Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

В. В. Смелов

РАЗРАБОТКА РАСПРЕДЕЛЕННЫХ ПРИЛОЖЕНИЙ В СРЕДЕ WINDOWS

Рекомендовано
учебно-методическим объединением высших учебных заведений Республики Беларусь по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для студентов учреждений, обеспечивающих получение высшего образования по направлению специальности «Информационные системы и технологии (издательско-полиграфический комплекс)»

Минск 2008

УДК 004.4(075.8)
ББК 22.18я7
С 50

Р е ц е н з е н т ы:
доцент кафедры информационных технологий
автоматизированных систем Белорусского государственного
университета информатики и радиоэлектроники,
кандидат технических наук *О. В. Герман*;
кафедра управления информационными ресурсами
Академии управления при Президенте Республики Беларусь,
(доцент, кандидат технических наук *Г. Л. Тимонович*)

Все права на данное издание защищены. Воспроизведение всей книги или ее части не может быть осуществлено без разрешения учреждения образования «Белорусский государственный технологический университет».

Смелов, В. В.

С 50 Разработка распределенных приложений в среде Windows : учеб.-метод. пособие для студентов специальности «Информационные системы и технологии (издательско-полиграфи-ческий комплекс)». – Минск : БГТУ, 2008. – 206 с.

ISBN 978-985-434-764-6

Пособие посвящено разработке распределенных приложений архитектуры «клиент – сервер» в среде Windows. Излагаются принципы работы стека протоколов TCP/IP и его служб. Описываются интерфейсы Windows Socket, Named Pipe, Mailslots. Отдельно рассматриваются принципы и методы построения параллельного сервера. Изложение сопровождается многочисленными схемами и примерами. Заключительная глава пособия представляет собой практикум, который может быть использован для закрепления изученного материала. Предполагается, что читатель имеет навыки программирования на языке C++ и обладает знаниями основ операционных систем.

Пособие предназначено для студентов специальности «Информацонные системы и технологии (издательско-полиграфический комплекс)», а также для студентов старших курсов высших учебных заведений, изучающих технологии разработки распределенных приложений.

УДК 004.4(075.8)
ББК 22.18я7

ISBN 978-985-434-764-6
© УО «Белорусский государственный технологический университет», 2008
© Смелов В. В., 2008

ПРЕДИСЛОВИЕ

Данное учебно-методическое пособие посвящено разработке простейших распределенных приложений в среде Windows. При этом предполагается, что читатель имеет навыки программирования на языке C++, обладает знаниями основ теории операционных систем, локальных компьютерных сетей.

Первая глава пособия посвящена описанию стандартных моделей взаимодействия процессов в распределенном приложении.

Во второй главе рассматривается структура стека протокола TCP/IP, назначение и основные принципы взаимодействия его компонентов.

В третьей главе пособия описываются главные возможности интерфейса Windows Socket API, являющегося основой для построения распределенных приложений.

Четвертая и пятая главы посвящены средствам межпроцессного взаимодействия, которые тоже используются для построения распределенных приложений в локальной сети.

В шестой главе рассматриваются основные принципы построения и программирования параллельного сервера.

Заключительная (седьмая) глава состоит из десяти практических работ, предназначенных для закрепления изученного материала. Чтобы их выполнить, необходимо наличие нескольких компьютеров с операционной системой Windows XP, связанных локальной сетью. Для разработки приложений на языке C++ требуется среда разработки Microsoft Visual Studio не ниже седьмой версии и доступ к соответствующей версии MSDN. Практические работы состоят из нескольких связанных заданий.

Следует предупредить читателя о том, что описания интерфейсов и функций операционной системы Windows, приведенные в пособии, не является полными и не могут быть использованы в качестве серьезного справочника, т. к. предназначены только для решения той узкой задачи, которую ставит перед собой автор. Для более глубокого и полного изучения следует обратиться к литературе, приведенной в конце пособия, и, конечно же, к соответствующей документации.

ВВЕДЕНИЕ

Термины ***«распределенная архитектура»***, ***«распределенная обработка данных»***, ***«распределенные вычисления»***, ***«распределенная система»*** и т. п. широко используются в технической литературе, посвященной современным компьютерным технологиям. Говорят о распределенной архитектуре компьютера (многопроцессорные или многомашинные вычислительные комплексы), распределенных операционных системах, распределенных базах данных. Все эти понятия объединяет наличие вычислительных и информационных ресурсов, распределенных в пространстве, а также процессов, использующих эти ресурсы.

Под вычислительными ресурсами, как правило, подразумевают компьютеры (процессоры, оперативная и вторичная память), а под информационными – файлы данных и (или) каналы передачи данных. Процессы – это программы, работающие на компьютерах и использующие вычислительные ресурсы для решения задач.

Простейшим примером распределенной системы могут служить два отдельно стоящих компьютера (даже не объединенных в сеть), решающие общую задачу. Такая архитектура системы может быть применена, например: для повышения надежности системы; для решения трудоемкой, но допускающей распараллеливание задачи (задача разбивается на подзадачи, каждая из которых решается на отдельном компьютере); для решения задачи, требующей постоянного диалога с двумя различными специалистами (примером может служить задача составления бухгалтерского баланса, когда два бухгалтера обрабатывают различные группы счетов).

В некоторых случаях может потребоваться обмен информацией между компьютерами. В простейшем случае это может быть сделано с помощью магнитного носителя (дискеты, CD-диска и т. д.), а может быть использована локальная сеть или другие специальные устройства. Заметим, что обмен информацией не может быть произведен в произвольное время: процесс-источник должен успеть подготовить информацию, а процесс-

приемник должен быть готов принять и использовать ее. В связи с тем, что процессы на разных компьютерах работают асинхронно (независимо друг от друга), для обмена данными требуется иногда синхронизировать их работу.

Рассмотрим более сложный пример распределенной системы. Пусть система содержит 300 компьютеров (например, банк), объединенных в единую сеть и решающий единую задачу (учет денежных потоков в банке). На каждом компьютере работает несколько программ, обеспечивающих работу нескольких подсистем единой банковской системы (например, кроме собственно банковской системы, в банке может работать внутренняя почтовая система, выход в глобальные сети и т. д.). Кроме того, как правило, в таких больших компьютерных системах работают компьютеры разного типа, управляемые различными операционными системами и объединенные различными каналами. Ясно, что управление в такой системе само по себе является сложной задачей.

Под ***распределенным приложением*** будем понимать несколько процессов, как правило, работающих на разных компьютерах или разных процессорах, предназначенных для решения общей задачи. Распределенное приложение может принимать различные формы: в простейшем случае распределенное приложение – это две программы, работающие на разных компьютерах и обменивающиеся данными; в другом – это может быть одна или несколько программ, работающих на разных компьютерах с сервером базы данных (например, Microsoft SQL Server или Oracle Server); в третьем – это Active Server Page-приложение, для доступа к которому необходим браузер. Заметим, что в рамках распределенного приложения могут использоваться уже готовые компоненты (серверы баз данных, веб-серверы, браузеры), позволяющие существенно упростить его разработку.

Проанализировав приведенные выше примеры, можно выделить следующие проблемы, с которыми сталкивается разработчик распределенного приложения:

1) обмена данными между процессами в распределенном приложении;

2) синхронизации процессов в распределенном приложении;

3) совместного использования ресурсов процессами распределенного приложения;

4) взаимодействия процессов, работающих в разных операционных средах;

5) масштабирования распределенного приложения.

Проблема совместного использования ресурсов возникает, например, при использовании файла данных двумя или более процессами одного или нескольких приложений. Если при этом один процесс записывает (или обновляет) данные, а другой читает, то требуются специальные механизмы для обеспечения согласованности этих операций. Такая же проблема возникает в многозадачных операционных системах [2].

Под масштабированием понимается возможность расширения распределенного приложения (увеличения числа взаимодействующих процессов, увеличение объема данных и т. д.).

Сетевые и распределенные операционные системы предоставляют собственные методы решения данных проблем. Например, для передачи данных в TCP/IP-сети можно использовать интерфейсы сокетов или именованных каналов, если компьютеры управляются операционными системами Windows или Unix. Для синхронизации процессов, работающих на различных компьютерах, операционная система NetWare предоставляет механизм семафоров. Проблемы совместного использования общих данных обычно решается с помощью серверов систем управления базами данных. Независимость распределенного приложения от операционной среды обеспечивается правильным выбором средств разработки. Например, распределенное приложение, разработанное на языке Java, которое использует для управления данными СУБД Oracle, может работать практически в любой операционной среде.

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 гг. 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, т. е. он 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 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 включает два подуровня: управление доступом к среде передачи данных и управление 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 определяет методы адресации и маршрутизации компьютеров в сети. В отличие от канального уровня он определяет единый метод адресации для всех компьютеров в сети независимо 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 включает процедуры, осуществляющие подготовку и доставку пакетов данных между конечными точками без ошибок и в правильной последовательности. Они формируют файлы для сеансового уровня из пакетов, полученных от сетевого уровня.

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

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

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

Прикладной уровень предназначен для определения способа взаимодействия пользователей с системой (определение интерфейса).

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, которые 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 их по физическому каналу; при движении от физического уровня до прикладного происходит обратное преобразование. При такой организации обмена данными фактическое взаимодействие осуществляется между одноименными уровнями (на рисунке это взаимодействие обозначено линиями со стрелками между уровнями двух систем).

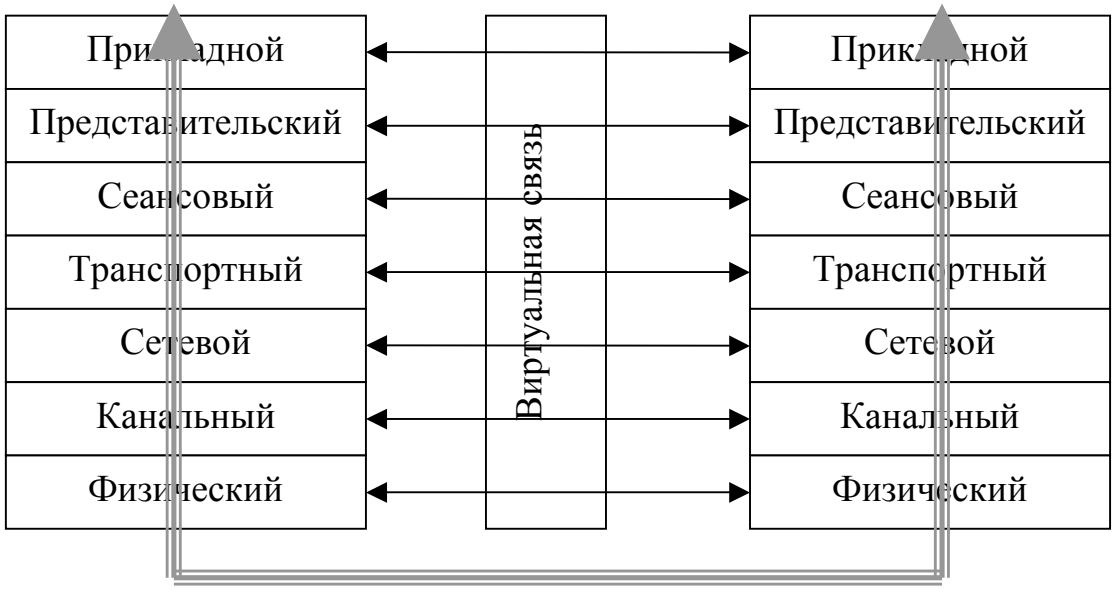

Рис. 1.2.1. Схема взаимодействия открытых систем

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.

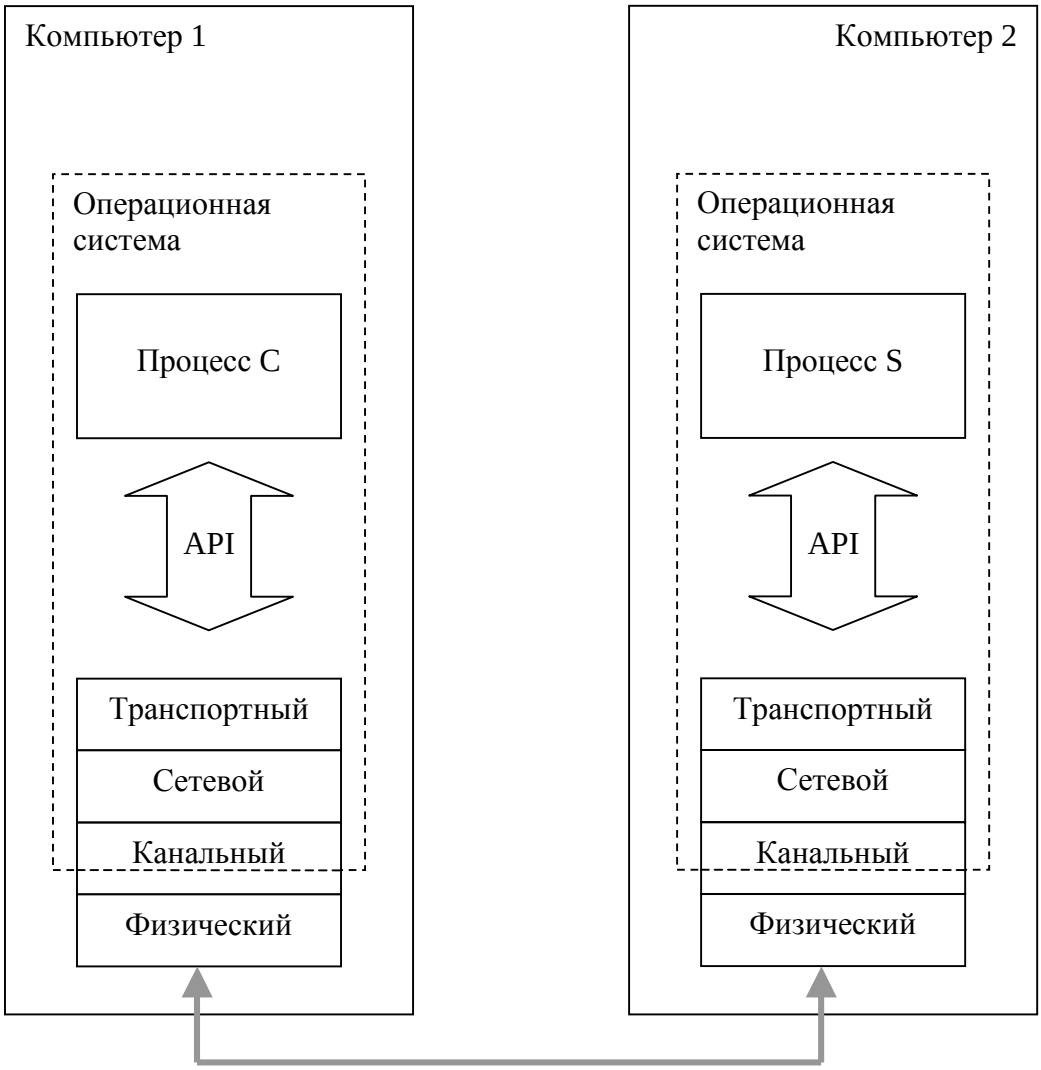

Рис. 1.2.2. Схема взаимодействия процессов в распределенном приложении

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 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 рис. 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 обеспечиваются самим распределенным приложением.

1.3. Архитектура «клиент – сервер»

Распределенное приложение, имеющее архитектуру «клиент – сервер», включает процессы-серверы и процессы-клиенты. Далее их будем назвать просто серверами и клиентами.

Следует отметить, что приведенное в разделе 1.1 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 он 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 распознан сервером для того, чтобы во-первых, можно было его отличить от других клиентов, а во-вторых для обмена с клиентом данными. Если основная вычислительная нагрузка ложится на сервер, а клиент лишь обеспечивает

интерфейсом пользователя с сервером, то такого клиента часто называют ***тонким***.

По методу облуживания серверы подразделяются на ***итеративные*** и ***параллельные*** (iterative and concurrent servers). Принципиальная разница между ними 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 ему. По всей видимости можно говорить об итеративно-параллельных серверах, имеющих ограниченные возможности по распараллеливанию своей работы. В этом случае только часть клиентских запросов будет обслуживаться параллельно.

1.4. 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, предоставленным 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 (WebSphere, JBOSS, WebLogic), 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, давших 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 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 процедур, которые должны быть им пройдены для превращения в официальный стандарт группы IETF. Если есть документы, имеющие одно название, но разные номера, то документ с самым большим номером считается текущей версией.

Более подробно с историей создания TCP/IP, назначением групп стандартизации и процедурами создания и утверждения документов RFC можно ознакомиться в книгах [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 рис. 2.2.1 изображены уровни обеих моделей и их соответствие. Они похожи, но не идентичны.

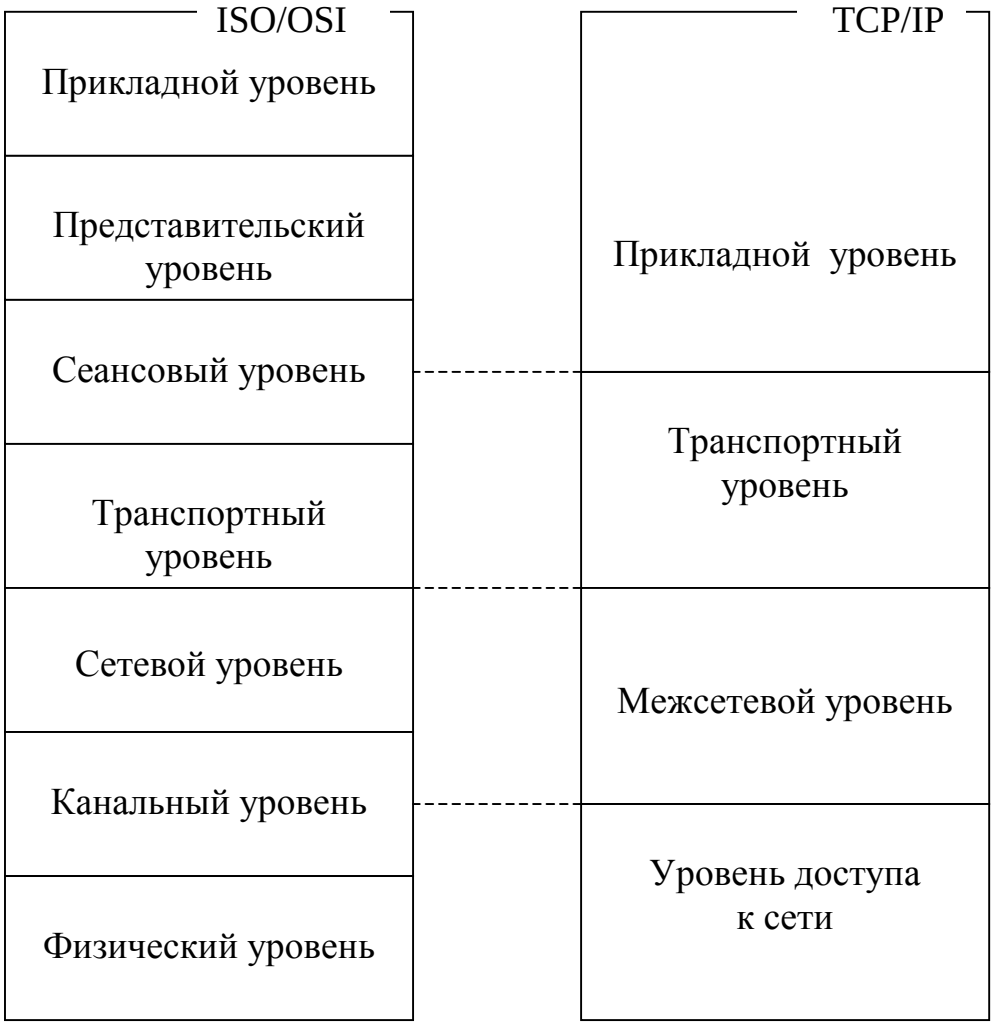

Рис. 2.2.1. Уровни моделей ISO/OSI и TCP/IP

Структура TCP/IP является более простой: в ней не выделяются физический, канальный, сетевой и представительский уровни. В 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.

В дальнейшем при описании протоколов стека TCP/IP будут 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), включающим 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 длину, большую MTU, то он производит фрагментацию, разбивая дейтаграмму на части, имеющие длину меньшую, чем MTU. В табл. 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. корпорациями DEC, Intel и Xerox (DIX) совместно.

На сегодня 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. Рань-ше он использовался для подключения домашних компьютеров
к Internet через последовательный порт RS-232. Протокол использует простейшую инкапсуляцию кадра и имеет ряд недостат-
ков: 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 недостат-ков были исправлены в новой версии протокола 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. PPP может применяться для технологии ISDN (Integrated Services Digital Network) и SONET (Synchronous Optical Network). Протокол

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
используемый протокол для последовательного канала, обеспечивающий соединение компьютера с сетью Internet, практически вытеснивший 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 ***дейтаграмм*** (так называется единица передачи данных в терминологии IP). При этом протокол по определению является ***ненадежным*** и ***не поддерживающим соединения***.

Ненадежность протокола IP обусловлена отсутствием гарантии того, 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 из них 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 физического 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

хоста-
ми (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. При записи IP-адреса, как правило, используются 4 десятичных числа, разделенные точкой. Каждое из них является десятичным представлением 8 битов (***октет*** в терминологии TCP/IP) адреса. На рис. 2.4.1 разобран пример перевода IP-адреса из двоичного формата в десятичный.

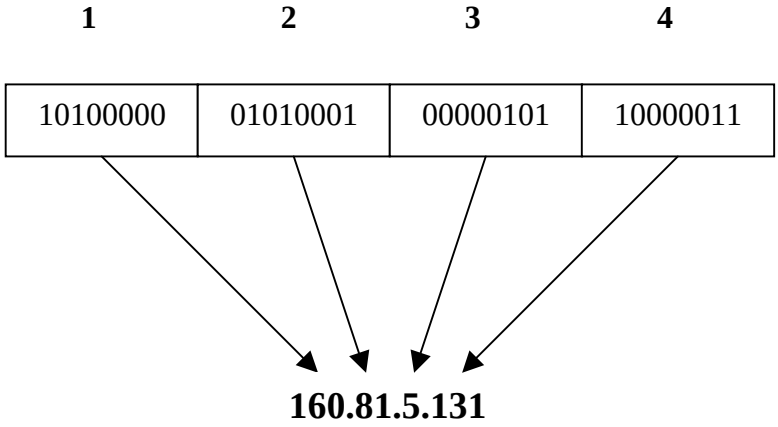

Рис. 2.4.1. Представление IP-адреса в десятичном формате

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 рис. 2.4.2 приведены форматы адресов для всех классов.

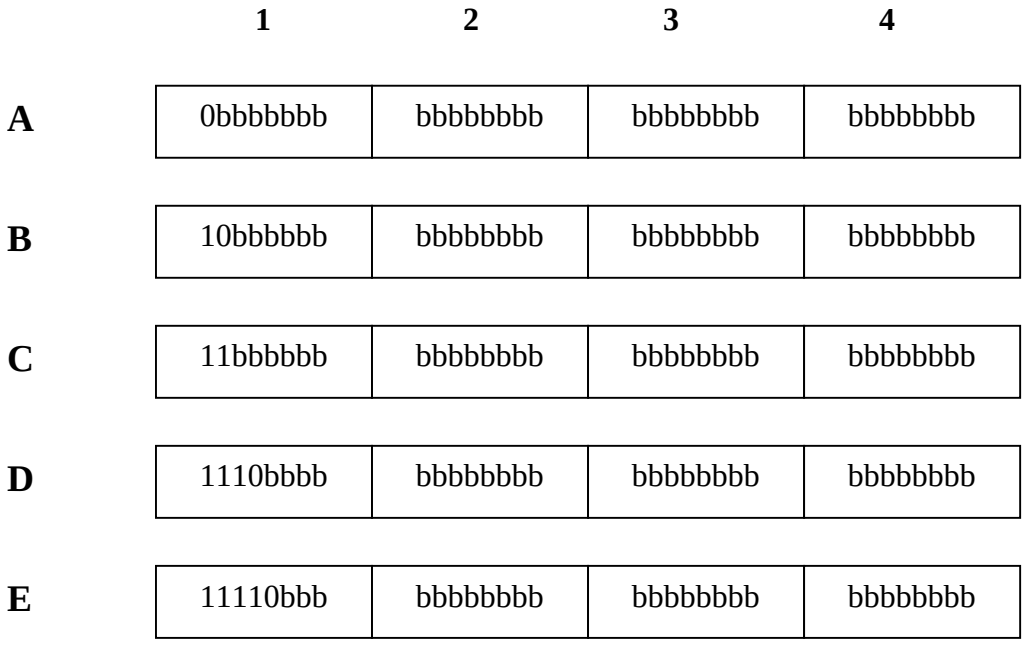

Рис. 2.4.2. Форматы IP-адресов в классовой модели адресации

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 источниках литературы [5, 6]. Распределение октетов IP-адреса на адрес сети и адрес хоста для классов A, B и C приводится на рис. 2.4.3. Закрашенные октеты обозначают часть IP-адреса, отведенную для адреса сети, незакрашенные – 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.

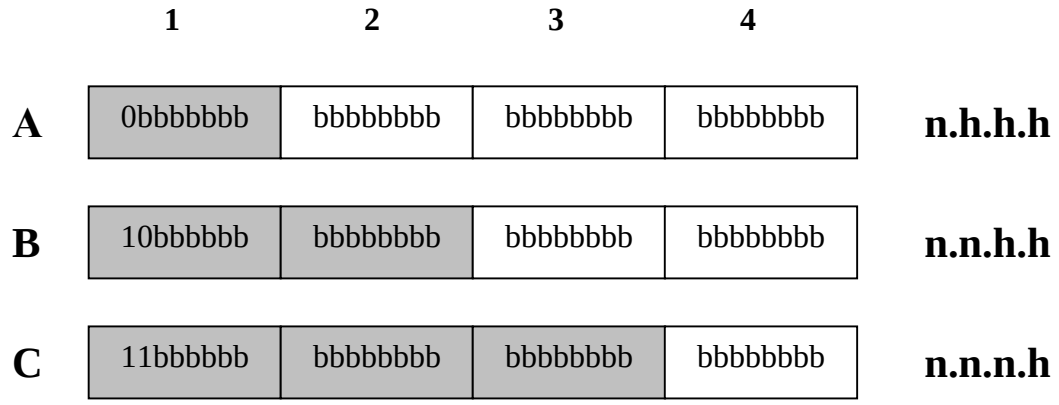

Рис. 2.4.3. Распределение разрядов IP-адреса на адрес сети и адрес хоста для классовой модели IP-адресов

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 невозможна. 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 использовать 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 табл. 2.4.1.

