Министерство науки и высшего образования Российской Федерации
КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Е.П. ЛУКАЩИК

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ

Учебное пособие

Краснодар
2021

УДК 004.415.25(075.8)
ББК 32.973 я 73
Л 84

Рецензенты:
Доктор технических наук, профессор
Л.А. Видовский
Кандидат физико-математических наук, доцент
М.Е. Бегларян

Лукащик, Е.П.
Л 84 Сетевое программирование: учебное пособие / Е.П. Лукащик; Министерство науки и высшего образования Российской Федерации, Кубанский государственный университет. – Краснодар: Кубанский государственный университет, 2021. – 216с. – 500 экз.
ISBN 978-5-8209-1928-2

Излагаются базовые сведения из теории компьютерных сетей и принципы разработки распределенных приложений в среде Windows. Рассматриваются стек протоколов TCP/IP, интерфейс Windows Socket, элементы параллельного программирования, необходимые для режимов множественного доступа. Изложение сопровождается многочисленными схемами и примерами. Для самостоятельной разработки сетевых проектов включены практические задания, предполагающие знание основ операционных систем и умение программирования на языке C++.

Адресуется студентам и магистрантам естественных специальностей для изучения базовых сетевых технологий, приобретения навыков разработки распределенных сетевых приложений. Работа поддержана грантом РФФИ № 19-01-00596.

УДК 004.415.25(075.8)
ББК 32.973. я 73

ISBN 978-5-8209-1928-2

ПРЕДИСЛОВИЕ

В области компьютеризации понятие *сетевое программирование* (*Network programming*), сильно схожее с понятиями *программирование сокетов* и *клиент-серверное программирование*, включает написание компьютерных программ, взаимодействующих с другими программами посредством компьютерной сети.

В предлагаемом издании раскрываются принципы сетевого программирования в среде Windows, излагаются необходимые при составлении сетевых приложений теоретические основы сетевых технологий, исследуются модели взаимодействия процессов сетевого приложения. Рассматриваются варианты архитектур распределенных вычислений, применяемых в сетевых технологиях.

В пособии рассматриваются следующие вопросы:

– теоретические основы построения и функционирования компьютерных сетей;

– стандартные модели взаимодействия процессов в распределенном приложении;

– клиент-серверная архитектура сетевых приложений;

– стек протоколов TCP/IP и основные принципы взаимодействия его компонентов, транспортные протоколы TCP и UDP, IP-протокол и IP-адресация;

– главные возможности интерфейса Windows Socket API, являющегося основой для построения распределенных приложений в среде TCP/IP;

– элементы параллельного программирования, необходимые при разработке сетевых приложений в режиме множественного доступа. Основные принципы построения и программирования параллельного сервера.

Пособие в большей степени ориентировано на приобретение практических навыков низкоуровневого сетевого программирования. С этой целью материалы разделов в качестве примеров сопровождаются исходными кодами на языке C++, реализующими описываемые подходы и концепции.

Для закрепления изученного материала предлагаются практические работы по разработке сетевых проектов. В качестве среды разработки приложений на языке C++ рекомендуется

Microsoft Visual Studio. Тестирование распределенных приложений можно проводить на отдельном компьютере, используя интерфейс внутренней петли. Однако полное выполнение заданий предусматривает освоение навыков работы в локальной сети и выход в сеть Internet. Подбор материала для практической части учитывает как актуальность тем, так и реальные возможности проведения лабораторных работ.

Содержание пособия по тематике тесно связано с курсами «Компьютерные сети», «Системное программирование», «Операционные системы», «Аппаратные и программные средства Web», «Сетевая безопасность», а также может служить в качестве основы для дальнейшего изучения информационных сетевых технологий более высокого уровня.

ВВЕДЕНИЕ

Компьютерные сети являются неотъемлемой частью современных информационных технологий. Значительная часть разрабатываемого и эксплуатируемого программного обеспечения так или иначе связана с работой в сетях.

Компьютерная сеть представляет собой *распределенную* вычислительную систему, т. е. сетевые узлы, производящие обработку и / или хранение данных, могут находиться на значительных расстояниях друг от друга. В силу чего сетевое приложение независимо от того, для решения прикладной или системной задачи (сетевого сервиса) оно предназначено, как правило, проектируется как *распределенное*, т. е. разбивается на процессы (независимые программные единицы), которые могут исполняться на различных сетевых узлах. Разбиение приложения на модули обычно происходит на основе клиент-серверной модели. Клиентом называется модуль, инициирующий запрос на сетевые ресурсы, сервером – модуль, ожидающий запросов от клиентов. Связь между процессами происходит путем передачи сообщений по каналам связи. Представление значительной части сетевых программ в виде распределенных приложений обусловлено желанием в полной мере использовать потенциальные возможности сетей по распараллеливанию вычислений.

