сервера, если несанкционированный доступ к его ресурсам в принципе не может принести ущерб. С другой стороны, для специализированного сервера, например, управляющего технологическим процессом, может потребоваться мощная система защиты.

В последних версиях операционной системы Windows, любой исполняемый процесс всегда выполняется от имени одного из пользователей операционной системы, имеющего учетную запись в базе данных менеджера учетных записей (SAM, Security Account Manager). Поэтому далее для простоты мы не будем различать понятия: клиент, пользователь (в смысле ученой записи) и процесс, запущенный от имени этого пользователя.

Клиенты сервера могут быть разбиты на две группы: группа администраторов и группа пользователей сервера.

Группа администраторов, включает в себя привилегированных клиентов сервера, которые могут выполнять команды управления сервером через консоль и (или) с помощью специального обслуживающего потока (в зависимости от реализации сервера). Как правило, администраторам доступны все ресурсы сервера. В общем случае, внутри группы администраторов возможна иерархия, которая зависит от сложности системы управления сервером.

Группа пользователей, включает в себя потребителей ресурсов сервера. В общем случае, внутри группы пользователей, тоже возможна иерархия, разграничивающая доступ пользователей к ресурсам сервера.

Система безопасности не может быть стационарной: в любой момент могут появиться новые пользователи, смениться администраторы, измениться перечень ресурсов и т.д. А это значит, что система безопасности должна быть настраиваемой.

Прежде, чем заниматься разработкой системы безопасности сервера, следует ответить на вопрос: будет ли это собственная система безопасности или же она будет опираться на систему безопасности операционной системы. Например, Microsoft SQL Server использует, как внутренний способ аутентификации пользователя, так аутентификацию Windows. Следует, однако, иметь в виду, что разработка собственной системы защиты от несанкционированного доступа не является простым делом. Затраты на разработку этой системы, могут оказаться сравнимыми с разработкой самого сервера.

Продуктивнее, с точки зрения автора, использовать систему безопасности операционной системы Windows. Действительно, в рамках системы безопасности Window уже поддерживается понятие пользователя и группы пользователей, есть готовые средства позволяющие поддерживать списки групп и пользователей, можно создавать охраняемые объекты, управлять доступом субъектов к объектам, контролировать привилегии т.д. и т.п. Кроме того, есть API позволяющий использовать все это.

В простейшем случае можно в рамках операционной системы Window создать две группы: группа администраторов и группа пользователей сервера. Заполнить эти группы именами пользователей, которым разрешается доступ к соответствующим ресурсам сервера. Пусть наименование этих групп являются параметрами конфигурации сервера. Будем также предполагать, что при подключении клиента к серверу, помимо наименования необходимого сервиса в строке connect содержится имя и пароль подключаемого клиента. Очевидно, что достаточно просто можно проверить, на принадлежность подключаемого клиента к одной из групп и сравнить установленные пароли.

API системы безопасности операционной системы Windows выходит за рамки данного пособия и достаточно подробно описан в [4].

6.15. Итоги главы

1. По принципу работы различают два типа серверов: итеративные серверы и параллельные серверы.
2. Итеративные серверы, как правило, используют для связи с клиентами протокол UDP и используются в тех случаях, когда предполагается, что запросы клиентов являются редкими, а исполнении их не требует много времени. В каждый момент времени итеративным сервером всегда обслуживается только один клиент.
3. В параллельных серверах с каждым клиентом устанавливается TCP-соединение. Одновременно к одному параллельному серверу может быть подключено несколько клиентов.
4. Для организации параллельной работы с несколькими соединениями, применяются механизмы потоков и (или) механизмы процессов. Механизм процессов является более затратным , но и более надежным, т.к. предполагает полное разделение параллельно работающих компонент сервера. С точки зрения программирования проще является, конечно, применение потоков, т.к. в этом случае потоки разделяют общую память процесса, что существенно упрощает разработку.
5. Наличие ресурсов, допускающих только последовательное использование, делает необходимым синхронизировать потоки и (или) процессы. В арсенале Windows имеется много различных механизмов синхронизации: критические секции, мьютексы,

семафоры, события, асинхронные процедуры, ожидающие таймеры и.д.

6. В тех случаях, когда требуется совместная работа нескольких потоков с одной общей переменной, может быть применен специальный механизм упрощенной синхронизации – атомарные операции.
7. Для подключения клиентов к параллельному серверу применяется не блокирующий режим работы сокета, позволяющий сделать этот процесс управляемым.
8. Одним из способов распределения ресурса процессорного времени между параллельными потоками (или процессами) сервера является назначение потокам (или процессам) приоритетов.
9. Параллельный сервер, обрабатывающий разнотипные запросы, должен распределять эти запросы с помощью специальной таблицы, связывающей код запроса и функцию обрабатывающего потока. Для того, чтобы сделать процедуры управления сервером независимыми от функций обслуживания, последние вместе с таблице кодов часто размещают в отдельной динамически подключаемой библиотеке.
10. При разработке системы безопасности сервера, целесообразно использовать систему безопасности операционной системы. Это значительно снизит затраты на ее разработку.