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

Применение классовой модели адресации не всегда удобно. Ее альтернативой является ***бесклассовая междоменная маршрутизация – CIDR*** (Classless Inter-Domain Routing). CIDR позволяет произвольным образом назначать границу сетевой и хостовой части IP-адреса. Для этого каждому из них 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 рис. 2.4.4 разобран пример вычисления адреса сети с помощью маски подсети.

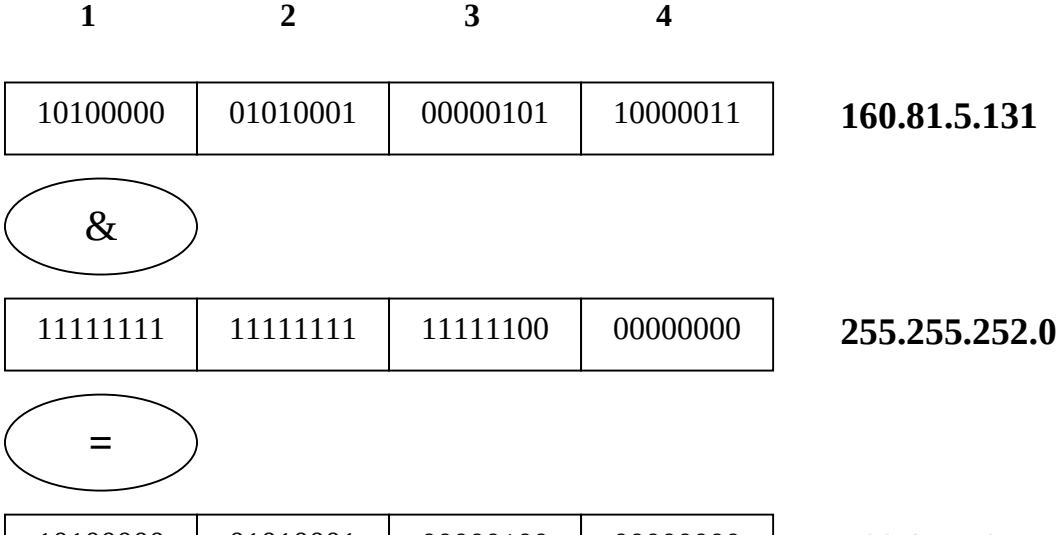

Рис. 2.4.4. Вычисление адреса сети с помощью маски подсети

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 рис. 2.4.5 изображен пример составной IP-сети. Шлюз имеет два интерфейса: один принадлежит сети 172.16.5.0, другой – сети 172.16.12.0. Обе сети имеют маску 255.255.252.0, что соответствует 22 битовому адресу. Для обмена данными с хостом, который находится в другой сети, используется ***таблица маршрутов***. Она имеется на каждом ***узле*** сети (хост или шлюз) и содержит информацию об адресах сетей, 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 книгах [5, 6].

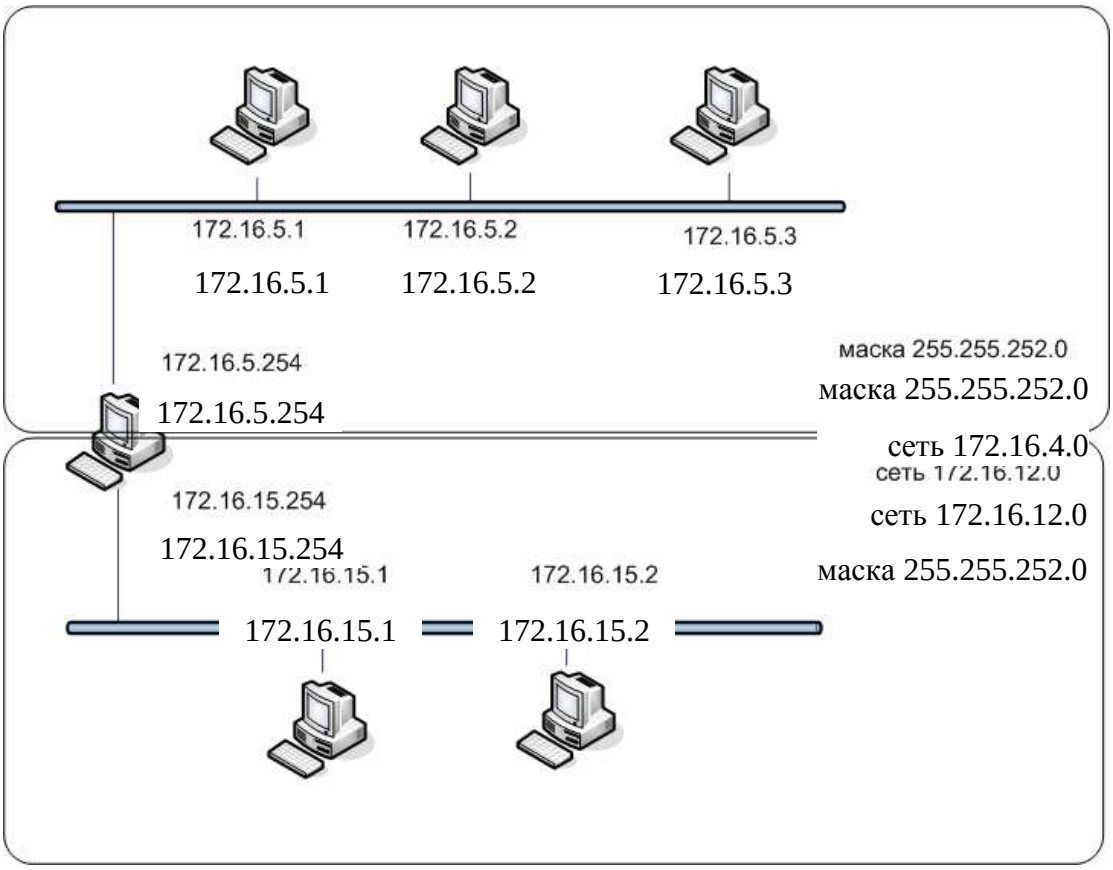

Рис. 2.4.5. Пример составной IP-сети

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. Опи-сание 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 шлю-зам 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 проекция IP-адресов в соответствующие MAC-адреса аппаратных средств (без вмешательства администратора, пользователя, прикладной программы). Эффективность работы ARP обеспечивается тем, что каждый хост кэширует специальную ARP-таблицу. Время существования записи в этой таблице составляет обычно 10–20 мин с момента ее создания и может быть изменено с помощью параметров реестра [6]. Просмотреть текущее состояние ARP-таблицы можно с помощью команды ***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 (Reverse ARP) по своей функции противоположен протоколу ARP. RARP применятся для получения IP-адреса по MAC-адресу. В настоящее время он 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 источниках литературы [5, 6].

Протокол IPv6 является новой усовершенствованной версией протокола IP**.** Наиболее распространенная на настоящий момент версия протокола IP является IPv4 – именно об этой версии говорилось выше. Этот протокол оказался самым удачным сетевым протоколом из 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 соответствия 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, 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 необходимостью поддерживать мобильных (перемещающихся) пользователей. Большое внимание в новой версии протокола уделяется вопросам безопасности. Более подробно о перспективном протоколе IPv6 можно узнать из книги [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) и **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 про-токолов UDP и TCP является то, что UDP – протокол без установления соединения (ориентирован на сообщения), а TCP – прото-кол на основе соединения (ориентирован на поток). На рис. 2.5.1 изображен стек протоколов TCP в двух разрезах и 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, пригодном для использования процессом.

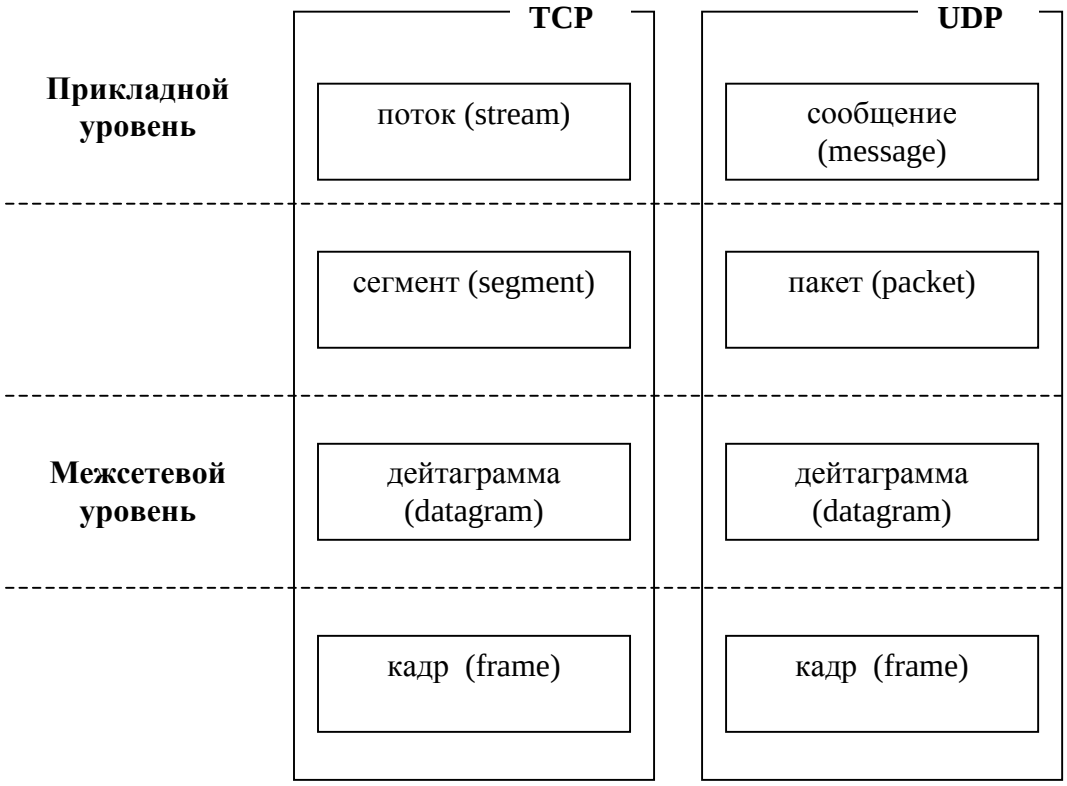

Рис. 2.5.1. Протоколы TCP и UDP в стеке TCP/IP

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), ***динамические номера портов*** (dynamic port number). Распределение номеров портов по диапазонам приведено в табл. 2.5.1.

Таблица 2.5.1

Хорошо известные номера портов	0 – 1023
Зарегистрированные номера портов	1024 – 49 151
Динамические номера портов	49 152– 65 535

Хорошо известные номера портов присваиваются ***базовым системным службам*** (core services), имеющим системные привилегии. Зарегистрированные номера портов присваиваются промышленным приложениям и процессам. Распределение некоторых
хорошо известных и зарегистрированных номеров портов приведено в табл. 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 является протоколом без установления соединения. Его спецификация описывается в документе RFC 768. Основные свойства протокола:

1) отсутствие механизмов обеспечения надежности: пакеты не упорядочиваются, 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 внимание, что все перечисленные отсутствующие характеристики UDP присутствуют в протоколе TCP. Фактически UDP – это тонкая прослойка интерфейса, обеспечивающая доступ процессов прикладного уровня непосредственно к протоколу IP.

Протокол TCP является ***надежным*** байт-ориентированным протоколом ***с установлением соединения***. При получении дейтаграммы в поле Protocol (со структурой IP-дейтаграммы можно ознакомиться в книгах [5, 6]) указан код 6 (код протокола TCP), IP-протокол извлекает из нее данные, предназначенные для транспортного уровня, и переправляет их модулю протокола TCP. Модуль TCP анализирует служебную информацию заголовка сегмента (структура TCP-сегмента приведена в книгах [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 неинтерпретируемый (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. Немедленная отправка данных может быть затребована процессом с помощью специальной функции PUCH, иначе TCP сам решает, когда отправлять и передавать данные 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 рис. 2.5.2 прерывистыми линиями изображены каналы между процессами прикладного уровня A, B, C, D и E.

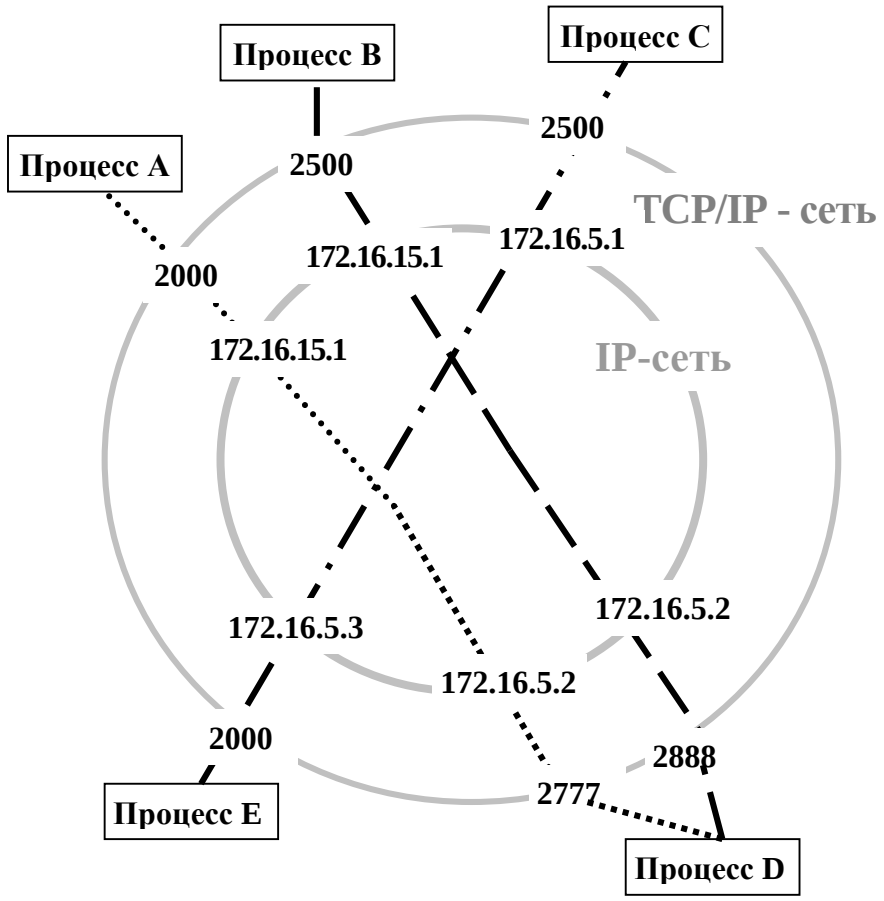

Рис. 2.5.2. Разделение каналов в сети TCP/IP

Процесс D работает на хосте с IP-адресом 172.16.5.2 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.

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.

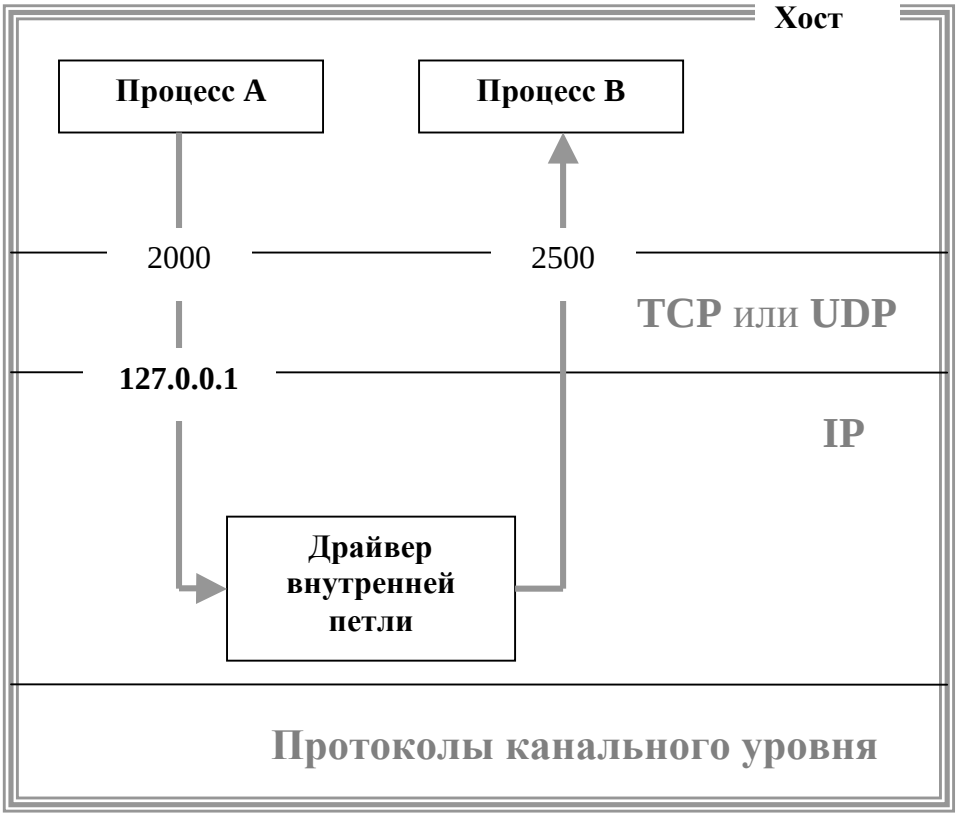

Рис. 2.6.1. Схема работы интерфейса внутренней петли

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***.

На рис. 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 В.

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 ***API*** (Application Programming Interface), 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 – 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

конечную точку соединения. Впервые этот интерфейс появился в 1980-х гг. 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.), 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, веб-серверы, серверы баз банных, приложений и т. п.

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 – ***TI-RPC*** (Transport Independent RPC) использует API TLI (Transport Layer Interface компании «AT&T») и способна работать с любым транспортным протоколом, поддерживаемым 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 преобразование передаваемых параметров в формат стандарта ***XDR*** (External Data Representation) и скрытие (подмена вызываемой удаленной процедуры локальным вызовом стаба) от пользователя механизма 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 процедурой, а также для сокрытия RPC-механизма от вызываемой процедуры (серверный стаб подменяет вызывающую процедуру на стороне сервера).

Стандарт XDR предназначен для 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. Номера портов получают сами удаленные процедуры

динамическим образом (эфемерные порты). Учет соответствия портов вызываемым процедурам осуществляет специальная программа ***PortMapper*** (регистратор портов). Она доступна по 111 порту
и 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***, ***DCOM*** и ***COM+***, 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, службы реализуются в виде сервера, предоставляющего услуги клиентам (другим процессам). В этом случае клиенты использует запросы, определенные

спецификациями, и получают соответствующий сервис, предусмотренный данным запросом. Однако ярко выраженной архитектурой «клиент – сервер» обладают не все службы. В некоторых случаях сами службы могут выступать в роли клиента или осуществлять межсерверный обмен данными (между однородными серверами), 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, появление которой в 1980-х гг. 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 IP-адрес 207.46.230.229 сервера компании Microsoft можно использовать 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, призванную отвечать на два вида запросов: выдать IP-адрес по символическому имени хоста, и наоборот – 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** и др. Они отражают 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*** (являются главными серверами, поддерживающими свою часть базы данных DNS), ***вторичные*** (всегда привязаны к некоторому первичному серверу и используются для дублирования данных первичного сервера), ***кэширующие*** (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 в настоящее время 15 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 источнике [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 хостам (маска подсети, адрес локального шлюза, адреса серверов DNS и т. п.). 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: ***ручное***, ***автоматическое*** и ***динамические*** [5, 6]. Обычно DHCP-серверы устанавливают на компьютерах, исполняющих роль сервера в сети.

DHCP-клиент представляет собой программный компонент, обычно реализуемый как часть стека протоколов TCP/IP и предназначенный для формирования и пересылки запросов к DHCP-серверу на выделение IP-адреса, продления его срока аренды 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 он 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 по разным причинам не могут быть доставлены DHCP-серверу. Ретранслятор в этом случае играет роль посредника между DHCP-клиентом и DHCP-сервером.

Протокол DHCPv6 – это новый протокол, работающий над IPv6. Основные задачи DHCPv6 не сильно отличаются от задач выполняемых 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 всей 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 г. в компании «Sytek», а позже заимствована «IBM» и «Mic-rosoft» 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 г. в документах RFC 1001, 1002.

В настоящее время 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 My Computer. 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 книгах [5, 6].

2.8.4. Служба и протокол Telnet

Сетевая служба Telnet как протокол описана в двух документах RFC 854 (собственно протокол Telnet, 1985 г.) и RFC 855 (дополнительные возможности Telnet). Служба создавалась для подключения пользователей к удаленному компьютеру, чтобы производить вычислений, работать с базами данных, подготавливать документы
и т. д. Кроме того, служба применяется для управления работой удаленного компьютера, 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 (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 форму, понятную для обеих сторон, 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. Для этого 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 существуют 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). Имя FTP носит служба, реализующая этот протокол. Описание протокола FTP содержится в документе RFC 959 (1985 г.).

Служба FTP имеет архитектуру «клиент – сервер» (рис. 2.8.5) и содержит 3 ключевых компонента: UI(User Interface), PI (Protocol Interpreter) и DTP (Data Transfer Process).

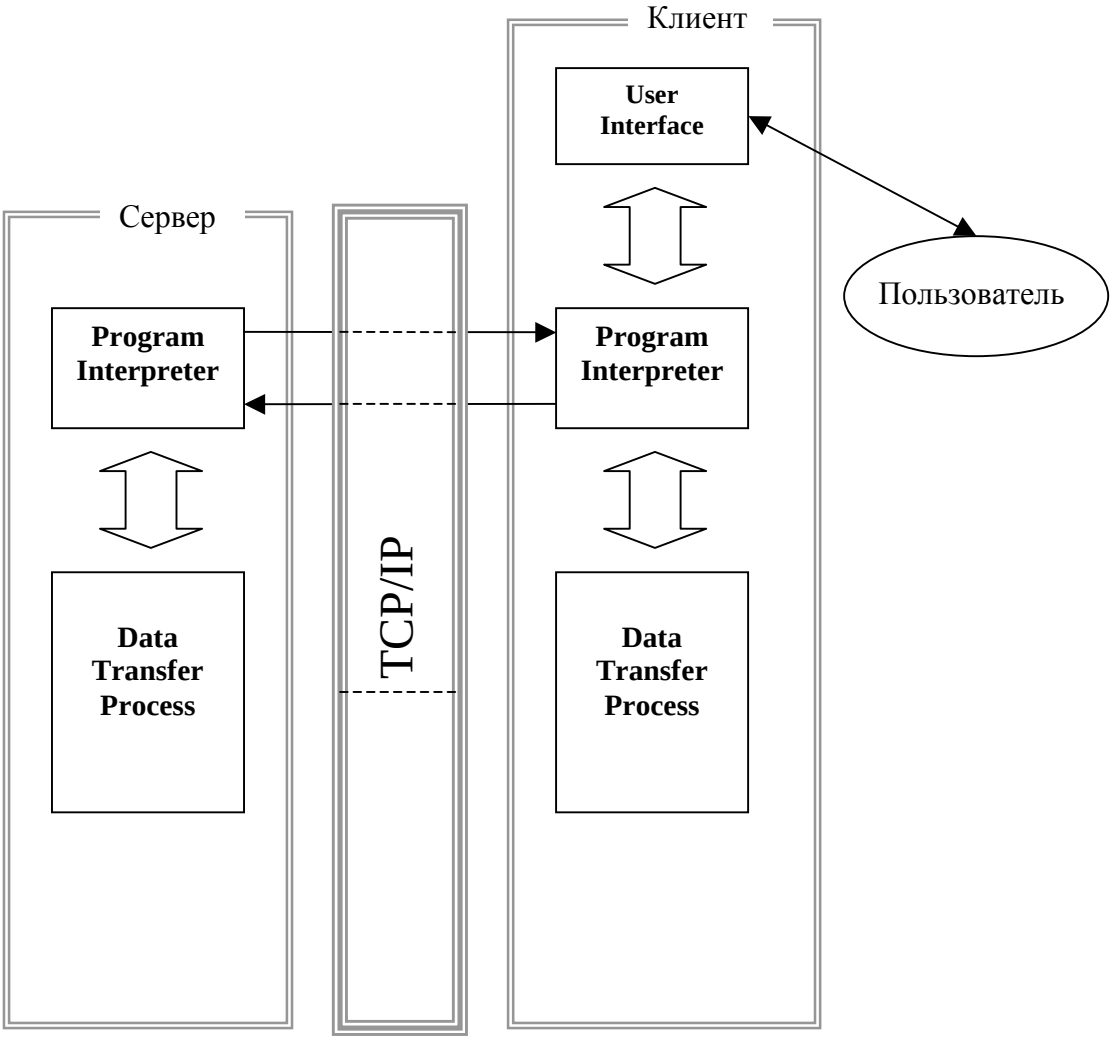

Рис. 2.8.5. Взаимодействие основных компонентов службы FTP

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 от него 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 переданными по каналу управления командами. 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.

Основными компонентами системы электронной почты являются: ***MTA*** (Mail Transport Agent), ***MDA*** (Mail Delivery Agent), ***POA*** (Post Office Agent) и ***MUA*** (Mail User Agent). На рис. 2.8.6 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 них; вызов MDA. Этот 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, который позволяет принимать почту по протоколам POP3 и IMAP4 и отправлять почту по протоколу SMTP.