Предлагаемое издание ориентировано на освоение навыков программирования для сетей TCP/IP. Представлены теоретические сведения по компьютерным технологиям, необходимые для сетевого практикума с использованием протоколов данного стека. Исследуется концепция сокетов, дающая понятную и удобную модель для разработки сетевых приложений. Рассмотрены различные варианты реализации интерфейса прикладного программирования, основанного на модели сокетов. Более полно раскрываются возможности программного интерфейса Windows Sockets, предоставляющего в распоряжение сетевых программистов удобные средства организации передачи данных с использованием дейтаграмм и виртуальных соединений между узлами сети. Описываются современные надстройки Winsock с целью адаптации

библиотеки к внедрению в сетевую практику новых версий протоколов стека TCP/IP.

Большое внимание уделяется организации взаимодействия процессов сетевых распределенных приложений. Для реализации необходимого большинству сетевых приложений многопользовательского режима исследуются элементы параллельного программирования. Показано использование механизма многопоточности при проектировании параллельного сервера.

Излагаются основные принципы разработки сетевых служб (сервисов). На примере построения чат-сервиса продемонстрировано применение изложенных методик в практических проектах.

При изложении материала в пособии предполагается, что читатель имеет навыки программирования на языке C++, обладает знаниями основ теории операционных систем.

1. ОСНОВЫ СЕТЕВЫХ ТЕХНОЛОГИЙ

1.1. Основные определения

Компьютерная сеть представляет собой совокупность узлов, или хостов (hosts), соединенных каналами передачи данных. Часть узлов предназначена для взаимодействия с конечными пользователями услуг сети – это терминальные узлы или терминалы. Если сеть достаточно велика и сложна, то выделяются специализированные управляющие (служебные) узлы для обеспечения ее функционирования: маршрутизаторы, шлюзы, серверы и т. д.

Каналы связи и служебные узлы образуют базовую сеть передачи данных (СПД). Базовая СПД – основа вычислительной сети. Она, как правило, более сложна, более консервативна и более универсальна, чем подключенные к ней устройства пользователей (терминалы), то есть одна и та же базовая СПД используется многими различными пользователями и их группами, которые могут решать с ее помощью различные задачи (рис.1).

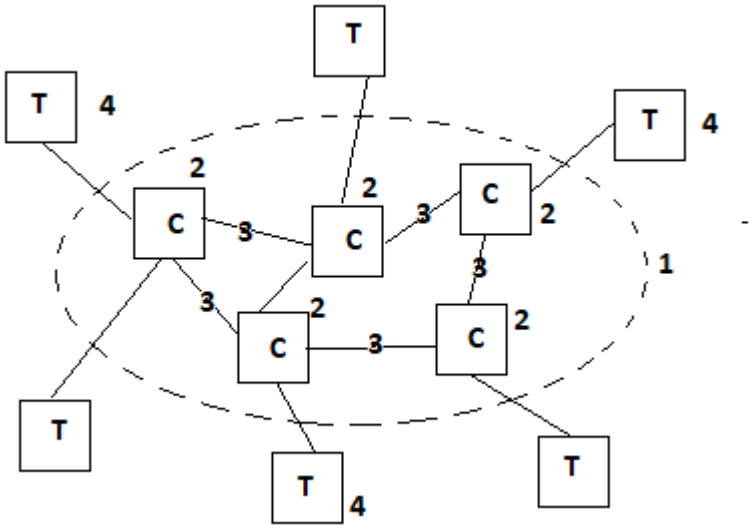


Рис.1. Общая структура сети: 1– базовая СПД, 2 – управляющие узлы, 3 – каналы связи, 4 – оконечные узлы

В свою очередь внутри базовой СПД могут выделяться магистральные каналы с большой пропускной способностью и

обслуживающие их узлы – опорная сеть (backbone). К опорной сети подключаются уже не отдельные узлы, а целые подсети. Иерархия построения позволяет оптимизировать и упорядочить функционирование сети: если передаваемый поток данных не может быть замкнут в пределах подсети, его целесообразно передавать не через соседние подсети, а через опорную сеть.