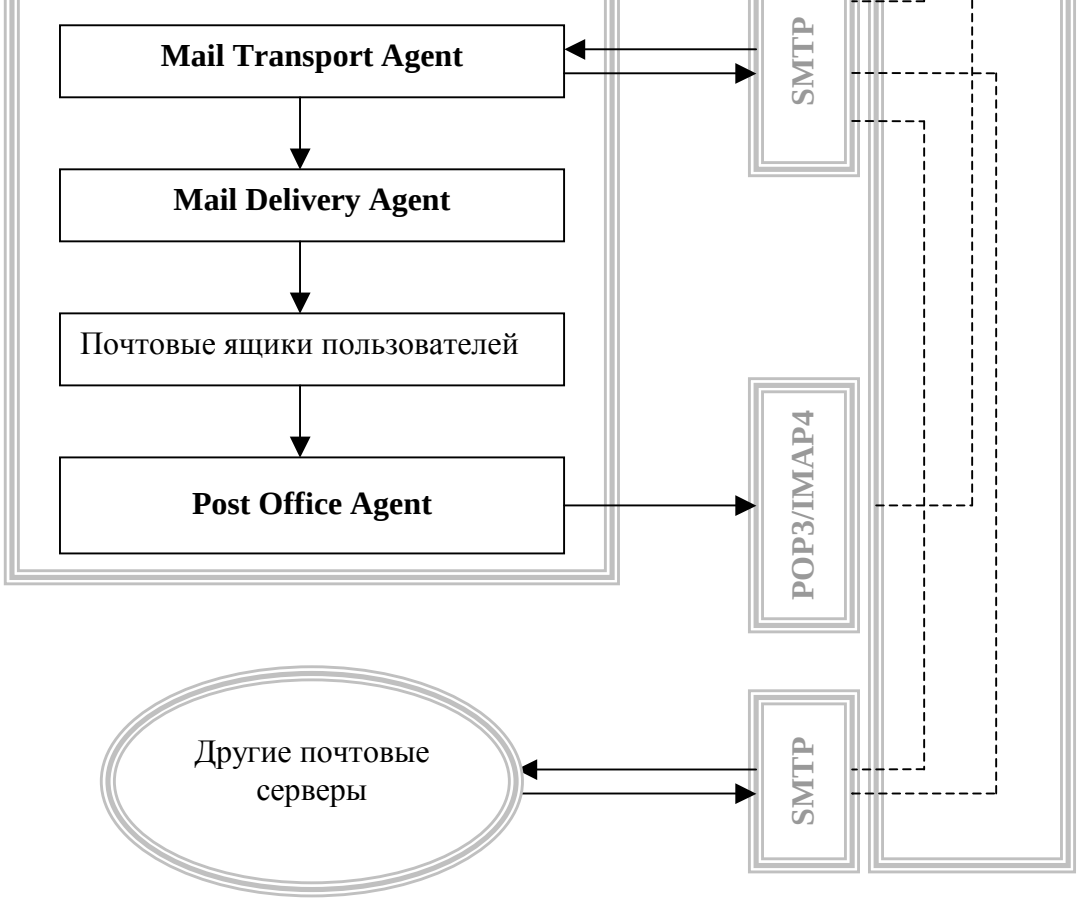

Рис. 2.8.6. Схема взаимодействия компонентов электронной почты

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, расположенных 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 почтовым сервером, 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 различным гипермедийным системам. Под ***гипермедийной системой*** понимается компьютерное представление системы данных, элементы которой хранятся в различных форматах (гипертекст, графические

изображения, видеоизображения, звук и т. д.) и обеспечивают автоматическую поддержку смысловых связей между этими элементами.

Протокол HTTP применяется в Internet с 1990 г. В настоящее время широкое распространение имеет версия HTTP 1.0, описанная в документе RFC 1945. Разработана новая версия HTTP 4.01 (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, предназначенный для указания места нахождения ресурса в сети. URN – унифицированное имя ресурса, идентифицирующее его 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) .

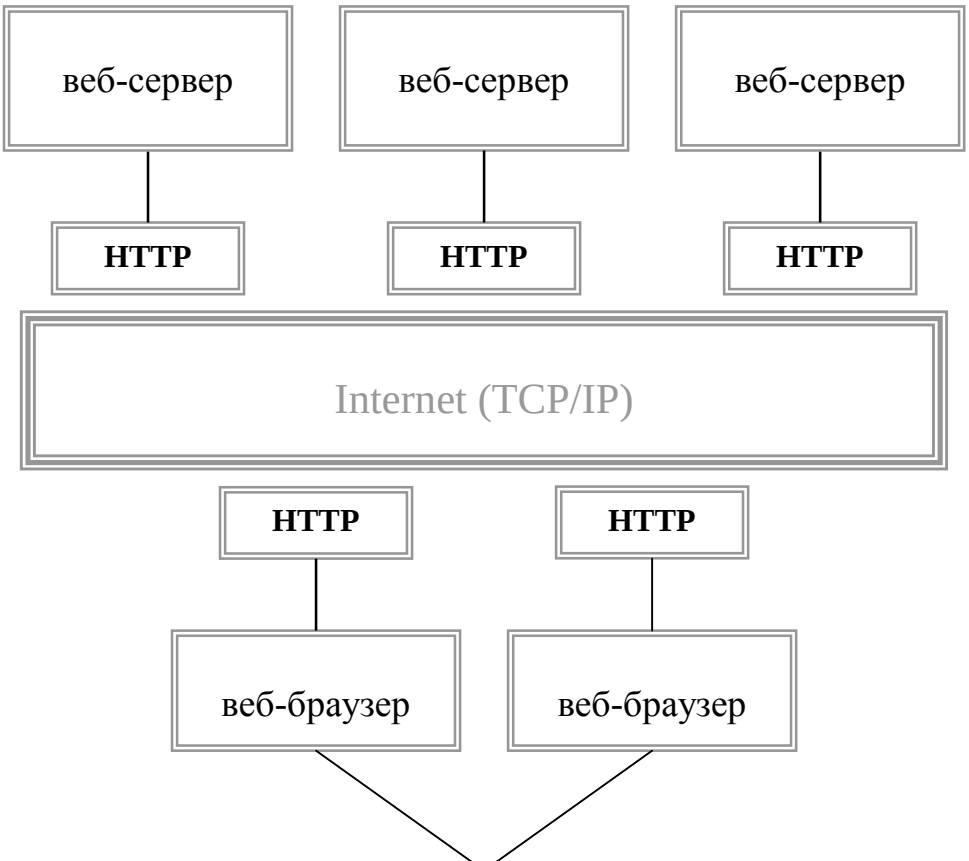

Рис. 2.8.7. Клиент-серверная архитектура службы WWW

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 является то, что форматирование в документе записывается только с помощью ASCII-символов. Одним из ключевых понятий гипертекстового документа является ***гипертекстовая ссылка*** – это объект гипертекстового документа, предназначенный для обозначения URI другого гипертекстового документа.

Как и все службы Internet, служба WWW имеет архитектуру «клиент – сервер» (рис. 2.8.7). Серверная и клиентская части службы (обычно называемые веб-сервером и веб-браузером) 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 веб-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 табл. 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

Утилита	Назначение
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 в своей работе использует протокол ICMP и предназначена для проверки соединения с удаленным хостом, осуществляемой 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-имя или DNS-имя хоста или его IP-адрес.

Утилит 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-имя или DNS-имя хоста или его IP-адрес.

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 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 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 ее может осуществлять только пользователь с правами администратора.

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

arp -a

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, адресов и имен почтовых серверов, ответственных за доставку почты для отдельных доменов DNS, почтового адреса администратора DNS-сервера и т. д. 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.

На рис. 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 предназначена для вывода на консоль имени хоста, на котором выполняется данная команда. Команда hostname не имеет никаких 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
```

Рис. 2.9.1. Пример выполнения команды nslookup

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, поскольку позволяет проверить параметры IP-интерфейсов, установленные в автоматическом режиме.

```
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
```

Рис. 2.9.2. Пример выполнения команды ipconfig

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 рис. 2.9.2.

Утилита nbtstat позволяет просматривать статистику текущих соединений, использующих протокол NBT (NetBIOS over TCP/IP). Она 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 является основным средством управления сетью для сетевого клиента Windows. Команду net часто включают в скрипты регистрации и командные файлы. С помощью ее можно зарегистрировать пользователя в рабочей группе Windows, 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
```

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 TCP/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 на 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 рис. 2.10.1.

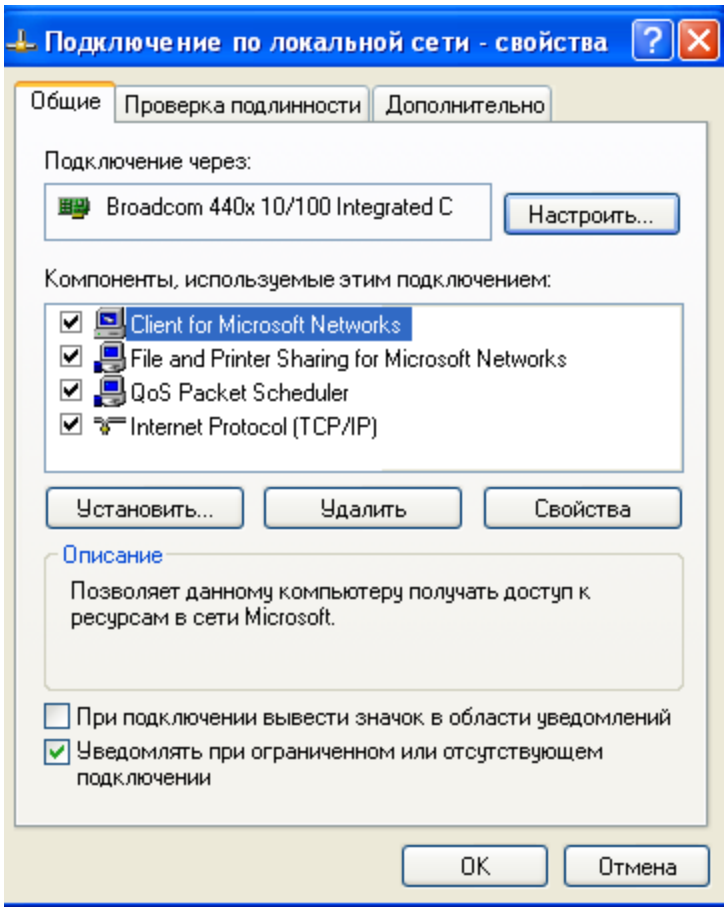

Рис. 2.10.1. Окно для настройки параметров TCP/IP

Если протокол TCP/IP отсутствует в данном подключении, то установка его может быть выполнена с помощью кнопки **Установить**.

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 соответствующие 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.

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 книгах [7, 10, 11].

2.11. Итоги главы

1. Стек протоколов TCP/IP является основным стандартом, используемым для взаимодействия распределенных в сети компонентов программных систем. TCP/IP поддерживается большинством современных операционных систем и применяется практически во всех 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 прикладном 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 они идентифицируют с помощью номера порта, которые разбиты на три группы: хорошо известные (отводятся для базовых служб TCP/IP), зарегистрированные (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, DCOM и COM+, а в Java-технологиях – механизм RMI.

9. Программную реализацию протоколов прикладного уровня TCP/IP называют службами. Они 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- 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
//.....................................................
```

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 книгах [4, 12].

В табл. 3.2.1 приведен список функций интерфейса Windows. Он включает не все функции, а только те, которые будут применяться в дальнейших примерах. Кроме того, описание этих функций, не будет полным, потому что они предназначены только для решения рассматриваемых в пособии задач. С полным описанием можно ознакомиться, например, на сайте www.microsoft.com.

Таблица 3.2.1

Функция	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

Окончание табл. 3.2.1

Функция	Назначение
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. Она вызывается непосредственно сразу после функции Winsock2, завершившейся с ошибкой. Все диагностирующие коды представлены в табл. 3.3.1. Описание функции WSAGetLastError приводится на рис. 3.3.1. На рис. 3.3.2 приведен пример ее использования.

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).

```
// -- 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.
```

Рис. 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;
}
```

Рис. 3.3.2. Пример использования функции WSAGetLastError

Таблица 3.3.1

Код 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

Продолжение табл. 3.3.1

Код возврата функции GetLastError	Причина ошибки
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
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

Окончание табл. 3.3.1

Код возврата функции GetLastError	Причина ошибки
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.

В первом случае 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 данными 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 его. Таким образом, предполагается, что к моменту выдачи запроса клиентом сервер должен быть уже активным, а клиент должен «знать» параметры сокета сервера. На рис. 3.4.1 и 3.4.2 изображены схемы взаимодействия клиента и сервера для первого и второго случаев.

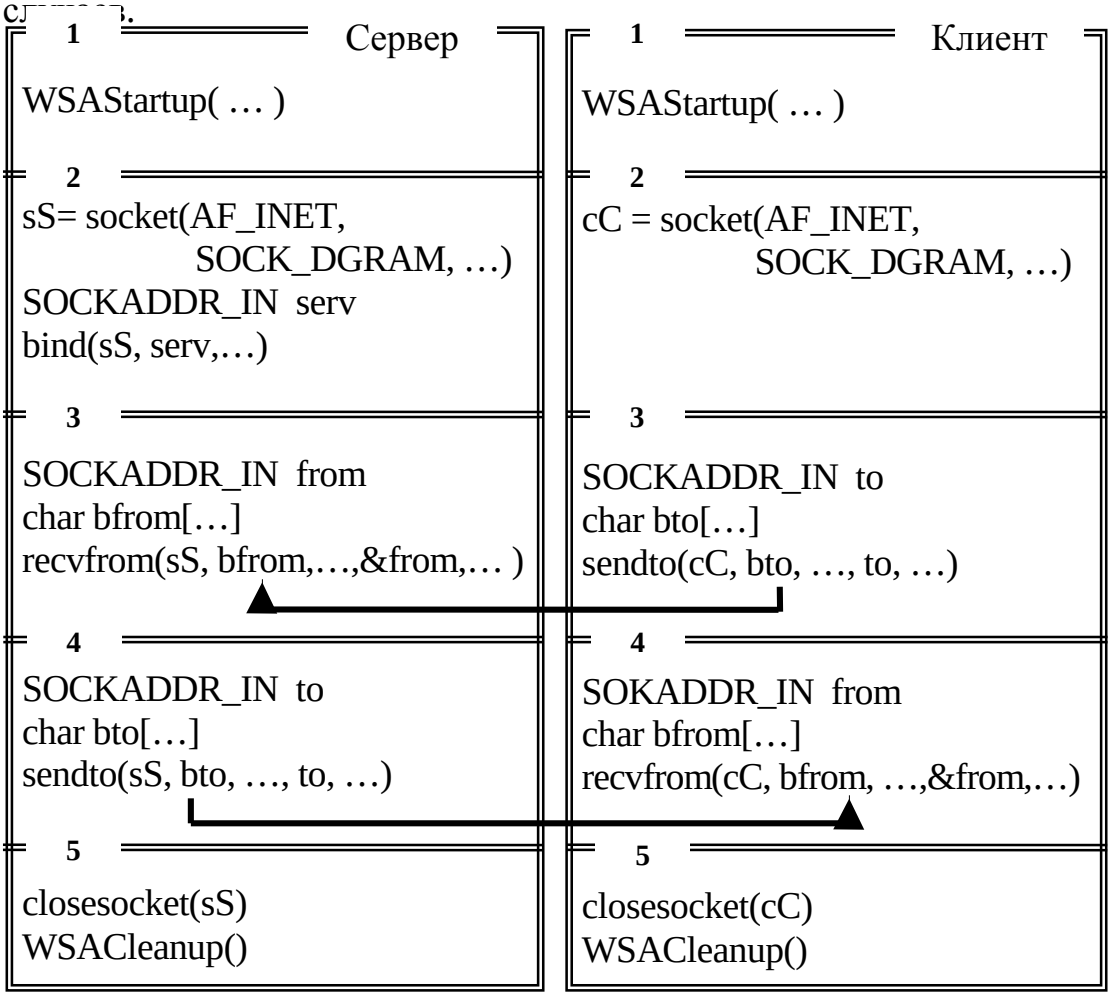

Рис. 3.4.1. Схема взаимодействия процессов
без установки соединения

На рис. 3.4.1 схематично изображены 2 программы, реализующие 2 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 его параметры. 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 (рис. 3.4.1). Перед выполнением функции bind, которая использует эту структуру в качестве параметра, необходимо ее заполнить данными. 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 и именно через него осуществляется передача данных.

Четвертый блок программы сервера предназначен для пересылки данных клиенту. Процесс 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 по сравнению 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. Поскольку 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 рис. 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, указывающее, что сокет будет использоваться для соединения (сокет, ориентированный на поток).

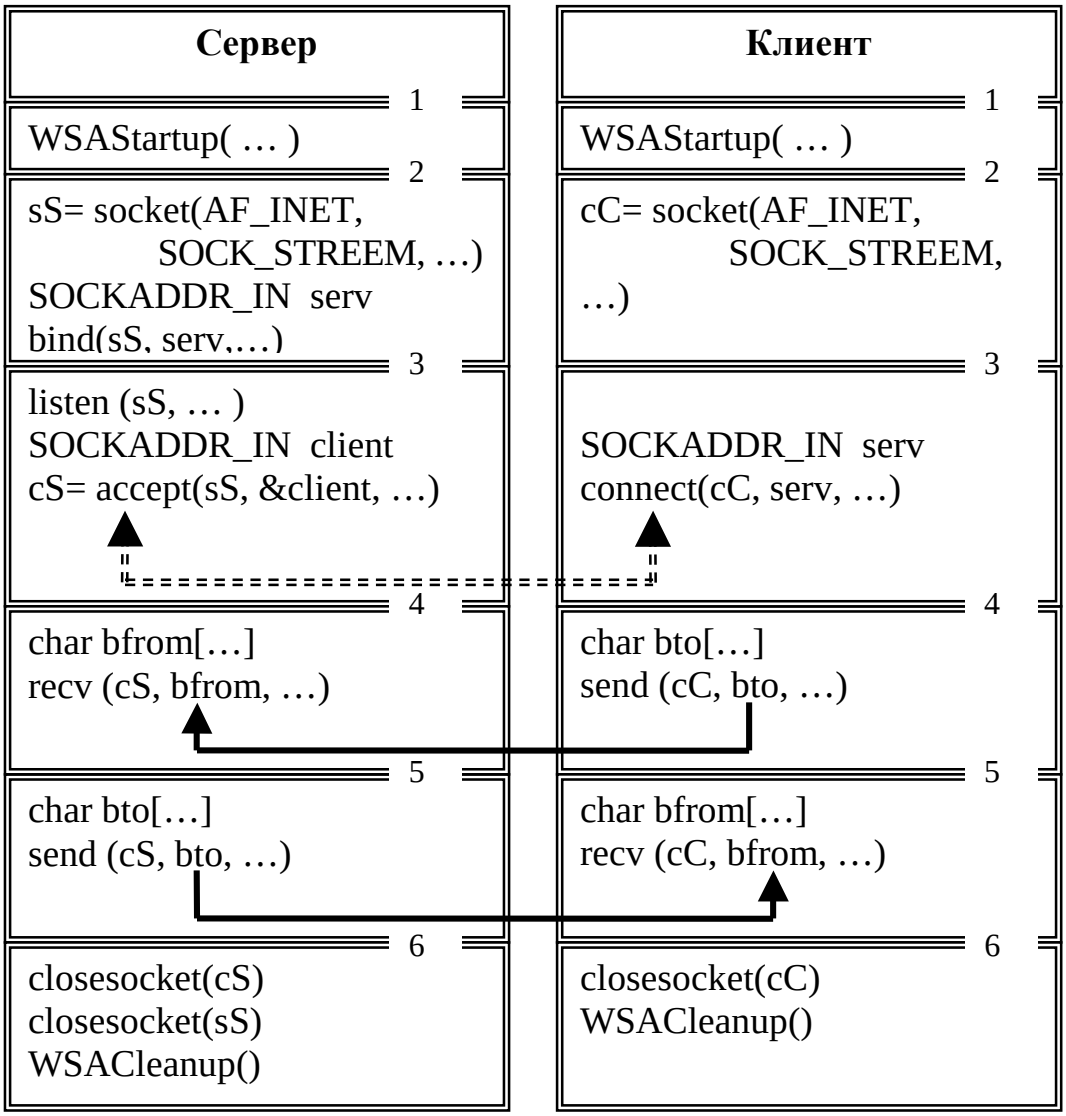

Рис. 3.4.2. Схема взаимодействия процессов
с установкой соединения

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 (установить соединение). 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 третьем 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 (рис. 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 ненулевое  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
```

Рис. 3.5.1. Функция WSAStartup

Как уже отмечалось раньше, функция WSAStartup должна быть выполнена до использования любых функций 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 (рис. 3.6.1). На рис. 3.6.2 приводится пример использования функций WSAStatrup и WSACleanup.

```
// -- 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.
```

Рис. 3.6.1. Функция 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)
```

Рис. 3.6.2. Пример использования функций
WSAStartup и WSACleanup

3.7. Создание и закрытие сокета

Для создания сокета используется функция socket (рис. 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.
```

Рис. 3.7.1. Функция socket

После завершения работы с сокетом обычно его закрывают (освобождают ресурс). Для закрытия сокета применяется функция closesocket (рис. 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.
```

Рис. 3.7.2. Функция closesocket

На рис. 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;
}
```

Рис. 3.7.3. Пример использования функций socket и closesocket

3.8. Установка параметров сокета

Для установки параметров существующего сокета используется функция bind (рис. 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.
```

Рис. 3.8.1. Функция bind

Функция связывает дескриптор сокета и структуру SOCKADDR_IN, которая предназначена для хранения параметров сокета. IIIаблон структуры SOCKADDR_IN содержится в файле Winsock2.h. Описание структуры SOCKADDR_IN и используемых вместе с ней констант приводится на рис. 3.8.2. Особое внимание следует обратить на строки, отмеченные тремя знаками плюс. В дальнейшем эти отмеченные поля и константы будут использоваться в текстах программ. IP-адрес и номер порта в структуре SOCKADDR_IN хранятся в специальном сетевом формате. Он 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;
```

Рис. 3.8.2. Структура SOCKADDR_IN

Для преобразования номера порта в формат TCP/IP следует использовать функцию htons (рис. 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 2 байта  сетевого формата.
//
   u_short htons (
               u_short hp   //[in] 16 битов данных
                  );
//
// Код возврата: 16  битов в формате TCP/IP.
//
```

Рис. 3.8.3. Функция htons

```
// -- 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
//      возвращает  IP-адрес в формате TCP/IP, иначе
//      возвращается INADDR_NONE.
```

Рис. 3.8.4. Функция inet_addr

На рис. 3.8.5 приведен фрагмент программы сервера. Функция bind связывает сокет с параметрами, заданными в структуре SOCKADDR_IN. Она 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());
//.....................................................
```

Рис. 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 (рис. 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.
```

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.

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

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).

```
// -- 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),
//       а указатель la содержит адрес 4 байт с
//       длинной (в байтах) структуры SOCKADDR_IN.
```

Рис. 3.10.1. Функция 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.
```

Рис. 3.10.2. Функция connect

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

Рис. 3.10.3. Фрагмент программы сервера с функцией accept

При этом предполагается, что серверный сокет создан (функции socket и bind) и для него уже выполнена функция listen. На рис. 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;}
```

Рис. 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 (рис. 3.11.1 и 3.11.2). Функция send пересылает по каналу указанного сокета определенное количество байт данных. Функция recv принимает их по каналу указанного сокета. Для работы обеих функций в программе необходимо выделить память (буфер) для приема или оправления данных. Ее размер 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 ее используют, чтобы обозначить конец

передачи. Для принимающей стороны операционная система буферизирует принимаемые данные. Если при очередном приеме данных размеры буфера будут исчерпаны, отправляющей стороне будет выдан 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.
```

Рис. 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.
```

Рис. 3.11.2 Функция recv

```
//.....................................................
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 байт

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

Рис. 3.11.3. Пример использования функций send и 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 приложения 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.

На рис. 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 (рис.

3.12.1 и 3.12.2). При этом предполагается, что сообщения будут курсировать между сокетами, ориентированными на сообщения.

```
// -- 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 переданных 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.
```

Рис. 3.12.1. Функция sendto

```
// -- 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 4 байт с
//       размером структуры SOCADDR_IN;
//     - для параметра flags рекомендуется установить
//       значение NULL.
```

Рис. 3.12.2. Функция recvfrom

Особенностью использования этих функций является то, что протоколом UDP не гарантируется доставка и правильная последовательность приема отправленных сообщений. Весь контроль надежности доставки и правильной последовательности 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.

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

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

Рис. 3.12.3. Пример использования функции recvfrom
в программе сервера

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

Рис. 3.12.4. Пример использования функции sendto
в программе клиента

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

Следует также обратить внимание, что обе функции используют в качестве параметров структуру SOCKADDR_IN. В случае выполнения функции sendto структура должна содержать параметры сокета получателя. У функции recvfrom 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 всем компьютерам сети (широковещательные сообщения). Для этого в параметрах сокета отправляющей стороны используются специальные широковещательные и групповые IP-адреса.

Использование групповых адресов функцией sendto по умолчанию

запрещено. Разрешение на использование широковещательных IP-адресов устанавливается функцией setsockopt.

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 (рис. 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.

Применение 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 до тех пор, пока не разорвется соединение или не поступит пустой блок данных (функция recv вернет нулевое значение).

На рис. 3.13.2 приведен фрагмент программы, использующей функцию TransmitFile. Следует обратить внимание, что пересылаемый файл должен быть открытым с помощью стандартной функции _open.

```
// -- 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
```

Рис. 3.13.1. Функция TransmitFile

Кроме того, в качестве параметра функция TransmitFile использует системный дескриптор файла, который в примере получается с помощью другой библиотечной функции _get_osfhandle.

```
//.....................................................
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());
   }
//.....................................................
```

Рис. 3.13.2. Пример использования функции TransmitFile

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, что 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 считываются 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 отправить 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 ему 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.

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

```
// -- 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
//       должно быть не нулевое значение (например,
//       0x00000001), если же параметр устанавливается
//       в состояние Disabled, то поле optval должно
//       содержать 0x00000000.
```

Рис. 3.15.1. Функция setsockopt

```
//.....................................................
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());
//.....................................................
```

Рис. 3.15.2. Пример применения setsockopt

На рис. 3.15.2 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 (рис. 3.16.1). При этом предполагается, что известно символическое имя компьютера, на котором находится программа сервера.

```
// -- получить адрес хоста по его имени
// Назначение: функция предназначена 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.
```

Рис. 3.16.1. Функция gethostbyname

Такое решение достаточно часто применяется разработчиками распределенных систем. Связав набор программ-серверов с определенными стандартными именами компьютеров, распределенное приложение становится независимым 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 администрат-ора сети.

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

```
// -- получить имя  хоста по его адресу
// Назначение: функция предназначена 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.
```

Рис. 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;
```

Рис. 3.16.3. Структура hostent

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

```
// -- получить имя хоста
// Назначение: функция предназначена 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.
```

Рис. 3.16.4. Функция gethostname

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

3.17. Итоги главы

1. Интерфейс сокетов – это набор специальных функций, входящий в состав операционной системы и предназначеных для доступа прикладных процессов к сетевым ресурсам. Первый 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 позволяет создавать сокеты, устанавливать параметры и режимы их работы, осуществлять пересылку данных между сокетами, преобразовать форматы данных, обрабатывать возникающие ошибки и др.
4. Winsock2 обеспечивает две схемы взаимодействия прикладных процессов: с установкой соединения и без нее. 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. Основное 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 с помощью пакетов

протокола UDP, при этом 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 функций, 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 синхронизации процессов и обмена данными между ними.

Разработчики программного обеспечения могут использовать IPC с помощью предоставляемых операционными системами программных интерфейсов (API). С IPC API 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 ним, – ***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 сооб-щениями. 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 табл. 4.2.1 перечислены основные функции интерфейса Named Pipe. Следует отметить, что функции CreateFile, ReadFile, WriteFile применяются не только для работы с именованными каналами, но и с файловой системой, сокетами и т. д. Поэтому эти универсальные функции часто не указывают в составе Named Pipe API.

Таблица 4.2.1

Функция	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

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.

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 функция CloseHandle Win32 API.

На рисунке 4.2.1 изображены 2 программы, реализующие 2 процесса распределенного приложения. Каждая из программ разбита на 4 блока. Сплошными направленными линиями обозначается

движение данных от одного процесса к другому. Прерывистой линией обозначается синхронизация процессов.

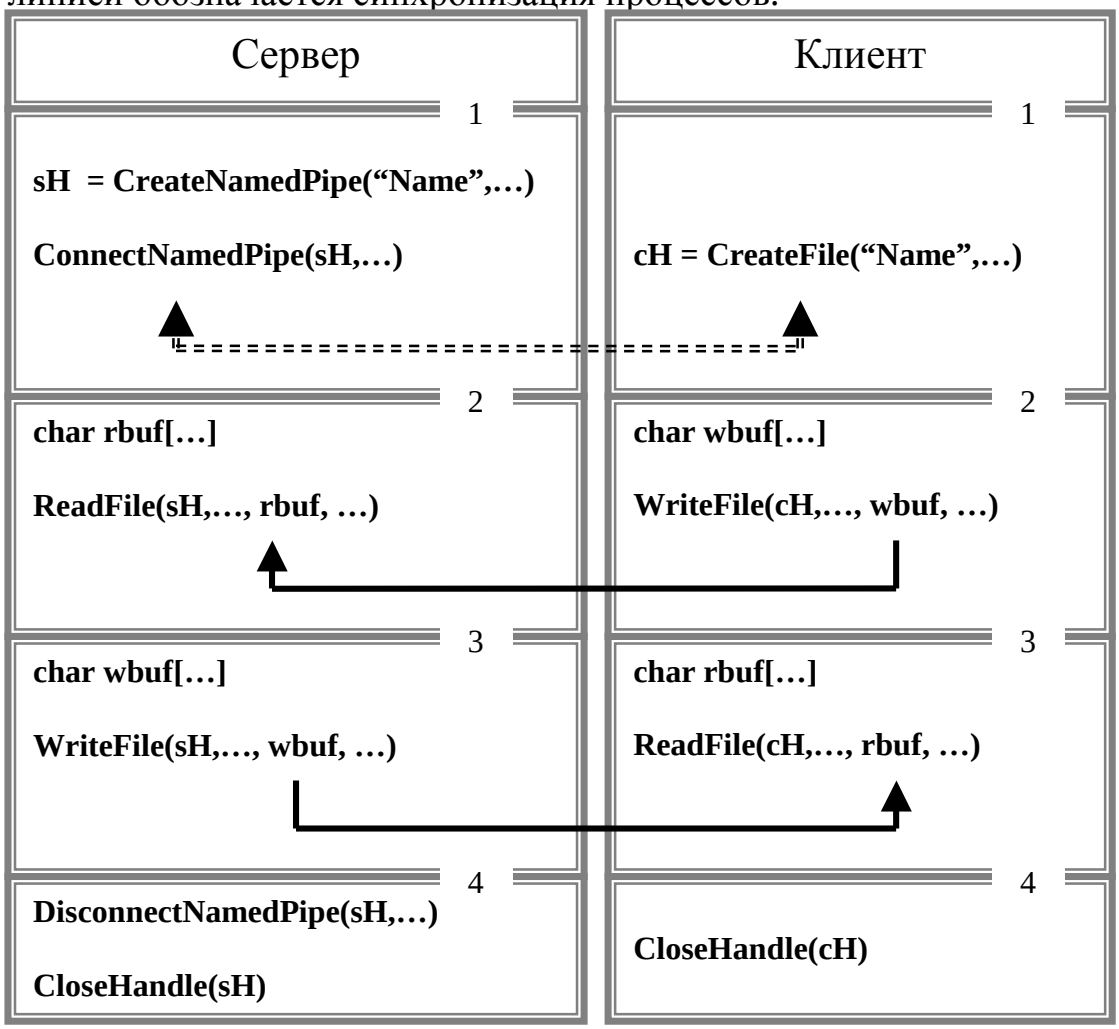

Рис. 4.2.1. Схема взаимодействия процессов, использующих Named Pipe API

В блоке 1 программы сервера выполняются 2 функции: Create-NamedPipe (создать именованный канал) и ConnectNamedPipe (подсоединить сервер к каналу). Одним из параметров функции CreateNamedPipe является имя канала (строка), а результатом ее работы (возвра-
щаемым значением) – дескриптор (HANDEL) канала. Функция ConnectNamedPipe приостанавливает выполнение программы сервера до момента, пока не выполнится функция CreateFile программой клиента.

В блоках 2 и 3 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 блоке 4 программы сервер разрывает соединение с помощью функции DisconnectNamedPipe и закрывает дескриптор именованного канала.

Для программы клиента остается пояснить только блок 1, поскольку всем остальным блокам есть аналогичные в программе сервера. В блоке 1 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 рис. 4.2.1 является общей для этих двух случаев.

В случае завершения функции Named Pipe API 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 с помощью функции CreateNamedPipe (рис. 4.3.1).

Первый параметр функции CreateNamedPipe – указатель на строку имени канала. В зависимости от контекста в функциях используется 2 формата имени канала: локальный (рис. 4.3.2) и сетевой (рис. 4.3.3).

Для связи сервера по одному именованному каналу с несколькими клиентами, он должен создать несколько экземпляров этого канала. Каждый из них 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 книге [4].

```
// -- 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 с 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.
```

Рис. 4.3.1. Функция CreateNamedPipe

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

Рис. 4.3.2. Локальный формат имени канала

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

Рис. 4.3.3. Сетевой формат имени канала

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

```
// -- 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.
```

Рис. 4.3.4. Функция ConnectNamedPipe

На рис. 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;}
//.............................................................
```

Рис. 4.3.5. Фрагмент программы сервера

В программе, приведенной на рис. 4.3.5, для обработки ошибок используется функция SetPipeError. Она аналогична функции SetErrorMsgText, которая использовалась в примерах главы 3 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 (рис. 4.4.1).

```
// -- 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.
```

Рис. 4.4.1. Функция WaitNamedPipe

После обнаружения свободного канала клиент может установить связь с каналом при помощью функции CreateFile (рис. 4.4.2). 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.

```
// -- 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 определяет атрибуты безопасности,
//       значение NULL устанавливает значения атрибутов по умолчанию;
//     - 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.
```

Рис. 4.4.2. Функция CreateFile

При установке в функциях Named Pipe API атрибутов безопасности по умолчанию, как это сделано во всех приведенных здесь примерах, подсоединиться к 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 рис. 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 удваивается.

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.

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 GetLast-Error ERROR_PIPE_CONNECTED. Даже в том случае, если функция WaitNamedPipe обнаружит свободный экземпляр канала и вернет TRUE, 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, демонстриру-ющий подключение к именованному каналу, изображен на рис. 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;}
//.....................................................
```

Рис. 4.4.3. Фрагмент программы клиента

4.5. Обмен данными по именованному каналу

Для обмена данными по именованному каналу используются 3 функции: ReadFile, WriteFile и PeekNamedPipe (рис. 4.5.1, 4.5.2, 4.5.3).

```
// -- читать данные из канала
// Назначение: функция  предназначена для 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.
```

Рис. 4.5.1. Функция ReadFile

```
// -- писать данные в канал
// Назначение: функция  предназначена для 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.
```

Рис. 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 они 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.
```

Рис. 4.5.3. Функция PeekNamedPipe

4.6. Передача транзакций по именованному каналу

Для обмена сообщениями по сети может использоваться функция TansactNamedPipe (рис. 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.

```
// -- 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 предполагается,
//       то следует указать  NULL.
```

Рис. 4.6.1. Функция TransactNamedPipe

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

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. В этом случае речь идет об одиночных эпизодических транзакциях.

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

```
// -- 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.
```

Рис. 4.6.2. Функция CallNamedPipe

Сначала осуществляется установка связи с именованным каналом, имя которого указывается в параметрах функции. При этом именованный канал должен быть открыт в режиме обмена 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.

4.7. Определение состояния и изменение характеристик именованного канала

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

```
// -- 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).
```

Рис. 4.7.1. Функция GetNamedPipeInfo

Функция GetNamedPipeInfo используется для получения информации об атрибутах именованного канала, которые являются статическими и не могу быть изменены. Входными параметрами служат дескриптор и тип канала, о котором предполагается получить информацию. При успешном завершении функция возвращает размеры

буферов ввода и вывода, а также максимальное количество экземпляров данного именованного канала.

Поскольку 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.

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

```
// -- 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.
```

Рис. 4.7.2. Функция GetNamedPipeHandleState

Функция GetNametPipeHandleState используется для получения динамических параметров именованного канала, которые могут быть изменены с помощью функции SetNamedHandleState (рис. 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.
```

Рис. 4.7.3. Функция SetNamedPipeHandleState

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

4.8. Итоги главы

1. Современные операционные системы имеют встроенные механизмы межпроцессорного взаимодействия (IPC), позволяющие создавать распределенные в локальной сети приложения. Для 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, обмена данными по каналу и функции работы с транзакциями.

Глава 5. ИНТЕРФЕЙС MAILSLOT

5.1. Предисловие к главе

В этой главе рассматривается еще один IPC-механизм, поддерживаемый операционной системой Windows и имеющий название Mailslots (почтовый ящик). Так же, 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 табл. 5.2.1. Функции CreateFile, ReadFile, WriteFile являются универсальными и используются также для работы с именованными каналами, файловой системой, сокетами и т. д.

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

Таблица 5.2.1

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

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

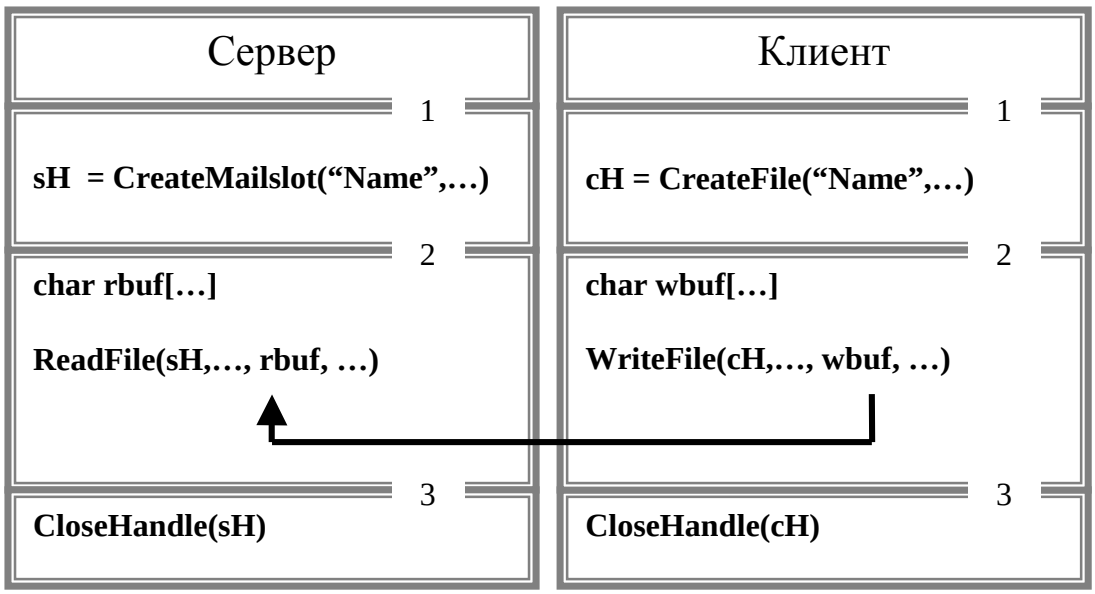

Рис. 5.2.1. Схема взаимодействия процессов,
использующих Mailslot API

В блоке 1 программы сервера выполняется функция CreateMailslot, создающая почтовый ящик. В случае успешного завершения она 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. В блоке 2 сервера осуществляется считывание данных из почтового ящика. В блоке 3 сервера
закрывается дескриптор почтового ящика, что приводит к его уничтожению.

В блоке 1 программы клиента осуществляется подсоединение клиента к почтовому ящику с помощью функции CreateFile (открыть почтовый ящик). В случае успешного выполнения функция формирует дескриптор почтового ящика, который потом используется функцией WriteFile (блок 2 клиента) для записи данных в почтовый сервер. При завершении программы следует закрыть дескриптор почтового ящика с помощью функции CloseHandle.

В принципе, обмен данными между процессами 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 (рис. 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,который устанавливает время ожидания
//       Cобщения  функцией ReadFile; для задания бесконечного
//       ожидания следует установить значение
//       MAILSLOT_WAIT_FOREVER;
//       sattr - предназначен для установки атрибутов
//       безопасности; по умолчанию  следует установить
//       значение NULL.
```

Рис. 5.3.1. Функциия CreateMailslot

Параметр функции CreateMailslot, определяющий имя почтового ящика, подразумевает, что это имя задано в локальном формате. На рис. 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.
```

Рис. 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.
```

Рис. 5.3.3. Сетевой формат имени почтового ящика

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

Рис. 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 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 (рис. 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 источнику [4] или http://msdn2.miscrosoft.com.

```
// -- 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  предназначен для установки атрибутов
//       безопасности; для установки атрибутов, действующих по
//       умолчанию, следует указать 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.
```

Рис. 5.4.1. Функция CreateFile

5.5. Обмен данными через почтовый ящик

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

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 – сервер (записывает в свой собственный ящик) и клиент.

На рис. 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.

```
//.....................................................
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";
//.....................................................
}
//.....................................................
```

Рис. 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";
//.............................................................
}
//.............................................................
```

Рис. 5.5.2. Соединение клиента с почтовым ящиком

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 (рис. 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.
```

Рис. 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 его создании с помощью функции CreateMailslot. В процессе работы иногда необходимо изменить значение этого интервала или вообще сделать его нулевым. Для этого применяется функция SetMailslotInfo (рис. 5.7.1). 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.
```

Рис. 5.7.1. Функция SetMailslotInfo

5.8. Итоги главы

1. Механизм Mailslots (почтовый ящик) является одним из IPC-механизмов операционной системы Windows, позволяющих 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, 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», имеющих более сложную, чем один сервер – один клиент структуру. На рис. 6.1.1 изображена структура распределенного приложения, на создание которого будет ориентировано дальнейшее изложение материала.

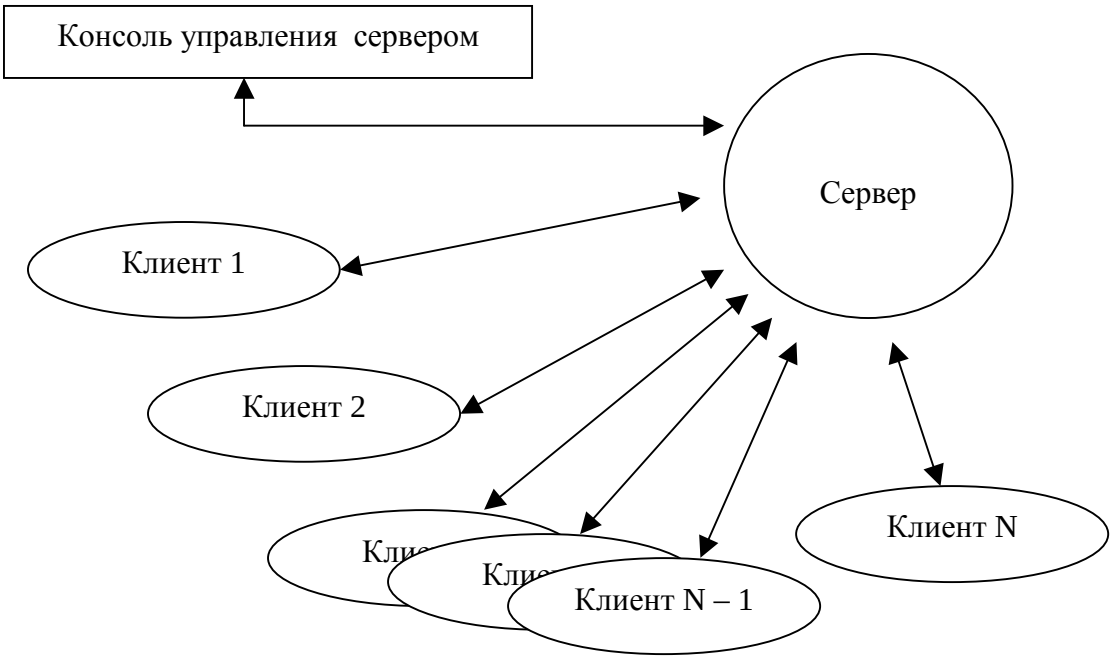

Рис. 6.1.1. Структура распределенного приложения с сервером, обслуживающим одновременно несколько клиентов

Распределенное приложение, изображенное на рис. 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, они 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 они лишь порождают 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 в рамках одного сервера параллельно работают и выполняют специфические функции в рамках сервера несколько процессов.

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, работающей параллельно с другими частями (процессами) независимо от 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, каждый из которых предназначен для обслуживания подключившегося клиента или выполнения каких-то внутренних задач сервера.

Критическим по времени для параллельного сервера является момент подключения клиента. Сервер не должен тратить много времени на это, 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 (желательно, чтобы он имел наивысший приоритет), который бы был занят только подключением клиентов.

С другой стороны, может потребоваться управление процессом подключения. Например, если оператор консоли управления ввел команду, запрещающую подключение новых клиентов к серверу. В этом случае необходимо 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 нецелесообразно. Но параллельному серверу 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. Для управления обслуживающими процессами и сбора статистики требуется динамическая структура данных, позволяющая добавлять и удалять элементы и 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). Решением этой проблемы являются механизмы синхронизации, позволяющие последовательно использовать такие критические ресурсы.

6.3. Структура параллельного сервера

Структура параллельного сервера зависит 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
(рис. 6.3.1), структура которого, по мнению автора, отражает все требующие внимания моменты.

На рис. 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 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 и получает управление от операционной системы. Он создает и запускает новые процессы AcceptServer, ConsolePipe и GarbageCleaner. AcceptServer, в свою очередь, создает несколько процессов с именем EchoServer. Сплошными двунаправленными линиями обозначается перемещение данных между Echo-Server-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 ее серверную часть. Штриховой линией, соединяющей изображение клиентских программ и процесса AcceptServer, обозначается процедура создания соединения между клиентом и сервером.

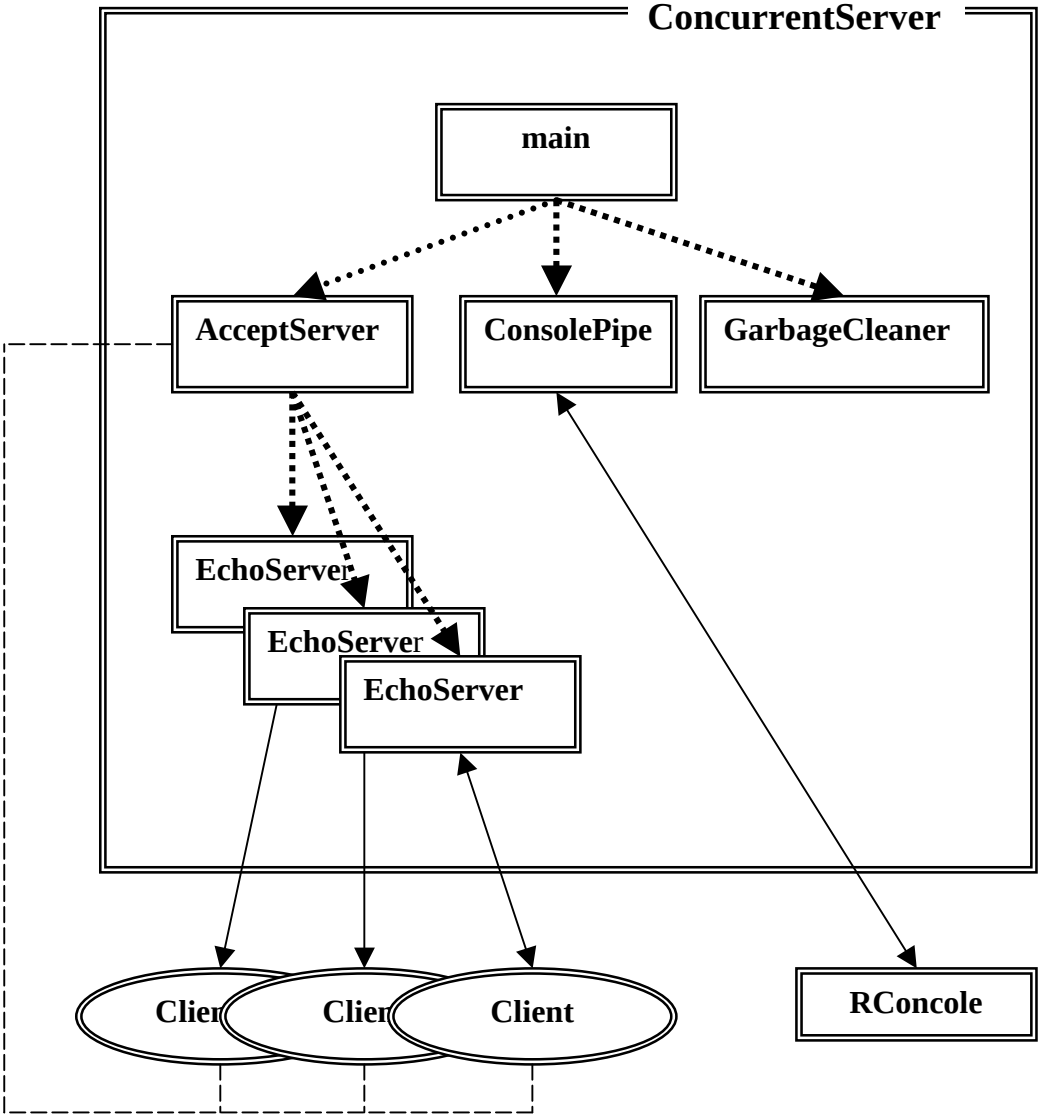

Рис. 6.3.1. Структура параллельного сервера

Опишем назначение компонентов изображенного на рис. 6.3.1 распределенного приложения.

Процесс main предназначен для запуска, инициализации и завершения 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 создается процессом main и предназначен для выполнения процедуры подключения клиентов к серверу, исполнения команд консоли управления, а также 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 создается процессом main и является сервером именованного канала, по которому осуществляется связь между программой RConsole (консоль управления сервером) и параллельным сервером.

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

Процесс EchoServer создается процессом AcceptServer по одному для каждого успешного подключения программы клиента. Основным назначением 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 предназначена для пересылки данных серверу и получения ответа от него. 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 предназначена для ввода команд управления сервером и вывода диагностических сообщений, полученных от сервера. RConsole является клиентом именованного канала.

Список подключений ListContact (не изображен на рисунке) создается на 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 EchoServer (по одному для каждого подключения).

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, который 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 программой могут вызываться функции. При вызове одной и той же функции в разных потоках управления важно, чтобы эта функция была ***безопасносной для потоков***, т. е. обладала свойством ***реентерабельности*** и обеспечивала 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 могут служить изменяемые функцией статические и глобальные переменные.

Каждое приложение, работающее в среде Windows, имеет, по крайней мере, один поток, который называется ***первичным***, или ***главным***. В консольных приложениях этот поток исполняет функцию ***main***, в приложениях с графическим интерфейсом – 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 табл. 6.4.1.

Таблица 6.4.1

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 (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). Она должна завершаться вызовом функции ExitThread с параметром, устанавливающим код возврата. Пример правильного оформления функции потока и именем AcceptServer приведен на рис. 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;
//    - параметр sSt может принимать значение
//      NULL, в этом случае размер стека потока будет
//      установлен по умолчанию (1МБ);
//    - если параметр flags принял значение 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.
```

Рис. 6.4.1. Функция CreateThread

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

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

Рис. 6.4.2. Структура функции потока

На рис. 6.4.3 приводится пример использования функции CreateThread. Здесь main создает три потока с потоковыми функциями AcceptServer, ConsolePipe, GarbageCleaner.

```
 #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;
};
```

Рис. 6.4.3. Пример использования функции CreateThread

Следует обратить внимание, что после запуска потоков перед завершением функции (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 main до завершения работы потока, соответствующего указанному в параметре дескриптору. Отсутствие функций WaitForSingleObject 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.

Другой важный момент в приведенном примере, на который следует обратить внимание, – это применение квалификатора volatile, который указывает компилятору на необходимость размещения переменной cmd в памяти без выполнения 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 с помощью функции TerminateThread (рис. 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.
```

Рис. 6.4.4. Функция TerminateThread

Исполнение каждого потока может быть приостановлено с помощью функции SuspendThread (рис. 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.
```

Рис. 6.4.5. Функция SuspendThread

Для уменьшения текущего значения счетчика приостановок потока используется функция ResumeThread (рис. 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.
```

Рис. 6.4.6. Функция ResumeThread

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

Под ***процессом*** операционной системы Windows понимается объект ядра, которому принадлежат системные ресурсы, используемые исполняемым приложением операционной системы. Выполнение
процесса начинается с первичного потока main, который может создавать новые потоки и новые процессы. С точки зрения техники программирования работа с процессами очень напоминает работу с потоками с одним существенным отличием. Оно 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.

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

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

Рис. 6.4.7. Функция Sleep

Создание и запуск отдельного процесса по затратам ресурсов значительно превосходит затраты на создание потока в рамках существующего процесса. Кроме того, более затратной 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, сделанным с применением механизма потоков.

При разработке параллельных серверов использовать механизмы управления процессами целесообразно в тех случаях, когда в рамках сервера используются такие компоненты (чаще все-
го это обслуживающие процессы), которые могут разрушить работу всего параллельного сервера. В этом случае целесообразно

их реализовать 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 источниках [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 рис. 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.

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 источниках [4, 15]. Здесь будет рассматриваться только механизм критических секций. С остальными способами синхронизации процессов и потоков операционной системы Windows можно ознакомиться в книгах [4, 14].

```
#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 ненормально
    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
```

Рис. 6.5.1. Пример реализации списка подключений ListContact

Критические секции являются 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 табл. 6.5.1. Описание этих функций представлено на рис. 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 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.

 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
                              )
```

Рис. 6.5.2. Функции, реализующие механизм критических секций

Таблица 6.5.1

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

Схема использования механизма критических секций изображена на рис. 6.5.3.

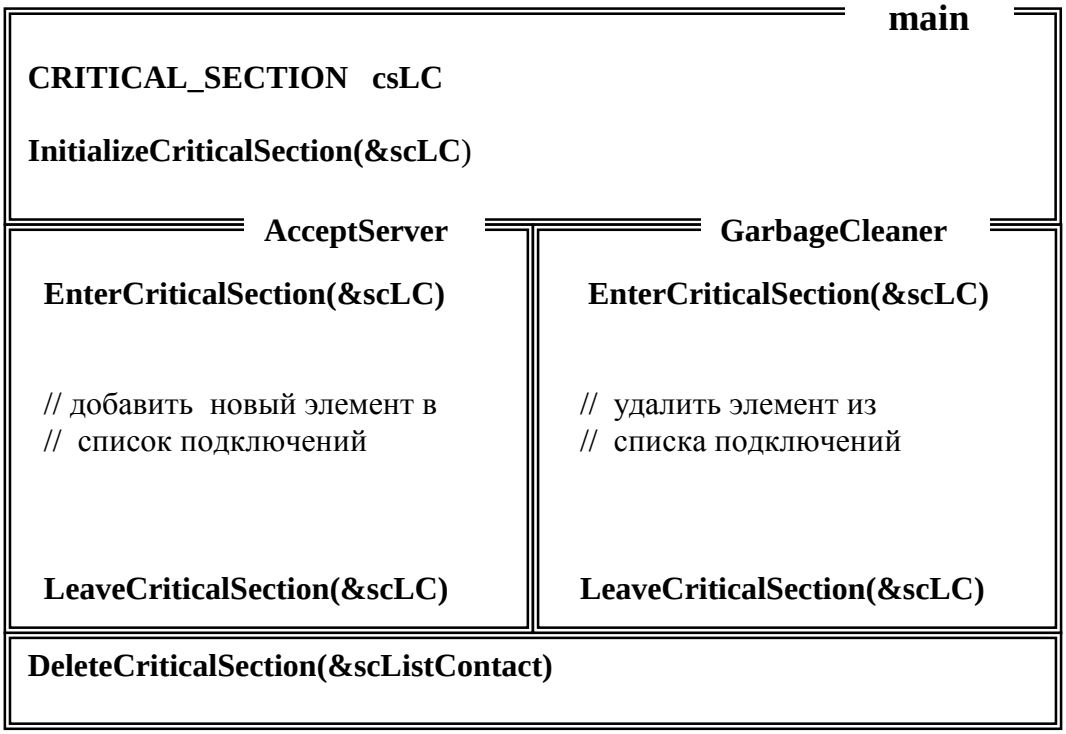

Рис. 6.5.3. Схема применения механизма
критических секций

На рис. 6.5.3 изображены 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 осу-ществлялось 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 согласован 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 ее исполнения необходимо определить асинхронную процедуру, указать поток, в контексте которого она будет выполняться, и дать разрешение на выполнение.

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 и этого момента с ним связь, поэтому 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 табл. 6.6.1.

Таблица 6.6.1

Функция	Назначение
QueueUserAPC	Поставить асинхронную процедуру в очередь
SleepEx	Приостановит поток для выполнения асинхронных процедур

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

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.

<table>
<tr><td colspan="2">ConcurrentServer
HANDEL hAcceptServer; // дескриптор AcceptServer</td></tr>
<tr><td colspan="2">VOID CALLBACK ASStartMessage(…)
{
// вывод на консоль сообщения о начале работы
}</td></tr>
<tr><td colspan="2">VOID CALLBACK ASFinishMessage(…)
{
// вывод на консоль сообщения об окончании работы
}</td></tr>
<tr><td>AccepServer</td><td>EchoServer</td></tr>
<tr><td>// цикл подключения

SleepEx (…, TRUE)

// клиентов и запуска
// серверов обслуживания</td><td>QueueUserAPC(
ASStartMessage,
hAcceptServer, …
)

// обслуживание клиента

QueueUserAPC(
ASFinishMessage,
hAcceptServer, …
)</td></tr>
</table>

Рис. 6.6.1. Схема использования механизма
асинхронного вызова процедур

В начале своей работы функция EchoServer выполняет функцию QueueUserAPC (рис. 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.

```
// -- 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.
```

Рис. 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 в заданный в параметре ms
//       интервал времени;
```

Рис. 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 (рис. 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 (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 него он не может перейти в сигнальное.

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 табл. 6.7.1.

Таблица 6.7.1

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

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

```
// -- 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 ожидающего таймера; в
//       последнем случае функция GetLastError  вернет
//       значение ERROR_ALREADY_EXISTS.
```

Рис. 6.7.1. Функция CreateWaitableTimer

Для перевода ожидающего таймера в активное состояние и его установки динамических параметров предназначена функция SetWaitableTimer (рис. 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 нс (10-7с);
//    - 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
```

Рис. 6.7.2. Функция SetWaitableTimer

Если создан поименованный таймер, то он может быть использован в контексте другого процесса с помощью функции OpenWaitableTimer (рис. 6.7.3). Для перевода таймера в неактивное состояние применяется функция CancelWaitableTimer (рис. 6.7.4). Для ожидания сигнального состояния таймера используется универсальная функция WaitForSingleObject (рис. 6.7.5).

```
// -- 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.
```

Рис. 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.
```

Рис. 6.7.4. Функция CancelWaitableTimer

```
// -- ждать сигнального состояния
// 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.
```

Рис. 6.7.5. Функция WaitForSingleObject

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 дес-крипторов объектов синхронизации и различные режимы ожидания и проверки сигнального состояния. Если кроме этого требуется исполнять асинхронные процедуры, то можно использовать функции WaitForSingleObjectEx и WaitForMultiplee-ObjectsEx.

После перехода ожидающего таймера в сигнальное состояние с помощью функции SetWaitableTimer в очередь асинхронных процедур может быть установлена функция завершения. Она должна иметь такой же прототип, как и у асинхронных процедур.

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 рис. 6.7.6 изображена схема использования ожидающего таймера в модели параллельного сервера ConcurrentServer.

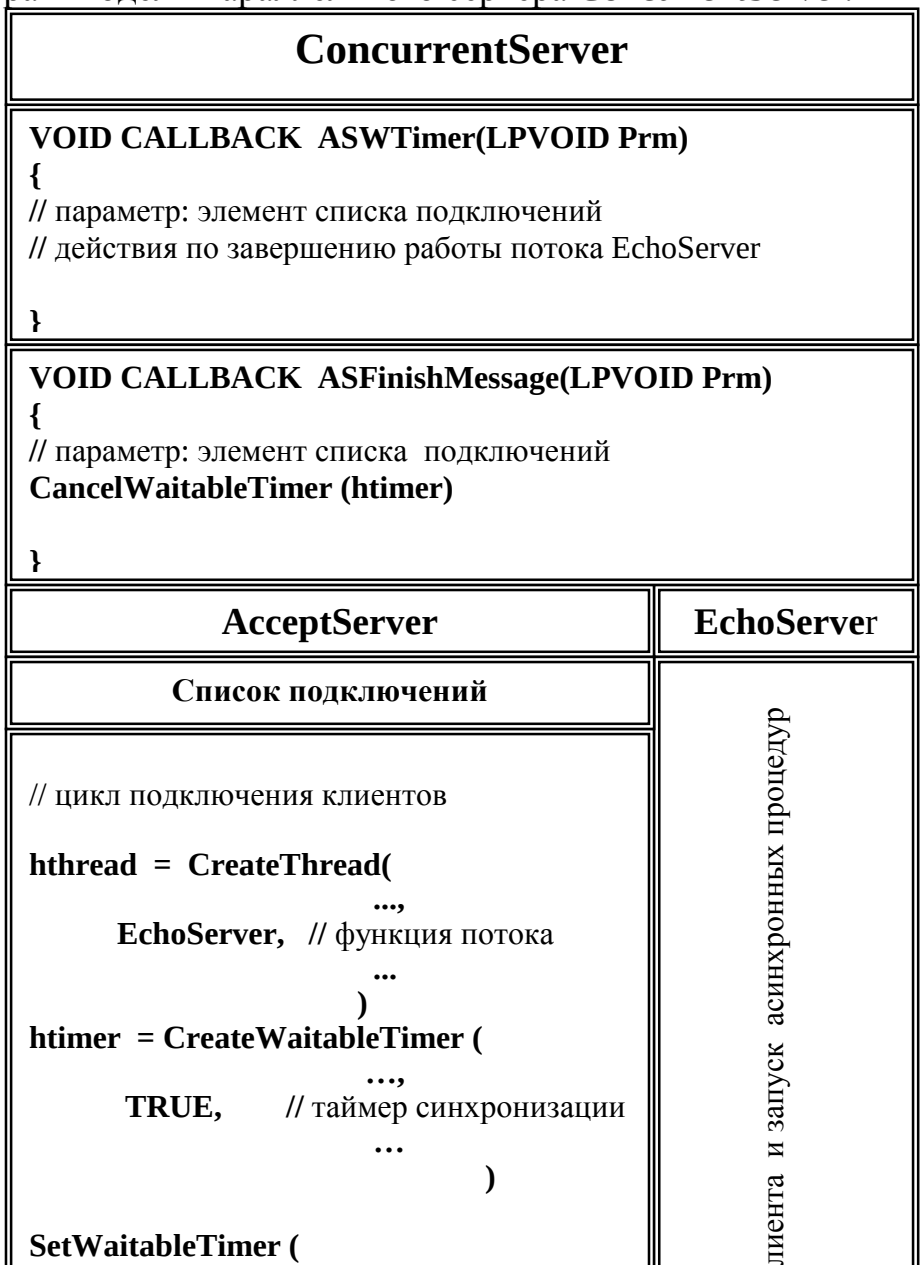

Рис. 6.7.6. Схема использования ожидающего таймера
в модели ConcurrentServer

Следует обратить внимание на новое применение асинхронной процедуры ASFinishMessage, назначение которой рассматривалось выше. Теперь в рамках этой асинхронной функции отменяется ожидающий таймер.

На рис. 6.7.6 обозначен список подключений. Структура элементов этого списка рассматривалась выше. Основное предназначение списка – хранение информации о каждом подключении. В модели сервера ConcurrentServer предполагается хранить дескрипторы обрабатывающего потока и ожидающего таймера в элементах спис-ка 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 ним. 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. Они выполняют несколько элементарных операций, которые объединяются в одну неделимую операцию, называемую ***атомарной***.

В табл. 6.8.1 приведены 4 блокирующие функции, используемые в операционных системах семейства Windows.

Таблица 6.8.1

Функция	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 выравнены на границу слова, т. е. были кратны 4. Для такого выравнивания достаточно, чтобы переменная была объявлена в программе со спецификаторами типов long, unsigned long или LONG, ULONG, DWORD.

Функция InterlockedExchange (рис. 6.8.1) позволят установить (заменить) значение переменной. Если требуется заменить конкретное известное значение, то можно использовать функцию InterlockedCompareExchange (рис. 6.8.2). Функции InterlockedIncrement и InterlockedDecrement (рис. 6.8.3 и 6.8.4) используются для увеличения или уменьшения значения переменной на единицу. Функция InterlockedExchangeAdd (рис. 6.8.5) позволяет увеличить или уменьшить значение переменной на заданную величину.

```
// -- заменить значение
// Назначение: функция предназначена для 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.
```

Рис. 6.8.1. Функция InterlockedExchange

```
// -- сравнить и заменить значение
// Назначение: функция предназначена для 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.
```

Рис. 6.8.2. Функция InterlockedCompareExchange

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

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

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

Рис. 6.8.3. Функция InterlockedIncrement

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

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

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

Рис. 6.8.4. Функция InterlockedDecrement

```
// -- изменить значение
// Назначение: функция предназначена для 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.
```

Рис. 6.8.5. 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, отказывает клиенту в подключении. При такой схеме работы нет параллельных потоков, одновременно изменяющих переменную ClientServiceNumber, т. к. функции Accept-Server, ASStartMessage,
ASFinishMessage работают в одном потоке последовательно. Более того, можно даже предусмотреть команду консоли ConsolePipe управления сервером, позволяющую уменьшать или увеличивать значение MaxClientServiceNumber, что регулирует 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 невозможна, т. к. не исключается несогласованное изменение переменной MaxClientServiceNumber.

Проблема возникнет и в том случае, если 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. Неблокирующий 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 сервера accept 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 будет 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 через сокет. В последнем случае, как и в предыдущем 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.

```
// -- 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.
```

Рис. 6.9.1. Функция ioctlsocket

Функция ioctlsocket (рис. 6.9.1) используется для переключения сокета в режим без блокировки. На рис. 6.9.2 представлен фрагмент функции потока AcceptServer, в котором используется функция ioctlsocket. После успешного выполнения ее вызывается функция CommandsCycle, фрагменты которой приведены на рис. 6.9.3.

```
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
 }
```

Рис. 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
};
};
```

Рис. 6.9.3. Пример обработки команд stop, start и exit

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 команд, применяемых для управления сервером (тип enum).

Функция CommandCycle предназначена для обработки команд управления сервером и подключения клиентов, осуществляемый функцией AcceptCycle (рис. 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.

```
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;
};
```

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

В случае успешного подключения клиента функция AcceptCylcle возвращает значение true. Значение squirt, передаваемое в качестве параметра функции AcceptCylcle, показывает максимальное количество 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.

После каждого вызова функции accept для сокета в режиме без блокировки (рис. 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 его зависит от типа операционной системы Windows, процессора и

приблизительно равна 20 мс. По истечении кванта времени исполнение текущего потока прерывается, его контекст сохраняется и процессор передается потоку с высшим приоритетом.

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

Таблица 6.10.1

Функция	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 с нормальным приоритетом, с высоким приоритетом и реального времени.

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

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

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 ограничена отведенным временем реакции на внешние события.

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

```
// -- 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.
```

Рис. 6.10.1. Функция GetPriorityClass

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 приоритета – 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.

```
// -- 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:
```

Рис. 6.10.2. Функция SetPriorityClass

```
// -- 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.
```

Рис. 6.10.3. Функция GetCurrentProcess

Уровень приоритета потока может быть низший, ниже нормального, нормальный, выше нормального, высший, приоритет фонового потока и приоритет потока реального времени. Значения базовых приоритетов потоков можно определить, воспользовавшись табл. 6.10.2.

Таблица 6.10.2

Приоритет потока	Приоритет процесса					
	реального времени	высокий	выше нормального	нормаль-ный	ниже нормального	фоновый
реального времени	31	15	15	15	15	15
высший	26	15	12	10	8	6
выше	25	14	11	9	7	5

нормальног о						
нормальный	24	13	10	8	6	4
ниже нормальног о	23	12	9	7	5	3
низший	22	11	8	6	4	2
фоновый	16	1	1	1	1	1

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 (рис. 6.10.4). Функции GetThreadPriority (рис. 6.10.5) предназначена для получения приоритета потока.

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

 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.
```

Рис. 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.
```

Рис. 6.10.5. Функция GetThreadPriority

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.

Для управления режимом динамического изменения базового приоритета потока могут быть использованы функции SetProcess-PriorityBoost и SetThreadPriorityBoost (рис. 6.10.6 и 6.10.7).

```
// -- 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.
```

Рис. 6.10.6. Функция SetProcessPriorityBoost

```
// -- 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  e // [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.
```

Рис. 6.10.7. Функция SetThreadPriorityBoost

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.

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

```
// -- определить состояние  режима процесса
// Назначение: функция предназначена для определения состояния
```

```
// -- 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.
```

Рис. 6.10.9. Функция GetThreadPriorityBoost

В модели параллельного сервера ConcurrentServer основная конкуренция за процессорное время происходит между потоком Accept-Server, обрабатывающими потоками EchoServer, потоками GarbageCleaner
и ConsolePipe. Принцип распределения приоритетов между этими потоками зависит от стратегии разработчика. 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 очередь подключений растет, то может 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

неизвестно, какого рода услуга будет запрошена клиентом у сервера.

Обычно сервер должен распознавать некоторое количество команд (запросов), которые клиент может направить в его адрес. Каждая из них идентифицируется своим кодом и может содержать определенный набор операндов. Например, часто первой командой, 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].

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

На рис. 6.11.1 изображена структура параллельного сервера ConcurrentServer, обслуживающего разнотипные запросы. Для простоты здесь не изображены некоторые потоки, которые уже рассматривались ранее. 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 нецелесообразно, т. к. его основное назначение – подключение клиентов.

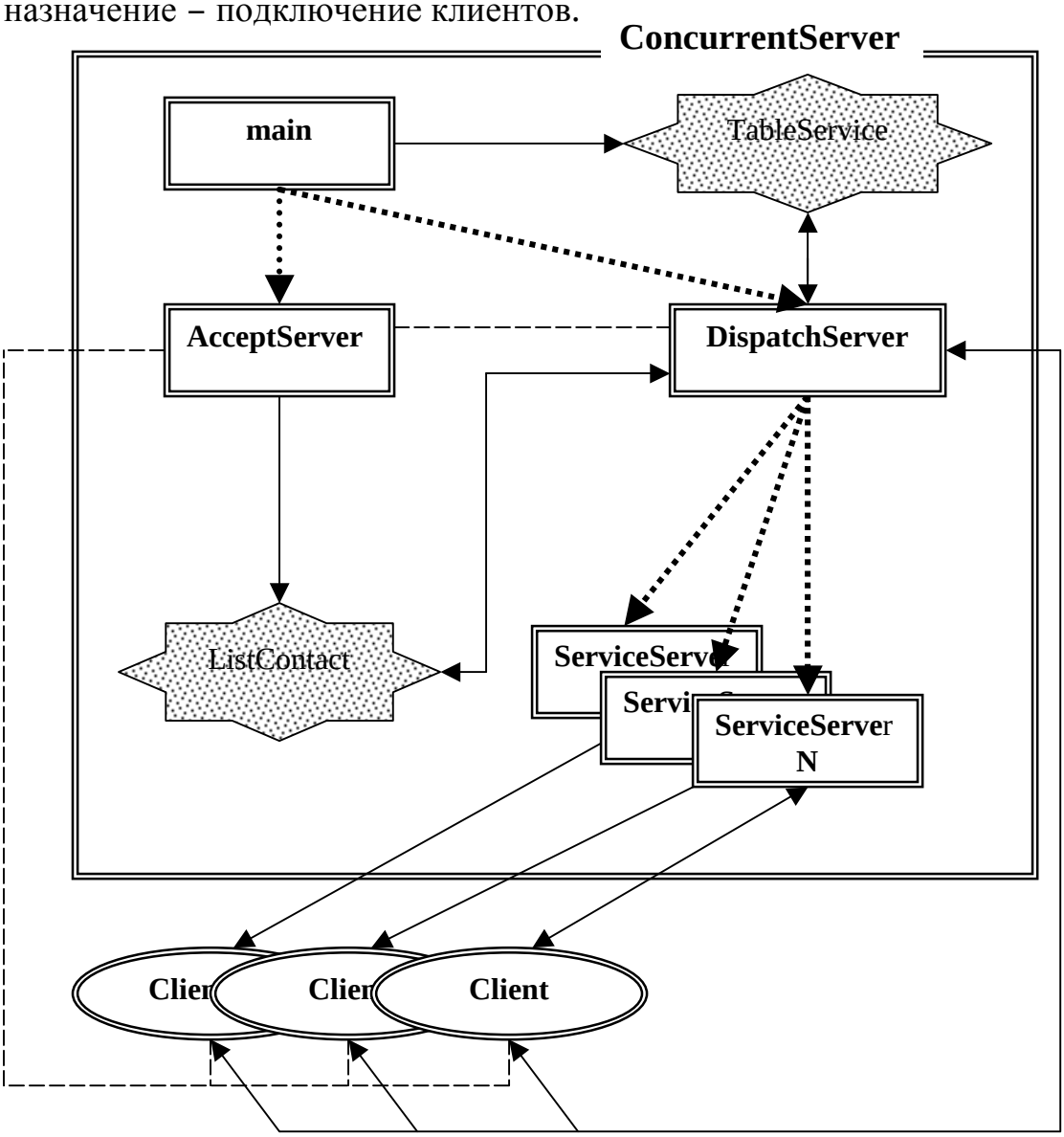

Рис. 6.11.1. Структура параллельного сервера
с диспетчером запросов

На поток DispatchServer возлагается прием первой команды от клиента после подключения, содержащей идентификатор запрашиваемого сервиса. Информация о связи идентификатора и адреса потоковой функции содержится в специальной таблице TableService, которая 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 с помощью ожидающего таймера должен отследить интервал времени 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. 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 табл. 6.12.1.

Таблица 6.12.1

Функция	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 (рис. 6.12.1). С помощью функции OpenEvent (рис. 6.12.2) можно получить дескриптор уже созданного события и установить некоторые его характеристики.

```
// -- 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 несигнальное;
//    - параметр ename задает имя события, что позволяет
//      разным процессам работать с общим событием;
//      если не предполагается использовать имя события, то для
//      этого параметра может быть установлено значение NULL.
```

Рис. 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:- параметр accss  может принимать любую
```

```
// -- перевести событие в сигнальное состояние
// Назначение: функция предназначена для 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.
```

Рис. 6.12.3. Функция SetEvent

```
// -- перевести событие в  несигнальное состояние
// Назначение: функция предназначена для 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.
```

Рис. 6.12.4. Функция ResetEvent

Функция PulseEvent (рис. 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.

```
// -- освободить ожидающие потоки
// Назначение: функция предназначена для 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.
```

Рис. 6.12.5. Функция PulseEvent

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

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).

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-библиотеку. В этом случае сервер может динамически ее загружать.

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 рис. 6.13.1 изображена структура параллельного сервера, использующего динамическую библиотеку ServiceLibrary для хранения таблицы TableService и функций обслуживающих потоков.

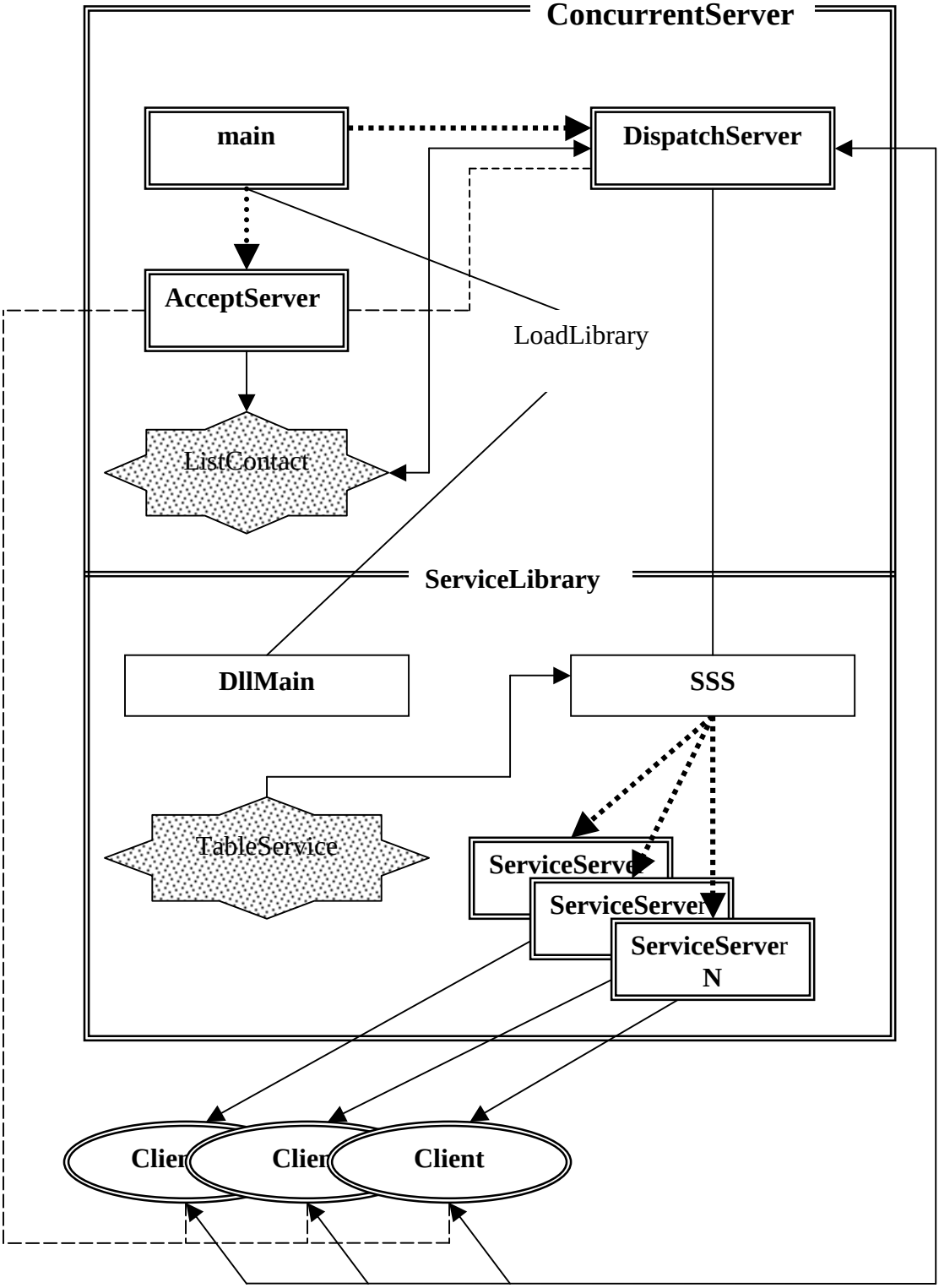

Рис. 6.13.1. Структура параллельного сервера
с динамической библиотекой

Библиотека загружается в память при инициализации сервера. В ее состав и входит стандартная функция DllMain (рис. 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,
//      функция 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 адресу 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.
```

Рис. 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 книге [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.

Некоторые функции, необходимые для работы с динамически подключаемыми библиотеками, приведены табл. 6.13.1. С полным перечнем функций можно ознакомиться в [4, 12, 14].

Таблица 6.13.1

Функция	Назначение
FreeLibrary	Отключить DLL-библиотеку от процесса
GetProcAddress	Импортировать функцию
LoadLibrary	Загрузить DLL-библиотеку

Функция LoadLibrary (рис. 6.13.3) применяется для загрузки динамической библиотеки в память компьютера. Для выгрузки библиотеки используется функция FreeLibrary (рис. 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, предоставляющий каждому клиенту библиотеки (так принято называть приложения, использующие функции DLL-библиотеки) полную независимость. При создании общей разделяемой 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 извне, говорят, что они экспортируются библиотекой. Когда рассматривают эти же функции со стороны клиента DLL-библиотеки, то говорят об ***импорте функций***. Для этого применяется функция GetProcAddress (рис. 6.13.5). Следует отметить, что импортировать можно не только функции, но и 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.
```

Рис. 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.
```

Рис. 6.13.4. Функция FreeLibrary

```
// -- импортировать функцию
// Назначение: функция предназначена для 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.
```

Рис. 6.13.5. Функция GetProcAddress

Экспортируемая DLL-библиотекой функция должна быть специальным образом оформлена: иметь прототип, изображенный на рис. 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.

```
// -- прототип экспортируемой DLL-библиотекой функции

extern “C” __declspec(dllexport) Funcname (.....);
```

Рис. 6.13.6. Прототип экспортируемой функции

На рис. 6.13.7 приводится текст программы простейшего DLL-модуля.

```
#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;
}
```

Рис. 6.13.7. Пример построения TableService и функции SSS

Главная функция DllMain имеет только одно назначение – возвращение значение TRUE. За время использования сервером DLL-библиотеки функция DllMain вызывается 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 (рис. 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)
```

Рис. 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 во фрагменте программы на рис. 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);
//.....................................................
```

Рис. 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. 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 учетной 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 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 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 и 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 Windows 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. Кроме того, есть API, позволяющий использовать все это.

В простейшем случае можно в рамках операционной системы Windows создать группу администраторов и группу пользователей сервера. Заполнить их именами пользователей, которым разрешается доступ к соответствующим ресурсам сервера. Пусть наименования 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 книге [4].

6.15. Итоги главы

1. По принципу работы различают два типа серверов: итеративные и параллельные.
2. Итеративные серверы, как правило, используют для связи с клиентами протокол UDP и применяются в тех случаях, когда предполагается, что запросы клиентов являются редкими, а исполнение 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 компонентов сервера. С точки зрения программирования проще конечно, применять потоки, т. к. в этом случае разделяется общая 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 т. 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 неблокирующий 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 их 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 таблицей 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.

Глава 7. ПРАКТИКУМ

7.1. Предисловие к главе

Для выполнения практических заданий, предлагаемых в этой главе, необходимо иметь несколько объединенных TCP/IP-сетью компьютеров с операционной системой Windows XP. Кроме того, для разработки приложений на языке C++ требуется среда разработки Microsoft Visual Studio не ниже седьмой версии и доступ к соответствующей версии MSDN.

Каждая практическая работа состоит из нескольких заданий. Задания, как правило, связаны между собой и требуют последовательного выполнения. Практическая работа считается выполненной, если успешно выполнены все ее задания.

Все разрабатываемые приложения должны компилироваться в режиме Debug (отладчика) для возможности исследования процесса работы приложений.

7.2. Практическая работа № 1
Сетевые утилиты

7.2.1. Цель и задачи работы

Целью работы является ознакомление с функциональными возможностями сетевых утилит операционной системы Windows.

В результате работы студент научится определять характеристики TCP/IP-сети, тестировать соединения компьютеров в сети, использовать сетевые утилиты при отладке приложений.

7.2.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в главе 2 этого пособия. В качестве дополнительной литературы рекомендуются источники [5, 6, 7, 10].

7.2.3. Утилита ipconfig

Задание 1. Получите справку о параметрах утилиты **ipconfig**.

Задание 2. Получите короткий отчет утилиты, исследуйте его.

Задание 3. Получите полный отчет утилиты. Выпишите символическое имя хоста, IP-адрес, маску подсети, MAC-адрес адаптера.

Задание 4. Определите, к какому классу адресов относится выписанный IP-адрес; вычислите максимальное количество хостов, которое может быть в подсети и укажите диапазон их адресов; определите код производителя сетевого адаптера.

7.2.4. Утилита hostname

Задание 5. Определите имя NetBIOS-имя компьютера с помощью утилиты **hostname**. Сравните его с именем, полученным с помощью утилиты **ipconfig**.

7.2.5. Утилита ping

Задание 6. Получите справку о параметрах утилиты **ping**.

Задание 7. С помощью **ping** проверьте работоспособность интерфейса внутренней петли компьютера.

Задание 8. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров его IP-адрес.

Задание 9. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров символическое имя хоста.

Задание 10. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров символическое имя хоста и увеличив размер буфера отправки до 1000 байт.

Задание 11. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров его IP-адрес и установив количество отправляемых запросов, равное 17.

Примечание. Обратите внимание на значение TTL, которое выдается в отчетах утилиты **ping**. Первоначальное значение TTL (Time To Live – «время жизни») по умолчанию равно 128. Это значение записывается в заголовок каждой дейтаграммы и уменьшается на единицу после прохождения каждого маршрутизатора.

Если в процессе движения дейтаграммы в сети значение TLL уменьшится до нуля, то она уничтожается. Такой подход

гарантирует от зацикливания дейтаграмм в сети. С помощью ключа i утилиты ping можно на период проверки значение TTL изменить.

7.2.6. Утилита tracert

Задание 12. Получите справку о параметрах утилиты **tracert**.

Задание 13. С помощью утилиты **tracert** определите маршрут хоста самого к себе (интерфейс внутренней петли).

Задание 14. С помощью утилиты **tracert** определите маршрут к хосту в локальной сети. Определите количество прыжков в полученном маршруте.

7.2.7. Утилита route

Задание 15. Получите справку о параметрах утилиты **route**.

Задание 16. Выведите на экран монитора таблицу активных маршрутов компьютера. Исследуйте полученный отчет. Определите строки таблицы, соответствующие интерфейсу внутренней петли и широковещательным адресам. Определите IP- адреса шлюзов.

7.2.8. Утилита arp

Задание 17. Получите справку о параметрах утилиты **arp**.

Задание 18. Выведите на экран монитора arp-таблицу. Исследуйте полученный отчет. Определите хосты, которым соответствуют строки arp-таблицы. Определите IP-адрес, которого нет в arp-таблице, но есть в локальной сети. Выполните утилиту **ping** в адрес этого хоста. Выведите снова arp-таблицу и объясните произошедшие изменения. Определите MAC-адреса двух хостов с ближайшими IP-адресами.

7.2.9. Утилита nslookup

Задание 19. Запустите утилиту **nslookup** в диалоговом режиме и наберите команду **help**. Ознакомьтесь с полученным отчетом, отражающим возможности утилиты **nslookup**.

Задание 20. Запустите утилиту **nslookup** в диалоговом режиме. Определите имя и IP-адрес хоста, на котором установлен DNS-сервер по умолчанию. Определите IP-адреса хостов по их именам (имена хостов выдаст преподаватель).

7.2.10. Утилита netstat

Задание 21. Получите справку о параметрах утилиты **netstat**.

Задание 22. Запустите утилиту **netstat -a** для отображения всех подключений и ожидающих портов. Исследуйте отчет. Выясните, какие из известных служб прослушивают порты. С какими из этих портов поддерживается внешнее соединение и по какому протоколу? Определите имена хостов и номера портов внешних соединений.

Задание 23. Запустите утилиту **netstat -b** для отображения исполняемых файлов, участвующих в создании подключений. Определите исполняемые файлы служб, прослушивающих порты, идентификаторы процессов операционной системы.

Задание 24. Запустите утилиту **netstat -ab**. Исследуйте полученный отчет. Для формирования файла отчета утилиты перенаправьте вывод утилиты в файл с помощью команды: **netstat -ab > c:\report.txt.** Проконтролируйте наличие отчета в файле.

7.2.11. Утилита nbstat

Задание 25. Получите справку о параметрах утилиты **nbtstat**. Выполните все команды, отраженные в справке. Исследуйте полученные отчеты.

7.2.12. Утилита net

Задание 26. Получите справку о параметрах утилиты **net**. Получите справку по отдельным командам утилиты с помощью команды **help**. Получите статистику рабочей станции и сервера компьютера
с помощью команды **statistics**. Перешлите сообщение на соседний компьютер с помощью команды **send**. Получите список пользователей компьютера с помощью команды **user**.

7.3. Практическая работа № 2
Обмен данными по TCP-соединению

7.3.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков разработки простейшего распределенного приложения архитектуры «клиент – сервер», осуществляющего обмен данными в локальной сети через Windows Sockets TCP-соединение.

Результатом практической работы является разработанное распределенное приложение со схемой взаимодействия процессов, описанной в разделе 3.4 и изображенной на рис. 3.4.2 пособия.

7.3.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в разделах 3.2–3.11, 3.14 пособия.

7.3.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой сервера, изображенной на рис. 3.4.2 пособия. Создайте с помощью Visual Studio консольное приложение **ServerT** (наименование проекта), которое будет использовано для построения серверной части приложения. Включите необходимые директивы компилятора, указанные в разделе 3.2 пособия для подключения динамической библиотеки **WS2_32.LIB**. Откомпилируйте приложение, убедитесь в отсутствии ошибок.

Задание 2. В рамках приложения **ServerT**, созданного в задании 1, разработайте функцию **SetErrorMsgText**, предназначенную для обработки стандартных ошибок библиотеки **WS2_32.LIB**. Предполагается, что функция **SetErrorMsgText** будет использоваться в операторе **throw** для генерации исключения при возникновении ошибок в функциях интерфейса Winsock2. Для получения кода ошибки функций Winsock2 примените функцию **WSAGetLastError**, описание которой приводится в разделе 3.3 пособия. Там же приводится полный список кодов возврата функции **WSAGetLastError** и пример ее использования.

Задание 3. Доработайте приложение **ServerT** таким образом, чтобы оно только инициализировало библиотеку **WS2_32.LIB** и завешало работу с ней. Для этого используйте функции **WSAStartup** и **WSACleanup**, описанные в разделах 3.5 и 3.6 пособия. Обработку ошибок осуществите с помощью конструкции **try-catch** и функции **SetErrorMsgText**, разработанной в задании 2. Используйте пример программы, приведенный в разделе 3.6 пособия. Убедитесь в работоспособности приложения.

Задание 4. Доработайте приложение **ServerT** таким образом, чтобы оно создавало и закрывало сокет, предназначенный для ориентированного на поток соединения. Для этого используйте функции **socket** и **closesocket**, описанные в разделе 3.8. Обратите внимание на параметр **type** функции **socket**, указывающий тип соединения. Воспользуйтесь примером из раздела 3.7 пособия. Убедитесь в работоспособности приложения.

Задание 5. Добавьте в приложение **ServerT** вызов функций **bind** и **listen** для установки параметров сокета и перевода его в режим прослушивания. Функция **bind** описана в разделе 3.8, а функция **listen** – в разделе 3.9. Используйте порт **2000** в качестве параметра сокета. Установка параметров сокета осуществляется с помощью структуры **SOCKADDR_IN**. Описание этой структуры приводится в разделе 3.8 пособия. Сверьте схему полученной программы со схемой сервера, изображенной на рис. 3.4.2. Выполните приложение, убедитесь в его работоспособности.

Задание 6. Добавьте в приложение **ServerT** вызов функции **accept**, описание которой приводится в разделе 3.10 пособия. Следует обратить внимание на то, что: 1) успешным результатом работы функции **accept** является новый сокет; 2) первым параметром функции **accept** является уже созданный ранее сокет; 3) второй параметр функции **accept** – указатель на структуру **SOCKADDR_IN** (не надо ее путать с уже применяемой выше), предназначенную для приема параметров подключившегося сокета со стороны клиента сокета. Запустите приложение в режиме отладки (Debug) и убедитесь, что после выполнения функции **accept**, программа переходит в режим ожидания (зависает). Завершите приложение. Сохраните программу **ServerT** для дальнейшего применения.

7.3.4. Разработка клиентской части распределенного приложения

Задание 7. Ознакомьтесь со схемой клиента, изображенной на рис. 3.4.2 пособия. Создайте с помощью Visual Studio новое консольное приложение **ClientT** (наименование проекта), которое будет использовано для построения клиентской части приложения. Повторите все те же действия для этого приложения, которые были сделаны в заданиях 2–4. Убедитесь в работоспособности приложения **ClientT**.

Задание 8. Добавьте в приложение **ClientT** вызов функции **connect**. Описание функции и примера ее использования приводится в разделе 3.10 пособия. Следует обратить внимание на следующее: 1) параметры сокета сервера устанавливаются в структуре **SOCKADDR_IN**; 2) для номера порта необходимо установить значение **2000** (такой же номер, что установлен при параметризации сокета сервера в задании 5; 3) для установки номера порта используются специальные функции, описание которых

приводится в разделе 3.8. Используйте в качестве IP-адреса собственный адрес компьютера **127.0.0.1** (интерфейс внутренней петли) – это даст возможность отладки приложения на одном компьютере. Запустите приложение на выполнение. Убедитесь, что функция **connect** завершилась с ошибкой и обработка ошибок осуществляется корректно. Найдите полученный код ошибки в табл. 3.3.1 пособия и проанализируйте его.

7.3.5. Обмен данными между сервером и клиентом

Задание 9. Запустите на выполнение приложение **ServerT** и убедитесь, что оно приостановилось на вызове функции **accept**. Запустите на выполнение на этом же компьютере (используется интерфейс внутренней петли) приложение **ClientT**. Убедитесь, что сервер **ServerT** вышел из состояния ожидания, а клиент **ClientT** завершился без ошибок.

Задание 10. Доработайте программу сервера **ServerT** таким образом, чтобы после подсоединения клиента на экран консоли **ServerT** выводился IP-адрес и порт подсоединившегося клиента. Необходимые значения находятся в структуре **SOCKADDR_IN**, которая заполняется функцией **accept**. Используйте функции **htons** и **inet_ntoa**, описанные в разделе 3.8. Убедитесь в работоспособности распределенного приложения **ClientT-ServerT**.

Задание 11. Добавьте в программу сервера **ServerT** вызов функции **recv**, а в программу клиента вызов функции **send**. Описание этих функций приводится в разделе 3.11 пособия. Перешлите текст ***Hello from Client*** от клиента серверу. Выведите полученный сервером текст на экран консоли. Обратите внимание на то, что команда **recv** в программе сервера использует сокет, созданный функцией **accept**, а не созданный ранее с помощью функции **socket**.

Задание 12. Установите программу **ServerT** на другой компьютер локальной сети, а в программу **ClientT** внесите необходимые изменения, позволяющие ей установить связь с сервером. Убедитесь в работоспособности распределенного в локальной сети приложения **ClientT-ServerT**.

Задание 13. Внесите изменения в программы **ClientT** и **ServerT**, позволяющие 1000 раз передать сообщение типа ***Hello from Client xxx*** (***xxx*** – номер сообщения) от клиента серверу. Убедитесь в работоспособности в сети.

Задание 14. Доработайте программы **ClientT** и **ServerT** таким образом, чтобы программа **ServerT** принимала последовательность сообщений вида ***Hello from Client xxx*** от программы **ClientT** и отправляла их без изменения обратно программе **ClientT**. Программа **ClientT**, получив вернувшееся сообщение, должна увеличить в нем счетчик ***xxx*** на единицу и вновь направить в адрес **ServerT**. Количество передаваемых сообщений введите через консоль программы **ClientT**. Условием окончания работы для программы **ServerT** является получение сообщения нулевой длины. Оцените время обмена 1000 сообщениями (с помощью функций **clock**) между **ClientT** и **ServerT** через локальную сеть. Запустите утилиту **netstat** на компьютерах клиента и сервера, проанали-зируйте отчет и найдите информацию о приложении **ClientT- ServerT**.

Задание 15. Доработайте программу **ServerT** таким образом, чтобы после отключения клиента она могла снова установить соединение с другим клиентом и продолжила свою работу.

7.4. Практическая работа № 3
Обмен данными без установки соединения

7.4.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков разработки простейшего распределенного приложения архитектуры «клиент – сервер», осуществляющего обмен данными в локальной сети через Windows Sockets без установки соединения (с помощью UDP-сообщений).

Результатом практической работы является разработанное распределенное приложение со схемой взаимодействия процессов, описанной в разделе 3.4 и изображенной на рис. 3.4.1 пособия.

7.4.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в разделах 3.2–3.12, 3.14 пособия.

7.4.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой взаимодействия процессов без установки соединения в распределенном приложении, приведенной в разделе 3.4 пособия (рис. 3.4.1). Определите основные отличия этой схемы от схемы взаимодействия процессов с установкой соединения.

Разработайте программу **ServerU**, реализующую блоки 1, 2 и 5 схемы сервера, изображенной на рис. 3.4.1. Подключите функции обработки ошибок, разработанные в практической работе № 2 (с применением команд структурной обработки ошибок **try-throw-catch**). Обратите внимание на параметр **type** функции **socket**, на отсутствие функций **listen** и **accept**, которые применялись в приложении с соединением. Убедитесь, что
разработанная программа выполняет все функции Winsock2 без ошибок.

Примечание. При разработке программ в заданиях этой практической работы рекомендуется использовать тексты программ, разработанных в предыдущей практической работе.

Задание 2. Реализуйте в программе **ServerU** блок 3 схемы сервера, изображенной на рис. 3.4.1. Используемая в блоке функция **recvfrom** описана в разделе 3.12 пособия. Установите номер серверного сокета равным **2000**. Убедитесь, что при запуске программа **ServerU** приостанавливает свое выполнение (переходит в состояние ожидания) сразу после вызова функции **recvfrom**. Завершите программу.

7.4.4. Разработка клиентской части распределенного приложения

Задание 3. Создайте новое C++-приложение с именем **ClientU**. Реализуйте блоки 1, 2, 3 и 5 схемы клиента, изображенной на рис. 3.4.1. Подключите функции обработки ошибок, разработанные в практической работе № 2. В параметре **to** команды **sendto** (раздел 3.12) установите адрес структуры **SOCKADDR_IN**, содержащей IP-адрес **127.0.0.1** и номер порта **2000**. Обеспечьте пересылку сообщения ***Hello from ClientU***. Запустите на выполнение программу **ClientU** при отсутствующем сервере. Проанализируйте полученный код возврата.

7.4.5. Обмен данными между сервером и клиентом

Задание 4. Запустите на выполнение программу **ServerU** и убедитесь, что она приостановила свое выполнение. Запустите на этом же компьютере программу **ClientU** и убедитесь, что программа сервера получила сообщение и завершилась нормально.

Задание 5. Реализуйте блоки 4 в обеих программах. Перешлите полученное сервером сообщение обратно в адрес клиента и убедитесь, что сообщение получено.

Задание 6. Внесите необходимые изменения в программу **ClientU** для того, чтобы программы можно было расположить на разных

компьютерах локальной сети. Убедитесь в работоспособности приложения.

Задание 7. Реализуйте последовательную пересылку данных от клиента к серверу и обратно по тому же принципу, как это было сделано в заданиях 13, 14 практической работы № 2. Проведите измерения, аналогичные оценке скорости передачи, сравните результаты.

Задание 8. Запустите сервер **ServerU** на одном из компьютеров и одновременно два клиента на двух других компьютерах локальной сети. Оцените количество сообщений, которые успел передать и получить каждый из клиентов.

Задание 9. Запустите сервер **ServerT,** разработанный в практической работе № 2, и программу клиента **ClientU**. Объясните полученный результат.

Задание 10. Запустите сервер **ServerU** и клиент **ClientT,** разработанный в практической работе № 2. Объясните полученный результат.

7.5. Практическая работа № 4
Применение широковещательных IP-адресов

7.5.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков использования широковещательных адресов распределенными в локальной сети приложениями.

Результатом практической работы является разработанное распределенное приложение, использующее широковещательные адреса.

7.5.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в разделах 3.2–3.12, 3.15 пособия.

7.5.3. Разработка серверной части распределенного приложения

Задание 1. Разработайте функцию **GetRequestFromClient**, описание которой представлено на рис. 7.5.1. Функция предназначена для ожидания запроса клиентской программы. Предполагается, что правильный запрос (позывной сервера) состоит из набора символов, который указывается функции в

качестве параметра **name**. Ожидание запроса в функции **GetRequestFromClient** осуществляется с помощью функции **recvfrom**. Если поступившее сообщение является позывным сервера, то функция заполняет возвращаемую структуру **SOCKADDR_IN** (параметры **from** и **flen** функции) и завершается с кодом возврата **true**. Если поступившее сообщение не является позывным сервера, то оно игнорируется, и функция вновь переходит в состояние ожидания. Если функция **recvfrom** завершается аварийно с кодом **WSAETIMEDOUT** (табл. 3.3.1), то функция **GetRequestFromClient** должна завершиться с кодом возврата **false**. Любой другой аварийный код завершения должен приводить к исключительной ситуации (оператор **throw**), соответствующей функциям обработки ошибок, разработанных в практическом занятии № 2.

```
// -- обработать запрос клиента
// Назначение: функция предназначена для обработки запроса
//      клиентской программы.

bool  GetRequestFromClient(
          char*            name, //[in] позывной сервера
          short            port, //[in] номер просушиваемого порта
          struct sockaddr* from, //[out] указатель на SOCKADDR_IN
          int*             flen  //[out] указатель на размер from
                           )

// Код возврата: в случае если пришел запрос клиента, то
//       функция возвращает значение true, иначе возвращается
//       значение false.
// Примечание: параметр name - строка, заканчивающаяся 0x00 и
//       содержащая позывной сервера (набор символов,
//       получаемый сервером от клиента и интерпретируемый
//       как запрос на установку соединения);
//       параметр from содержит указатель структуры,
//       содержащей параметры сокета клиента, приславшего
//       запрос.
```

Рис. 7.5.1. Описание функции GetRequestFromClient

Создайте новое приложение **ServerB**, вызывающее функцию **GetRequestFromClient**. Пусть позывной сервера будет ***Hello***. Запустите приложение **ServerB** и убедитесь, что программа перешла в состояние ожидания. Запустите приложение **ClientU**, разработанное в практической работе № 3. Убедитесь, что **ServerB** не реагирует на ошибочный позывной. Исправьте в приложении

ClientU посылаемую строку на ***Hello*** и убедитесь, что **ServerB** реагирует на правильный позывной.

Примечание. При разработке функции **GetRequestFromClient** и других функций в этом практическом задании вызов функций **WSAStartup** и **WSACleanup** целесообразно осуществлять вне разрабатываемых функций.

Задание 2. Разработайте функцию **PutAnswerToClient**, описание которой приводится на рис. 7.5.2. Функция предназначена для подтверждения сервером запроса клиента на установку соединения. Функция отправляет в адрес клиента (параметры сокета клиента указываются в параметре **to**) свой позывной, что предполагает готовность сервера к дальнейшей работе с клиентом. Предполагается, что функция будет использоваться после завершения функции **GetRequestFromClient**. Внесите изменения в программу **ServerB**, чтобы сервер смог отвечать с помощью функции **PutAnswerToClient** на правильный позывной, полученный от клиента. Проверьте правильность работы сервера **ServerB** с помощью программы **ClientU**.

```
// -- ответить на запрос клиента
// Назначение: функция предназначена для пересылки позывного
//      сервера программе клиента.

bool  PutAnswerToClient(
          char*            name, //[in] позывной сервера
          struct sockaddr* to,   //[in] указатель на SOCKADDR_IN
          int*             lto   //[in] указатель на размер from
                        )

// Код возврата: в случае успешного завершения функция
//        возвращает значение true, иначе возвращается
//        значение false.
// Примечание: параметр name - строка, заканчивающаяся 0x00 и
//        содержащая позывной сервера (набор символов,
//        получаемый сервером от клиента и интерпретируемый
//        как запрос на установку соединения);
//        параметр to  содержит указатель структуры,
//        содержащей параметры сокета клиента.
```

Рис. 7.5.2. Описание функции PutAnswerToClient

Задание 3. Внесите изменения в программу **ServerB** таким образом, чтобы сервер отвечал на многократные запросы от разных

клиентов (необходимо построить цикл с функциями **GetRequestFromClient** и **PutAnswerToClient**). Проверьте работоспособность сервера.

7.5.4. Разработка клиентской части распределенного приложения

Задание 4. Разработайте функцию **GetServer**, описание которой приводится на рис. 7.5.3. Функция предназначена для отправки широковещательного запроса в локальную сеть (всем компьютерам сегмента локальной сети) с позывным сервера. Предполагается, что на одном или на нескольких компьютерах сети есть сервер **ServerB**, который просушивает порт с номером, указанным в параметре **port** функции **GetServer**. Для отправки широковещательного запроса функция **GetServer** должна использовать широковещательный IP-адрес (раздел 3.15 пособия). Использование широковещательного IP-адреса требует специального режима работы сокета, который устанавливается с помощью функции **setsockopt**, входящей в состав Winsock2. Описание этой функции и пример ее использования приводятся в разделе 3.15 пособия. После отправки широковещательного запроса с помощью функции **sendto** функция **GetServer** должна вызвать функцию **recvfrom** для ожидания отклика сервера. При правильном отклике (отклик должен совпадать с позывным) функция формирует структуру **SOCKADDR_IN** с параметрами сокета сервера, возвращает значение **true** и завешается. Если сообщение в адрес клиента приходит, но отклик не содержит правильный позывной или **recvfrom** аварийно завершается с кодом **WSAETIMEDOUT**, функция должна завешаться с кодом возврата **false**. Любой другой аварийный код завершения должен приводить к исключительной ситуации (оператор **throw**), соответствующей разработанной в практическом занятии № 2 функции обработки ошибок.

```
// -- послать запрос серверу и получить ответ
// Назначение: функция предназначена для широковещательной
//      пересылки позывного серверу и приема от
//      него ответа.

bool  GetServer(
          char*            call, //[in] позывной сервера
          short            port, //[in] номер порта сервера
          struct sockaddr* from, //[out] указатель на SOCKADDR_IN
          int*             flen  //[out] указатель на размер from
              )
// Код возврата: в случае успешного завершения функция
//        возвращает значение true, иначе возвращается
//        значение false.
// Примечание: - параметр call – строка, заканчивающаяся 0x00
//        и содержащая позывной сервера;
//        - параметр from – содержит указатель структуры,
//        которая содержит параметры сокета откликнувшегося сервера.
```

Рис. 7.5.3. Описание функции GetServer

Создайте новое приложение **ClientB**, вызывающее функцию **GetServer**. Запустите сервер **ServerU,** разработанный в практической работе № 3. Убедитесь с помощью отладчика, что происходит обмен данными и функция **GetServer** завершается с возвратом **false** после получения неверного отклика.

7.5.5. Взаимодействие сервера и клиента

Задание 5. Запустите на разных компьютерах программы **ClientB** и **ServerB**. Внесите изменения в программу **ClientB** для того, чтобы она выводила на экран консоли параметры сокета сервера, откликнувшегося на позывной. Внесите изменения в программу **ServerB** для того, чтобы она выводила на экран консоли параметры клиента, отправившего правильный позывной в адрес сервера.

Примечание. Разработанные функции **GetRequestFromClient** и **GetServer** имеют существенный недостаток. После вызова функции **recvfrom** они переводят поток в режим ожидания. Выход из этого состояния возможен лишь в том случае, если в адрес сокета поступило сообщение или исчерпан допустимый интервал ожидания. Такой алгоритм работы делает эти функции мало применимыми. Сохраните тексты этих функций, они будут дорабатываться в следующих практических работах.

Задание 6. Измените программу **ServerB** таким образом, чтобы при запуске она проверяла наличие в локальной сети еще одного такого же сервера (точнее сервера, с тем же позывным) и выдавала на экран консоли предупредительное сообщение о количестве существующих серверов и их IP-адресах.

7.6. Практическая работа № 5
Использование символических имен компьютеров

7.6.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков применения символических имен компьютеров при разработке распределенного в локальной сети приложения.

Результатом практической работы являются разработанное распределенное приложение, использующее символические имена компьютеров.

7.6.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в разделах 2.8.1, 2.8.3, 3.16 пособия.

7.6.3. Определение адреса компьютера по его символическому имени

Задание 1. Разработайте функцию **GetServerByName**, описание которой приводится на рис. 7.6.1. Функция предназначена для поиска сервера по его символическому имени и позывному. При этом предполагается, что в локальной сети работает одна из систем (DNS, NetBIOS over TCP/IP), разрешающих символические имена компьютеров. Функция **GetServerByName** является, в некотором смысле, альтернативой функции **GetServer** (практическая работа № 4) и должна использоваться в том случае, если известно символическое имя компьютера, на котором запущен сервер. Для поиска сервера фунция **GetServerByName** должна применить функцию **gethostbyname**, описание которой приводится в разделе 3.16. Там же имеется описание структуры **hostent**, которая используется этой функцией для хранения результата работы функции **gethostbyname**. После того, как IP-адрес сервера определен, необходимо установить необходимый номер порта и послать позывной в адрес сокета сервера. В остальном функция **GetServerByName** должна работать по тому же принципу, что и функция **GetServer**. Создайте новое приложение **ClentS**, вызывающее функцию **GetServerByName**. Проверьте работоспособность приложения при работе с программой **ServerB**.

Примечание. Функция **GetServerByName** имеет те же недостатки, что и функция **GetServer.** Сохраните текст этой функции, она будет дорабатываться в следующих практических работах.

```
// -- послать запрос серверу, заданному символическим именем
// Назначение: функция предназначена для пересылки позывного
//      серверу, адрес которого задан в виде
//      символического имени компьютера.

bool  GetServerByName(
       char*            name, //[in] имя компьютера в сети
       char*            call, //[in] позывной
        struct sockaddr* from, //[in,out] указатель на SOCKADDR_IN
       int*             flen  //[in,out] указатель на размер from
                    )
// Код возврата: в случае успешного завершения (сервер
//       откликнулся на позывной) возвращает значение true,
//       иначе возвращается  значение false.
// Примечание: - параметр name – строка, заканчивающаяся 0x00
//       и содержащая  символическое имя компьютера;
//     - параметр call – строка, заканчивающаяся 0x00 и
//       содержащая позывной сервера;
//     - параметр from  содержит указатель структуры
//       SOCKADDR_IN, которая включает  параметры сокета
//       откликнувшегося сервера; перед вызовом функции поле
//       sin_port должно  быть заполнено; если после вызова
//       функции код возврата равен true, то структура
//       SOCKADDR_IN содержит все параметры сокета сервера.
```

Рис. 7.6.1. Описание функции GetServerByName

7.6.4. Определение имени компьютера по его сетевому адресу

Задание 2. Доработайте программу **ServerB** таким образом, чтобы она выводила на экран консоли символическое имя собственного компьютера и символические имена компьютеров клиентов, которые подключаются к серверу. Программа **ServerB** должна использовать функции **gethostname** и **gethostbyaddr**, описание которых приводится в разделе 3.16 пособия.

7.7. Практическая работа № 6
Работа с интерфейсом Named Pipe

7.7.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков использования интерфейса Named Pipe для разработки распределенного приложения.

Результатом практической работы являются разработанное распределенное приложение, применяющее интерфейс Named Pipe.

7.7.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в главе 4 пособия.

7.7.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой сервера, изображенной на рис. 4.2.1. Создайте с помощью Visual Studio новое консольное приложение **ServerNP** (наименование проекта), которое будет использовано для построения серверной части распределенного приложения (сервера). Реализуйте блоки 1 и 4 сервера. В блоке 1 используются функции **CreateNamedPipe** и **ConnectNamedPipe**, описание которых приводится в разделе 4.3 пособия. Там же приведены примеры использования этих функций. Следует обратить внимание на формат имени канала, используемый функцией **CreateNamedPipe** – он должен быть локальным. Пусть имя создаваемого именного канала будет ***Tube***. В блоке 4 используется функция **DisconnectNamedPipe** для разрыва соединения. Описание функции приводится в разделе 4.3 пособия. Разработайте функции обработки ошибок интерфейса Named Pipe, работающие по тому же принципу, что и функции обработки ошибок Winsock2, используемые в предыдущих практических работах. Запустите приложение **ServerNP** и убедитесь, что поток приостановился для ожидания соединения после вызова функции **ConnectNamedPipe**.

7.7.4. Разработка клиентской части распределенного приложения

Задание 2. Ознакомьтесь со схемой клиента, изображенной на рис. 4.2.1. Создайте с помощью Visual Studio новое консольное приложение **ClientNP**, которое будет использовано для построения клиентской части распределенного приложения (клиента). Реализуйте блоки 1 и 4 клиента. В блоке 1 применяется функция **CreateFile**, описание которой приводится в разделе 4.4 пособия. Установите в параметрах вызова функции **CreateFile** такое же имя именованного канала, что и для сервера (задание 1). Запустите программу **ClientNP** отдельно (без сервера) и проверьте работоспособность функций обработки ошибок.

7.7.5. Обмен данными между клиентом и сервером

Задание 3. Запустите на выполнение программу **ServerNP**. После того, как программа **ServerNP** перейдет в состояние ожидания, запустите на этом же компьютере программу **ClientNP**. Убедитесь, что программа **ServerNP** вышла из состояния ожидания и успешно

завершилась. Программа **ClientNP** тоже должна завершиться без ошибок.

Задание 4. Реализуйте блоки 2 и 3 программ **ServerNP** и **ClientNP**. Добейтесь, чтобы программы обменялись сообщениями с помощью функций **WriteFile** и **ReadFile**, описание которых приводится в разделе 4.5 пособия.

Задание 5. Внесите изменения в программы **ServerNP** и **ClientNP** таким образом, чтобы программы взаимодействовали так же, как и программы **ServerT** и **ClientT** в заданиях 14 и 15 практической работы № 2.

Задание 6. Внесите изменения в программу **ClientNP** и добейтесь взаимодействия программ клиента и сервера в случае расположения на разных компьютерах локальной сети. Следует иметь в виду, что при вызове функции **CreateFile** теперь следует использовать сетевой формат имени именованного канала, описанный в разделе 4.3 пособия. В сетевом формате используется символическое имя серверного компьютера. Напомним, что это имя можно получить с помощью утилиты **hostname**, описанной в разделе 2.9.

Задание 7. Разработайте новую программу **ClientNPt**, использующую функцию **TransactNamedPipe** (раздел 4.6 пособия) вместо пары функций **WriteFile** и **ReadFile**. Добейтесь взаимодействия программы **ClientNPt** с сервером **ServerNP**.

Задание 8. Разработайте еще одну новую программу **ClientNPct**, использующую функцию **CallNamedPipe** (раздел 4.6 пособия). Добейтесь взаимодействия программы **ClientNPct** с сервером **ServerNP**.

7.8. Практическая работа № 7
Работа с интерфейсом Mailslots

7.8.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков использования интерфейса Mailslots для разработки распределенного приложения.

Результатом практической работы является разработанное распределенное приложение, применяющее интерфейс Mailslots .

7.8.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в главе 5 пособия.

7.8.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой сервера, изображенной на рис. 5.2.1. Создайте с помощью Visual Studio новое консольное приложение **ServerMS** (наименование проекта), которое будет использовано для построения серверной части распределенного приложения (сервера). Реализуйте блоки 1 и 3 сервера. В блоке 1 используется функция **CreateMaislot**, описание которой приводится в разделе 5.3 пособия. Следует обратить внимание на формат имени канала, используемый функцей **CreateMaislot –** он должен быть локальным. Пусть имя создаваемого именного канала будет ***Box***. Кроме того, установите в параметрах функции **CreateMaislot** бесконечный интервал ожидания для функции **ReadFile** и максимальную длину сообщения, равную 300 байт. Разработайте функции обработки ошибок интерфейса Mailslot, работающие по тому же принципу, что и функции обработки ошибок Winsock2, используемые в предыдущих практических работах. Запустите приложение **ServerMS** и убедитесь, что поток не приостановился для ожидания и программа **ServerMS** завершилась без ошибок.

Задание 2. Реализуйте в программе **ServerMS** блок 2 схемы сервера (рис. 5.2.1). В блоке 2 применяется универсальная функция **ReadFile**, описание которой приводится в разделе 4.5 пособия. Запустите программу. Убедитесь, что главный поток программы **ServerMS** приостановился для ожидания сообщения от клиента. Завершите программу с помощью отладчика.

Задание 3. Измените в параметрах вызова функции **CreateMaislot** программы **ServerMS** интервал времени ожидания на значение, соответствующее 3 мин. Запустите программу **ServerMS** и убедитесь, что главный поток приостановился на 3 мин для ожидания сообщения клиента. Обработайте этот случай: выведите соответствующее сообщение на консоль сервера.

7.8.4. Разработка клиентской части распределенного приложения

Задание 4. Ознакомьтесь со схемой клиента, изображенной на рис. 5.2.1. Создайте с помощью Visual Studio новое консольное приложение **ClientMS**, которое будет использовано для построения клиентской части распределенного приложения (клиента). Реализуйте схему клиента полностью. Для реализации блока 1 необходимо использовать функцию **CreateFile**, описание которой приводится в разделе 5.4. Установите локальный формат имени почтового сервера в параметрах функции. Для отправки сообщения
используется универсальная функция **WriteFile**, описанная разделе 4.5. Разработайте функции обработки ошибок интерфейса Mailslot. Программа **ClientNS** должна пересылать серверу сообщение ***Hello from Maislot-client***. Запустите программу **ClientMS** без наличия на компьютере сервера. Убедитесь, что обработка ошибок работает корректно.

7.8.5. Обмен данными между клиентом и сервером

Задание 5. Запустите на выполнение программу **ServerMS**. После того, как программа **ServerMS** перейдет в состояние ожидания запустите на этом же компьютере программу **ClientMS**. С помощью отладчика убедитесь, что сообщение от клиента к серверу передается корректно.

Задание 6. Внесите такие изменения в программу **ClientMS**, чтобы она могла пересылать сообщение серверу **ServerMS**, который расположен на другом компьютере. Добейтесь работоспособности распределенного приложения **ClientMS – ServerMS**.

Задание 7. Внесите такие изменения в программу **ClientMS**, чтобы она могла пересылать сообщение нескольким экземплярам сервера **ServerMS**, расположенным на других компьютерах локальной сети. Добейтесь работоспособности одной клиентской программы с тремя экземплярами сервера.

Задание 8. Увеличьте максимальный размер пересылаемого сообщения до 500 байт (параметры функции **CreateMaislot**) и проверьте возможность работы одного клиента с тремя серверами.

Задание 9. Оцените скорость пересылки 1000 сообщений от одного клиента одному серверу по тому же принципу, как это было сделано в задании 14 практической работы № 2.

7.9. Практическая работа № 8

Разработка параллельного сервера

7.9.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков разработки параллельного сервера, одновременно обслуживающего разнотипные запросы нескольких клиентов.

Результатом практической работы является разработанный параллельный сервер на основе модели ConcurrentServer, описанной в главе 6 пособия.

7.9.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в главах 3, 5, 6 пособия.

7.9.3. Предварительные замечания

Разработка параллельного сервера будет опираться на модель ConcurrentServer, описанную в главе 6 пособия. Последняя версия модели изложена в разделе 6.13. Именно на этот вариант будут ориентированны все дальнейшие построения.

Для того чтобы избежать двойного толкования, будем считать далее, что модель ConcurrentServer – спецификация, описывающая разрабатываемый здесь параллельный сервер с именем ConcurrentServer.

Кроме того, в изложении часто будут использоваться названия функций, которые являются функциями потоков (раздел 6.4 пособия). Для краткости, если это не приводит к двусмысленности, потоки будут именоваться по имени потоковой функции. Например, функция **AcceptServer** является потоковой функцией, тогда соответствующий ей поток будем именовать – поток **AcceptServer**. Следует отметить, что может быть создано несколько различных потоков, имеющих общую потоковую функцию. В таких случаях будем говорить о нескольких экземплярах потока.

7.9.4. Исследование структуры параллельного сервера

Задание 1. Исследуйте структуру сервера **ConcurrentServer**, изложенную в разделах 6.1–6.3 и 6.11–6.13 и изображенную на рис. 6.13.1. В табл. 7.9.1 приведены все основные программные компоненты сервера. Составьте краткую спецификацию каждого программного компонента, отражающую назначение, краткое

описание алгоритма, используемые структуры данных и способ взаимодействия с другими компонентами.

Таблица 7.9.1

Компонент	Назначение
main	Главный поток сервера
AcceptServer	Подключение клиентов к серверу
ConsolePipe	Серверная часть консоли управления
GarbageCleaner	Сборщик мусора
DispatchServer	Диспетчеризация запросов клиента
DllMain	Главная функция динамически подключаемой библиотеки
SSS	Запуск обслуживающего потока
EchoServer	Отладочный поток обслуживания

Задание 2. В табл. 7.9.2 приведены все основные структуры данных, используемые сервером. Составьте краткое описание каждой структуры, отразите состав информации, назначение и способ их использования программными компонентами.

Таблица 7.9.2

Структуры	Состав информации
ListContact	Список подключения клиентов (раздел 6.5)
TableService	Таблица обслуживающих потоков (раздел 6.13)

Задание 3. В табл. 7.9.3 приведено краткое описание команд управления сервером, которые вводятся с удаленной консоли **RConsole** (клиентская часть консоли управления сервером) и исполняются сервером. Добавьте к описанию каждой команды принципы ее реализации, название программного компонента, реализующего команду. Опишите структуры данных, которые предполагается применить для хранения списка команд (далее будем называть его **TalkersCmd**), а также для сбора необходимой статистики (будем называть эту структуру **StatsInfo**).

Таблица 7.9.3

Команда	Краткое описание команды
start	Команда разрешает подключение клиентов к серверу
stop	Команда запрещает подключение клиентов к серверу
exit	Команда завершает работу сервера
statistics	Вывод статистики: общее количество подключений; количество

	активных подключений, количество отказов в обслуживании
wait	Команда приостанавливает подключение клиентов до тех пор, пока не обслужится последний клиент, подключенный к серверу
shutdown	Команда равносильна последовательности команд: wait, exit
getcomand	Служебная команда, которая не предназначена для ввода с консоли управления, а устанавливается сервером для указания, что сервер готов принять и обработать, очередную команду управления

Задание 4. В табл. 7.9.4 приведены коды команды (или запроса), которые будет обрабатывать поток **DispatchServer** после того, как клиент подключится к клиенту. Эти коды хранятся в таблице **TableService**, и каждому коду соответствует свой обслуживающий поток (разделы 6.12–6.13 пособия).

Таблица 7.9.4

Код команды (запрос**)**	Краткое описание команды
echo	Вызов обслуживающего потока EchoServer. EchoServer – потоковая функция, обеспечивающая прием данных от подключившегося клиента и пересылку этих же данных обратно без изменения. Условие завершения работы: прием данных нулевой длины
time	Вызов обслуживающего потока TimeServer. TimeServer – потоковая функция, принимающая только символьную последовательность ***time*** и отвечающая системным значением даты и времени серверного компьютера в формате ***dd.mm.yyyy/hh:mm:ss***. Условие завершение работы: прием любой последовательности символов, отличной от ***time***
rand	Вызов обслуживающего потока RandServer. RandServer – потоковая функция, принимающая только строку ***rand*** и отвечающая случайным целым четырехбайтовым числом в intel-формате. Условие завершение работы: прием любой последовательности символов, отличной от ***rand***

Краткое описание обслуживающих потоков приводится в табл. 7.9.4. На первом этапе предполагается, что сервер будет обслуживать три различных запроса: ***echo***, ***time*** и ***rand***. Опишите принципы реализации обслуживающих потоков. Для получения системной даты и времени, а также случайных чисел в обслуживающих потоках следует воспользоваться функциями стандартной библиотеки C++ [13].

7.9.5. Создание главного потока main

Задание 5. Создайте с помощью Visual Studio новое консольное приложение **ConcurrentServer** (наименование проекта). Включите необходимые директивы компилятора для подключения библиотеки **WS2_32.LIB** (раздел 3.2 пособия)**.** Подключите функции обработки ошибок интерфейса Winsock2, разработанные в практической работе № 2. Откомпилируйте приложение и убедитесь в отсутствие ошибок.

Задание 6. Создайте в рамках проекта **ConcurrentServer** потоковую функцию **AcceptServer**. Пусть на первом этапе эта функция циклически выдает сообщение ***AcceptServer*** на экран консоли каждые 5 с (используйте функцию **Sleep**, описанную в разделе 6.4). Обратите внимание на правильное оформление потоковой функции. Опираясь на описание функции **CreateThread** и пример в разделе 6.4, обеспечьте создание потока **AcceptServer**. Следует помнить о необходимости использовать функции **WaitForSingleObject** и **CloseHandle**. Проверьте работоспособность приложения.

Примечание. Единственный параметр функции **AcceptServer** будет в дальнейшем использоваться для передачи адреса области памяти, в котором будет храниться команда управления, записанная туда потоком **ConsolePipe**. Поток **ConsolePipe** (серверная часть консоли управления) получает команды управления (табл. 7.9.3) от удаленной консоли **RConsole** через именованный канал (глава 4).

Задание 7. Обеспечьте передачу номера порта сервера через параметры функции **main**. Причем если параметр не устанавливается в вызове командной строки сервера **ConcurrentServer**, то по умолчанию должен применяться порт 2000. Выведите на консоль сервера номер используемого порта. Кроме того, следует передать через единственный параметр потока **AcceptServer** начальную команду управления сервером start (табл. 7.9.3), запускающую процесс подключения клиентов. Убедитесь в корректности передачи параметров, запустив приложение **ConcurrentServer** через командную строку.

7.9.6. Реализация потока AcceptServer

Задание 8. Удалите отладочный цикл вывода в функции **AcceptServer**, созданный в предшествующем задании. Реализуйте в рамках этой функции блоки 1, 2, 3 и 6 схемы сервера изображенной на рис. 3.4.2. При этом следуйте указаниям заданий 2–5 практической работы № 2. Убедитесь в работоспособности всего приложения **ConcurrentServer**.

Примечание. Номер используемого сервером порта, полученный через параметры функции **main** или установленный по умолчанию, следует каким-нибудь образом передать в функцию **AcceptServer**. Очевидно, самый простой способ – это организация общедоступной области памяти для потоков **main** и **AcceptServer**.

Задание 9. Реализуйте цикл обработки команд в рамках функции (и потока) **AcceptServer**. Возьмите за основу пример функции **CommandsCycle**, приведенный в разделе 6.9 пособия. На этом этапе разработки замените вызов функции **AcceptCycle** на вызов функции **Sleep** (раздел 6.4). Убедитесь, что команда управления **start** корректно передана в функцию **CommandsCycle**. Следует обратить внимание на применение функции **ioctlsocket** (описание в разделе 6.9 пособия), которая переключает сокет в режим без блокировки.

Задание 10. Реализуйте цикл подключения в рамках функции (и потока) **AcceptServer** (в предыдущем примере вместо вызова этой функции вызывалась функция **Sleep**). Используйте пример функции **AcceptCycle**, приведенный в разделе 6.9 пособия. Обратите внимание, что вся информация о подключении клиента записывается в список **ListContact** для последующего его использования в потоке **DispathServer**. Кроме того, следует обратить внимание на обработку ошибок функции **accept**, применяемую для сокета в режиме без блокировки. Используйте программу **ClientT**, разработанную в практической работе № 2, в качестве клиента для проверки процесса подключения клиентов функцией **AcceptServer**.

7.9.7. Реализация потока DispatchServer

Задание 11. Поток **DispathServer** должен создаваться и запускаться в функции **main** сразу после запуска потока **AcceptServer**. В качестве параметра, передаваемого в потоковую функцию
DispathServer, будем использовать адрес области памяти, в которой хранится команда управления, записанная туда потоком **ConsolePipe** (эта область уже используется потоком **AcceptServer**). На этапе этого задания по аналогии с заданием 6 зациклите вывод сообщения ***DispathServer***. Не забудьте о необходимости применения функций **WaitForSingleObject** и **CloseHandle**. Убедитесь, что поток **DispathServer** создается и работает.

Примечание. Обратите внимание, что область памяти команды управления используется двумя потоками: **AcceptServer** и **DispathServer**. В табл. 7.9.3 приведены команды управления, на которые будет реагировать только поток **AcceptServer** (будем называть их командами управления **AcceptServer**). В будущем предполагается расширить список команд, добавив туда команды управления
DispathServer. Примером такой команды может служить команда ***exclude sss***, позволяющая исключить из таблицы обслуживающих потоков **TableService** строку с кодом запроса ***sss***. Для управления каждым потоком будем использовать свой (кроме общей команды **exit**) набор команд управления. Рекомендуется при объявлении общей области памяти для хранения команды управления использовать квалификатор **volatile** как это сделано в примерах раздела 6.4.

Задание 12. В соответствии со схемой сервера **ConcurrrenServer**, поток **DispathServer** должен последовательно просматривать список **ListContact**, в котором хранятся все сведения о подключившихся клиентах, и считывать запрос подключившегося клиента. Если этот запрос соответствует одной из строк таблицы **TableService**
(запросы приведены в табл. 7.9.4), то **DispathServer** должен запустить соответствующий запросу поток, передав ему в качестве параметра соответствующий элемент списка **ListContact**. Удалите отладочный цикл, построенный в предыдущем задании, и реализуйте цикл сканирования списка **ListContact**. Условием выхода из цикла пусть будет команда управления **exit**. При обнаружении в списке подключившегося, но не обслуженного клиента (это можно определить по специальным признакам, записанным в элемент списка), следует прочитать первое сообщение клиента, которое должно быть кодом запроса, с помощью функции **recv** (раздел 3.11). Для отладки выведите считанный код запроса подключившегося клиента на экран консоли. Убедитесь, что код запроса считывается корректно. Используйте программу **ClientT** при отладке потока **DispathServer**. Проверьте работу сервера с несколькими клиентами в сети.

Примечание. Заметим, что сокет, обрабатываемый потоком **DispathServer**, находится в режиме без блокировки, так как получен командой **accept**, использующей неблокирующий сокет, и поэтому

требует соответствующего алгоритма обработки Если бы использовался альтернативный режим (с блокировкой), то был бы возможен случай, когда **DispathServer** остановился в ожидании запроса от клиента, который по неизвестной причине после подключения не отправил серверу запрос. При этом все остальные клиенты, отправившие запрос, ожидали бы обслуживания. После того, как запрос получен и обработан, следует переключить сокет в блокирующий режим работы, т. к. теперь с ним будет работать обрабатывающий поток.

7.9.8. Реализация библиотеки обслуживающих серверов

Задание 13. Создайте с помощью Visual Studio новое DLL-приложение с именем **ServiceLibrary**. Реализуйте в рамках динамически подключаемой библиотеки потоковую функцию **EchoServer**. В качестве параметра этой функции должен передаваться элемент списка **ListContact**, в котором хранятся все параметры подключения (в том числе и сокет). Алгоритм работы функции совпадает с алгоритмом работы **ServerT**, разработанным в практической работе № 2, с учетом того, что подключение клиента уже выполнено. Кроме того, перед завершением работы функция **EchoServer** должна сделать соответствующую отметку в переданном ей элементе списка **ListContact**. Реализуйте таблицу **TableService** и экспортируемую функцию **SSS**, используя пример в разделе 6.13 пособия. Заполните таблицу кодами запросов, приведенными в табл. 7.9.4. Для всех кодов при заполнении таблицы **TableService** для отладки используйте один и тот же адрес обслуживающего потока **EchoServer**.

Задание 14. Обеспечьте передачу имени файла библиотеки через второй параметр функции **main**. Если параметр не задан, то по умолчанию будем предполагать, что файл библиотеки имеет имя **ServiceLibrary**. Загрузите динамически подключаемую библиотеку в рамках функции **main** с помощью функции **LoadLibrary**, описание которой приводится в разделе 6.13 пособия. После загрузки библиотеки импортируйте функцию **SSS**, применив функцию **GetProcAddress** (раздел 6.13). Обеспечьте возможность вызова импортированной функции потоком **DispatchServer**. Перед завершением функции **main** (после завершения всех потоков)

отключите библиотеку с помощью функции **FreeLibrary** (раздел 6.13). Используйте примеры программ, приведенные в разделе 6.13. С помощью отладчика убедитесь в корректной передаче параметра с именем функции, успешной загрузке библиотеки и импорте функции.

Примечание. Параметризация имени файла библиотеки дает возможность загружать разные версии библиотеки **ServiceLibrary** при инициализации сервера. Целесообразно было бы разработать несколько функций, которые бы инкапсулировали функции **LoadLibrary**, **FreeLibrary**, **GetProcAddress**, и уже эти функции использовать в **main**. Кроме того, имеет смысл поддерживать номер версии библиотеки (как это сделано, например, для библиотеки WS2_32.DLL, описанной в главе 3).

7.9.9. Запуск обслуживающего потока из функции DispatchServer

Задание 15. Удалите отладочный вывод из функции **DispatchServer**, реализованный в предшествующих заданиях. Используя импортированную из динамической библиотеки функцию **SSS**, обеспечьте запуск обрабатывающего потока, соответствующего запросу клиента. Кроме того, обработайте неправильные запросы, не соответствующие таблице **TableService** и таблице 7.9.4. В случае неправильного запроса функция **DispatchServer** должна передать клиенту сообщение ***ErrorInquiry***, разорвать соединение (выполнить функцию **closesocket**, описанную в разделе 3.7 пособия) и сделать соответствующую отметку в элементе списка **ListContact** (адрес элемента найден в результате сканирования списка **ListContact**, раздел 7.9.7). Кроме того, следует иметь в виду, что клиент может иногда задержать отправку запроса после подключения. Необходимо предусмотреть диагностику для этого случая. Используйте программу **ClientT**, разработанную в практической работе № 2. Внесите необходимые изменения в программу **ClientT** для того, чтобы она в первом сообщении серверу передавала запрос (табл. 7.9.4). Убедитесь, что **ConcurrentServer** запускает обслуживающий поток для корректных запросов, а также отсылает сообщение и разрывает соединение в ответ на ошибочные запросы.

7.9.10. Реализация потока GarbageCleaner

Задание 16. Поток **GarbageCleaner** должен создаваться и запускаться в функции **main** сразу после запуска потока **DispatchServer**. В качестве параметра, передаваемого в потоковую функцию **GarbageCleaner**, будем использовать адрес области памяти, в которой хранится команда управления, записанная туда потоком **ConsolePipe** (эта область уже используется другими потоками). На этапе этого задания по аналогии с заданием 6 зациклите вывод сообщения ***GarbageCleaner***. Кроме того, обеспечьте доступ потока ***GarbageCleaner*** к списку **ListContact**. Следует помнить о необходимости применения функций **WaitForSingleObject** и **CloseHandle**. Убедитесь, что поток **GarbageCleaner** создается и работает.

Задание 17. Удалите отладочный вывод в функции **GarbageCleaner** и создайте цикл сканирования списка **ListContact**. Функция **GarbageCleaner** должна выявлять неиспользуемые элементы списка **ListContact** (об этом есть соответствующая отметка, сделанная потоком **DispatchServer** или обслуживающим сервером). Запустите сервер **ConcurrentServer** и обеспечьте подсоединение к нему одного клиента. Убедитесь с помощью отладчика, что поток **GarbageCleaner** удаляет неиспользуемые элементы списка **ListContact** в случае успешного обслуживания клиента, выдачи клиентом неправильного запроса, а также, если клиент завершился аварийно во время сеанса связи. В последнем случае может потребоваться доработка функции **EchoServer**.

Примечание. Такой алгоритм работы потока **GarbageCleaner** не является оптимальным, т. к. нерационально используется ценный ресурс процессора. Для исправления этого недостатка можно использовать функцию **Sleep** (раздел 6.4) внутри цикла сканирования для организации задержки (освобождения ресурса), а интервал времени задержки можно параметризировать. Другой способ устранения нерационального использования процессорного времени, основанный на назначении приоритетов, будет предложен в следующих заданиях.

7.9.11. Синхронизация потоков AcceptServer и GarbageCleaner

Задание 18. Ознакомьтесь со схемой применения механизма критических секций, приведенной в разделе 6.5 пособия. Критическим ресурсом в сервере **ConcurrentServer** является список **ListContact**: одновременное удаление (поток **GarbageCleaner**) и добавление (поток **AcceptServer**) элементов списка может привести к

его разрушению. Используйте механизм критических секций (раздел 6.5 пособия) для синхронизации потоков **AcceptServer** и **GarbageCleaner**. Убедитесь в работоспособности сервера при интенсивных подключениях и отключениях нескольких клиентов одновременно. Разработайте на базе программы **ClentT** новую программу клиента **ClientTy**, осуществляющую в цикле подключение к серверу и отключение. Используйте эту программу для тестирования сервера.

7.9.12. Синхронизация потока AcceptServer и обслуживающих потоков

Задание 19. Ознакомьтесь со схемой использования механизма асинхронного вызова процедур, приведенной в разделе 6.6 пособия. Обеспечьте вывод сообщений на консоль сервера в рамках потока **AcceptServer** о старте и завершении работы обслуживающего потока с помощью механизма асинхронных процедур. Выводимое на консоль сообщение должно содержать наименование обслуживающего сервера, а также дату и время начала и завершения его работы (с функциями для работы с системным временем можно ознакомиться в [13, 14]). В параметрах функции **QueueUserAPC** (раздел 6.6) используется дескриптор потока, исполняющего асинхронную процедуру. В нашем случае это поток **AcceptServer**. Рекомендуется при вызове асинхронных процедур в качестве параметра указывать адрес соответствующего элемента списка **ListContact**. Обслуживающий поток должен иметь доступ к этому дескриптору. Следует обратить внимание на второй параметр функции **SleepEx** (раздел 6.6), которая будет использоваться в функции **AcceptServer** для запуска асинхронных процедур. Значение этого параметра должно быть **TRUE**. Значение первого параметра функции **SleepEx** рекомендуется установить равным нулю. Убедитесь в работоспособности сервера при обслуживании нескольких клиентов одновременно.

Примечание. Полезной была бы разработка функции, инкапсулирующую функцию **QueueUserAPC**. Процесс разработки обслуживающих серверов должен сопровождаться разработкой технологией и соответствующего API, упрощающих разработку новых обслуживающих серверов.

7.9.13. Предупреждение зацикливания обслуживающих потоков

Задание 20. Ознакомьтесь со схемой использования ожидающего таймера, приведенной в разделе 6.7. Обеспечьте создание в функции **DispatchServer** для каждого запускаемого обслуживающего сервера ожидающий таймер, отслеживающий максимальное время работы обслуживающего сервера. Установите значение ожидающего таймера так, чтобы он переходил в сигнальное состояние через 1 мин после запуска обслуживающего сервера (т. е. 1 мин – это максимально допустимое время работы обслуживающего сервера). Дескриптор ожидающего таймера рекомендуется создать в соответствующем элементе списка **ListContact**. Если обслуживающий сервер завершает свою работу до истечения установленного максимального интервала его работы, то ожидающий таймер должен быть отменен с помощью функции **CancelWaitableTimer** (раздел 6.7). Это рекомендуется выполнить в асинхронной процедуре, разработанной для вывода сообщения о завершении работы обслуживающего потока (раздел 7.9.12). Если же таймер срабатывает (обслуживающий поток работает больше установленного максимального интервала времени), следует предпринять действия по завершению потока. Используйте процедуру завершения ожидающего таймера, которая помещается в очередь асинхронных процедур потока, создавшего ожидающий таймер (раздел 6.7). Рекомендуется в качестве параметра этой процедуры использовать адрес соответствующего элемента списка **ListContact**. Запуск этой процедуры можно осуществить с помощью функции **SleepEx**. В самой процедуре завершения ожидающего таймера уже можно завершить зациклившийся поток с помощью функции **TeminateThread** (раздел 6.4). Внесите изменения в программу клиента **ClientT**, созданную в предыдущих заданиях для тестирования, чтобы она осуществляла обмен данными с сервером заведомо больше времени, чем 1 мин и используйте эту программу для тестирования. Убедитесь в работоспособности созданного механизма предупреждения зацикливания обслуживающих потоков.

Примечание. Применение функции **TeminateThread** может привести к проблемам в работе сервера. Дело в том, что такое завершение потока не гарантирует корректного освобождения всех используемых потоком ресурсов. Правильнее было бы использовать специальное API обслуживающих серверов (это уже обсуждалось в примечаниях выше), которое бы могло корректно завершать работу потока.

7.9.14. Синхронизация потоков AcceptServer и DispatchServer

Задание 21. Поток **DispatchServer** постоянно сканирует список **ListContact** в поиске подключившегося, но не получившего обслуживания клиентского подключения. Очевидно, что сканировать список нужно только после того, как осуществится очередное подключение в потоке **AcceptServer**. Ознакомьтесь с принципами применения механизма события, изложенными в разделе 6.12 пособия. Примените этот механизм для синхронизации потоков **AcceptServer** и **DispatchServer**. Событие должно быть создано в потоке **AcceptServer**, а момент его наступления должен контролироваться в потоке **DispatchServer**. Наступление события должно соответствовать подключению очередного клиента в потоке **DispatchServer**.

7.9.15. Реализация потока ConsolePipe

Задание 22. Поток **ConsolePipe** должен создаваться и запускаться в функции **main** после запуска потоков **AcceptServer**, **DispatchServer** и **GarbageCleaner**. Как уже описывалось выше, основным назначением этого потока является ввод команд управления, а также вывод на консоль **RConsole** простейших диагностических сообщений. При этом предполагается, что обмен информацией между **ConsolePipe** и **RConsole** будет происходить по именованному каналу. Полученную от удаленной консоли команду функция **ConsolePipe** должна преобразовать в коды команд понятные потокам-получателям **AcceptServer**, **DispatchServer**, **GarbageCleaner** и помещена в общую для этих потоков область памяти (об этом уже говорилось выше). Поток-получатель команды управления после считывания команды из общей памяти должен поместить туда код служебной команды **getcommand**. Поток **ConsolePipe** циклически проверяет общую область памяти потоков и после получения команды **getcommand** запрашивает следующую команду управления сервером. По аналогии с другими потоками указатель на общую память для хранения команд следует передать потоку **ConsolePipe** при запуске в параметре потоковой функции. Ознакомьтесь с принципами организации обмена данными по именованному каналу, описанными в главе 4 пособия. Разработайте потоковую функцию **ConsolePipe**, позволяющую вводить команды

управления **start** и **stop** (табл. 7.9.3). При получении этих команд выведите диагностирующие сообщения на экран консоли сервера. Используйте результаты практической работы № 6. Символическое имя канала должно быть передано как параметр функции **main**. Используйте для отладки программу **CllientNP**, разработанную в практической работе № 6. Убедитесь в том, что сервер реагирует на команды **start** и **stop** (на экране консоли сервера должны быть соответствующие диагностические сообщения**)**.

Задание 23. Доработайте функцию **ConsolePipe** таким образом, чтобы она могла принимать и обрабатывать все возможные команды, приведенные в таблице 7.9.3. Если код пересланной команды является корректным, обеспечьте отправку в адрес удаленной консоли (пока ее функцию выполняет программа **CllientNP**) этого же кода, в ином случае оправьте сообщение ***nocmd***. Убедитесь в работоспособности функции **ConsolePipe**.

7.9.16. Обработка команд управления в потоке AcceptServer

Задание 24. Для обработки команд управления в рамках потока **AcceptServer** предусмотрена специальная функция **CommandsCycle** (или ее аналог), которая уже рассматривалась в разделе 7.9.6. Доработайте эту функцию так, чтобы она реализовывала выполнение всех команд управления, приведенных в табл. 7.9.3, кроме команды **statistics**. Убедитесь, что сервер выполняет все вводимые команды управления.

7.9.17. Сбор и вывод статистических данных о работе сервера

Задание 25. Сервер **ConcurrentServer** должен поддерживать следующую статистику: общее количество подключений с начала работы сервера, общее количество отказов в обслуживании (по всему спектру причин суммарно) за все время работы сервера, количество подсоединений, которые завершились успешно, количество активных подключений на момент запроса. Создайте необходимые переменные-счетчики, в которых будет отражаться данные статистики. Определите программные компоненты сервера, которые должны участвовать в сборе статистики. Поскольку эти компоненты работают асинхронно, используете для изменения значений переменных-счетчиков механизм атомарных операций, описанный в разделе 6.8. пособия.

Реализуйте выполнение команды управления **statistics**. Убедитесь в корректном подсчете статистики.

7.9.18. Разработка обслуживающих потоков

Задание 26. До сих пор для отладки сервера использовался только один обслуживающий поток **EchoServer**, который подключался для всех возможных запросов (табл. 7.9.4). Проанализируйте структуру функций обслуживающих потоков и разработайте спецификацию потоковой функции обслуживающего потока. Разработайте API, который бы инкапсулировал все функции Windows API и Winsock2 API. Используя разработанную спецификацию и применив API обслуживающего потока, реализуйте все потоковые функции обслуживающих потоков в соответствии с табл. 7.9.4 в рамках динамической библиотеки и откоррек-тируйте таблицу **TableService**. Убедитесь в работоспособности сервера.

7.9.19. Навигация сервера в локальной сети

Задание 27. Доработайте функции **GetRequestFromClient** и **PutAnswerToClient**, разработанные в практической работе № 4 для того, чтобы исправить отмеченные в примечании к разделу 7.5.5 недостатки. Используйте режим работы сокета без блокировки (раздел 6.9 пособия). Проверьте работоспособность функций в рамках распределенного приложения **ClientB-ServerB**, разработанного в практической работе № 4.

Задание 28. Реализуйте новый поток (потоковую функцию) **ResponseServer** в рамках сервера **ConcurrentServer**. Поток должен быть создан и запущен в функции **main** сервера, и предназначен для приема позывного на широковещательные запросы клиента для поиска сервера в локальной сети. Кроме того, поток **ResponseServer** должен сформировать ответ-приглашение для подключения. Используйте программу **ServerB** (практическая работа № 4) и доработанные в предыдущем задании функции **GetRequestFromClient** и **PutAnswerToClient**. Позывной сервера и номер UDP-порта для приема широковещательных сообщений следует сделать параметрами сервера. Продумайте и реализуйте механизмы управления потоком **ResponseServer**. Используйте программу **ClientB** для проверки работоспособности потока **ResponseServer**.

7.9.20. Установка приоритетов потоков сервера

Задание 29. Ознакомьтесь с принципами использования системы приоритетов в операционной системе Windows, описанными в разделе 6.10 пособия. Установите приоритеты для потоков **AcceptServer**, **DispatchServer**, **ConsolePipe**, **GarbageCleaner**, **ResponseServer** и для обслуживающих потоков. Убедитесь в работоспособности сервера **ConcurrentServer**.

Задание 30. Предложите методы динамического назначения приоритетов для потоков сервера, позволяющие управлять производительностью сервера **ConncurrentServer**.

7.10. Практическая работа № 9
Разработка клиента параллельного сервера

7.10.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков проектирования API и разработки на его основе программы – клиента параллельного сервера.

Результатом практической работы является разработанный набор функций (API), предназначенный для разработки клиентских программ, взаимодействующих с сервером **ConncurrentServer** .

7.10.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в главах 3, 5 и 6 пособия.

7.10.3. Разработка API клиента

Задание 1. Разработайте спецификацию (протокол), определяющую принципы взаимодействия клиентских программ с сервером **ConncurrentServer**. Разработайте API, предназначенный для использования на стороне клиента и инкапсулирующий все детали взаимодействия клиента с сервером.

7.10.4. Разработка клиентских приложений

Задание 2. Доработайте функцию **GetServer**, разработанную в практической работе № 4, для того, чтобы исправить отмеченные в примечании к разделу 7.5.5 недостатки. Используйте режим работы сокета без блокировки (раздел 6.9 пособия). Проверьте работоспособность функции в рамках распределенного приложения

ClientB-ServerB, разработанного в практической работе № 4. Включите функцию **GetServer** в состав API клиента.

Задание 3. Разработайте с применением API (задание 2) три клиентских программы: **ClientEcho**, **ClientTime** и **ClientRand**, предназначенные для тестирования сервера **ConncurrentServer**. Программа **ClientEcho** предназначена для тестирования запроса **echo**, **ClientTime** – для тестирования запроса **time** и **ClientRand** – для тестирования запроса **rand**. Все программы должны подключаться к серверу, отыскав его IP-адрес с помощью широковещательных сообщений, корректным и некорректным образом осуществлять запросы, проверять на максимальное время работы и т. д. Протестируйте работу клиентского API.

7.11. Практическая работа № 10
Разработка удаленной консоли

7.11.1. Цель и задачи работы. Основной целью практической работы является приобретение навыков проектирования API и разработки на его основе программы – удаленной консоли параллельного сервера.

Результатом практической работы является разработанный набор функций (API), предназначенный для разработки программы, реализующей клиентскую сторону консоли сервера **ConncurrentServer**,
а также программа **RConsole**, разработанная с применением этого API.

7.11.2. Теоретические сведения

Теоретические сведения, необходимые для выполнения практической работы, изложены в главах 3, 5 и 6 пособия.

7.11.3. Разработка API и программы RConsole

Задание 1. Разработайте спецификацию (протокол), определяющую принципы взаимодействия клиентской части консоли управления сервером **ConncurrentServer**. Разработайте API, предназначенный для использования на клиентской стороне консоли управления, которая бы инкап-сулировала все детали взаимодействия удаленной консоли с сервером.

Задание 2. Создайте с помощью Visual Studio новое консольное приложение **RConsole** (наименование проекта). Используя API,

разработанный в предыдущем задании, разработайте клиентскую часть консоли управления сервером **ConncurrentServer**. Программа **RConsole** должна позволять вводить команды управления сервером (табл. 7.9.3) и получать диагностические сообщения сервера. Убедитесь в работоспособности программы **RConsole**.

7.12. Выводы главы

1. Практическая разработка распределенных приложений является сложным и кропотливым процессом. Разработчик таких приложений должен обладать знаниями в области компьютерных сетей, уметь использовать программные интерфейсы, позволяющие организовать обмен данными в сети, обладать навыками параллельного программирования.
2. Построение распределенных приложений требует от разработчика решений, связанных с управлением распределенным приложением.
3. Для обеспечения масшабируемости распределенного приложения необходима разработка стандартов (спецификаций и API) взаимодействия различных его компонентов.
4. Распределенные приложения должны обладать высокой степенью параметризации, позволяющей настроить их для работы в постоянно изменяющихся условиях распределенной среды.

ЛИТЕРАТУРА

1. Гецци, К. Основы инженерии программного обеспечения / К. Гецци, М. Джайзайери, Д. Мандиролли. – СПб.: БХВ-Петербург, 2005. – 832 с.

2. Майер, Б. Объектно-ориентированное конструирование программных систем / Б. Майер. – М.: Русская Редакция, 2005. – 1232 с.

3. Орлов, С. Технологии разработки программного обеспечения / С. Орлов. – СПб.: Питер, 2003. – 480 с.

4. Побегайло, А. П. Системное программирование в Windows / А. П. Побегайло. – СПб.: БХВ-Петербург, 2005. – 1056 с.

5. Стивенс, У. Р. Протоколы TCP/IP: практическое руководство / У. Р. Стивенс. – СПб.: Невский диалект, 2003. – 672 с.

6. Чапел, Л. TCP/IP. Учебный курс / Л. Чапел, Э. Титтел. – СПб.: БХВ-Петербург, 2003. – 976 с.

7. Кенин, А. Самоучитель системного администратора / А. Кенин. – СПб.: БХВ-Петербург, 2006. – 464 с.

8. Айлебрехт, Л. Web-сервер Apache / Л. Айлебрехт. – Минск: Новое знание, 2002. – 592 с.

9. Apache Tomcat для профессионалов / А. Бакор [и др.]. – М.: КУДИЦ-ОБРАЗ, 2005. – 544 с.

10. Тимонович, Г. Л. Технология доступа к Интернет-ресурсам: практикум / Г. Л. Тимонович. – Минск: Акад. упр. при Президенте Респ. Беларусь, 2006. – 164 с.

11. Microsoft Windows XP. Руководство системного администратора / А. Г. Андреев [и др.]. – СПб.: БВХ-Петербург, 2004. – 848 с.

12. Арчер, Т. Visual C++ .NET. Библия пользователя / Т. Арчер, Э. Уайтчепел . – М.: Вильямс, 2003. – 1216 с.

13. Лишнер, Р. C++: справочник / Р. Лишнер. – СПб.: Питер, 2005. – 907 с.

14. Верма, Р. Д. Справочник по Win32 API / Р. Д. Верма. – М.: Горячая линия, 2005. – 551 с.

15. Таненбаум, Э. Операционные системы: разработка и реализация / Э. Таненбаум, А. Вудхалл . – СПб.: Питер, 2006. – 576 с.

16. Ахо, А. Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М.: Мир, 1979. – 536 с.

17. Ахо, А. Компиляторы: принципы, технологии и инструменты / А. Ахо, Р. Сети, Дж. Ульман. – М.: Вильямс, 2003. – 768 с.

ОГЛАВЛЕНИЕ

Предисловие 3
Введение 4

Глава 1. Взаимодействие процессов распределенного приложения 7
1.1. Предисловие к главе 7
1.2. Модель взаимодействия открытых систем 7
1.3. Архитектура «клиент – сервер» 11
1.4. Итоги главы 12

Глава 2. Стек протоколов TCP/IP 14
2.1. Предисловие к главе 14
2.2. Структура TCP/IP 15
2.3. Протоколы уровня доступа к сети 16
2.4. Протоколы межсетевого уровня 18
2.5. Протоколы транспортного уровня 25
2.6. Интерфейс внутренней петли 30
2.7. Интерфейсы сокетов и RPC 31
2.8. Основные службы TCP/IP 34
2.8.1. Служба и протокол DNS 35
2.8.2. Служба и протокол DHCP 36
2.8.3. NetBIOS over TCP/IP 37
2.8.4. Служба и протокол Telnet 38
2.8.5. Служба и протокол FTP 39
2.8.6. Электронная почта и протоколы SMTP, POP3, IMAP4 41
2.8.7. Протокол HTTP и служба WWW 43
2.9. Сетевые утилиты 45
2.10. Настройка TCP/IP в Windows 49
2.11. Итоги главы 51

Глава 3. Основы интерфейса Windows Sockets 53
3.1. Предисловие к главе 53

3.2. Версии, структура и состав интерфейса Windows Sockets 53
3.3. Коды возврата функций интерфейса Windows Sockets 55
3.4. Схемы взаимодействия процессов распределенного приложения 58
3.5. Инициализация библиотеки Windows Sockets 62
3.6. Завершение работы с библиотекой Windows Sockets ... 63
3.7. Создание и закрытие сокета 64
3.8. Установка параметров сокета 65
3.9. Переключение сокета в режим прослушивания 68
3.10. Создание канала связи 69
3.11. Обмен данными по каналу связи 72
3.12. Обмен данными без соединения 74
3.13. Пересылка файлов и областей памяти 77
3.14. Применение интерфейса внутренней петли 79
3.15. Использование широковещательных IP-адресов 79
3.16. Применение символических имен компьютеров 82
3.17. Итоги главы 84

Глава 4. Интерфейс Named Pipe 86
4.1. Предисловие к главе 86
4.2. Назначение и состав интерфейса Named Pipe 86
4.3. Создание именованного канала 89
4.4. Соединение клиентов с именованным каналом 92
4.5. Обмен данными по именованному каналу 95
4.6. Передача транзакций по именованному каналу 96
4.7. Определение состояния и изменение характеристик именованного канала 98
4.8. Итоги главы 100

Глава 5. Интерфейс Mailslots 101
5.1. Предисловие к главе 101
5.2. Назначение и состав интерфейса Mailslots 101
5.3. Создание почтового ящика 103
5.4. Соединение клиентов с почтовым ящиком 104
5.5. Обмен данными через почтовый ящик 106
5.6. Получение информации о почтовом ящике 107
5.7. Изменение интервала ожидания сообщения 108
5.8. Итоги главы 108

Глава 6. Разработка параллельного сервера 110

6.1. Предисловие к главе 110
6.2. Особенности разработки параллельного сервера 112
6.3. Структура параллельного сервера 113
6.4. Потоки и процессы в Windows 116
6.5. Синхронизация потоков параллельного сервера 123
6.6. Асинхронный вызов процедур 127
6.7. Использование ожидающего таймера 130
6.8. Применение атомарных операций 136
6.9. Блокирующий и не блокирующий режимы работы сокета 139
6.10. Использование приоритетов 144
6.11. Обработка запросов клиента 150
6.12. Применение механизма событий 153
6.13. Использование динамически подключаемых библиотек 156
6.14. Принципы разработки системы безопасности 163
6.15. Итоги главы 164

Глава 7. Практикум 166
7.1. Предисловие к главе 166
7.2. Практическая работа № 1. Сетевые утилиты 166
7.3. Практическая работа № 2. Обмен данными по TCP-соединению 169
7.4. Практическая работа № 3. Обмен данными без установки соединения 173
7.5. Практическая работа № 4. Применение широковещательных IP-адресов 175
7.6. Практическая работа № 5. Использование символических имен компьютеров 179
7.7. Практическая работа № 6. Работа с интерфейсом Named Pipe 181
7.8. Практическая работа № 7. Работа с интерфейсом Mailslots 183
7.9. Практическая работа № 8. Разработка параллельного сервера 185
7.10. Практическая работа № 9. Разработка клиента параллельного сервера 198
7.11. Практическая работа № 10. Разработка удаленной консоли 199
7.12. Выводы главы 200

Литература 201