Как разновидность вычислительных систем компьютерная сеть характеризуется следующими признаками:

– неоднородность: состоит из разнотипных компонентов с различными функциями;

– децентрализованное управление: каждый компонент сети достаточно самостоятелен и сам контролирует свое функционирование;

– распределенность в пространстве: компоненты сети территориально удалены друг от друга.

Важным признаком компьютерных сетей является то, что в сетях основное внимание уделяется размещению, поиску и транспортировке данных в пределах сети безотносительно к их обработке конкретными узлами. В силу этого сетям менее всего свойственно единство цели: одна и та же сеть может служить для решения многих задач, иногда не связанных друг с другом.

1.2. Классификация сетей

Помимо указанных общих квалификационных признаков вычислительные сети принято подразделять по некоторым специфическим категориям, которые рассматриваются далее.

По типу управления сетью

Как отмечалось, сеть может быть представлена состоящей из своего рода ядра – базовой СПД – и подключенных к ней терминалов. В простейшем случае СПД состоит только из каналов передачи данных, и тогда все узлы сети играют роль терминалов, которые равноправны между собой и более или менее однородны по характеристикам. Такая сеть называется *одноранговой*. Терминалы, или рабочие станции, такой сети могут предоставлять сетевой доступ к своим ресурсам и выполнять некоторые сервисные

функции в рамках всей сети, но это не становится их основным назначением. Одноранговые сети просты в построении и обслуживании, но возможности наращивания размеров и расширения круга решаемых задач для них ограничены.

Увеличение требований к сети приводит к необходимости выделения специализированных узлов, или серверов, выполняющих только служебные функции и предоставляющих свои ресурсы остальным пользователям, причем более надежно и эффективно, чем рабочие станции, делающие это «по совместительству». Функции сервера могут касаться как обслуживания пользователей, так и обеспечения работы самой сети; во втором случае его правильнее относить уже к базовой СПД. Расширение функционала сети, как правило, на практике сопровождается выделением отдельного сервера для *централизованного* управления работой сети. Сети с централизованным управлением сложнее, но обладают гораздо большими административными возможностями.

По территориальному признаку

Как распределенная вычислительная система компьютерная сеть может состоять из различного, возможно, большого числа узлов, размещенных на определенной территории, и эти количественные показатели существенно влияют на многие особенности ее построения и функционирования. Традиционно по территориальному и структурному признаку сети делятся на локальные и глобальные, между которыми возможны и промежуточные типы.

Локальные сети объединяют относительно небольшое количество близких по назначению и характеристикам рабочих мест на небольшой территории, используя простое сетевое оборудование и дешевые, но достаточно быстродействующие каналы передачи данных. Обычно самостоятельные подсети в этом случае не выделяются, и задача выбора путей транспортировки информации не возникает. Локальные сети характеризуются простотой, малыми затратами на постройку и эксплуатацию, однако имеют ограниченные возможности.

Глобальные сети объединяют не столько отдельных абонентов, сколько сети более низкого ранга, или подсети, и охватывают большую территорию (в настоящее время – практически весь

земной шар и ближайшее околоземное пространство). Каждая из подсетей сохраняет самостоятельность и может также состоять из подсетей. Другая важная особенность глобальной сети – состав ее участников и круг выполняемых ею задач заранее не определены, поэтому возможность ее перестройки, расширения и развития необходимо предусмотреть заранее.

Региональные сети занимают промежуточное положение: они сложнее и крупнее локальных, имеют внутреннюю иерархию, но не достигли масштабов глобальных. Примерами могут служить сети крупных предприятий, ведомств, административных единиц.

По функциональному назначению

Имеет место также альтернативная классификация, рассматривающая в комплексе признаков назначение сетей: сети рабочих групп, сети отделов, сети кампусов и корпоративные.

Сети рабочих групп, или отделов, предназначены для обслуживания небольших обособленных групп пользователей, решающих общие задачи или относящихся к одному подразделению. Основная цель такой сети – обеспечение информационного обмена между пользователями и совместное использование ресурсов (принтера, архива файлов), а главные характеристики – простота построения и обслуживания. Специальных средств эффективного структурирования обычно не требуется, хотя сеть отдела может включать несколько подсетей отдельных рабочих групп.

Сети кампусов возникли как объединение более мелких сетей, что первоначально имело место в студенческих городках при университетах. В настоящее время к этому классу относятся и сети, объединяющие сети отделов одного предприятия, находящиеся в одном или нескольких близко расположенных зданиях. Сеть кампуса должна обеспечить обмен информацией между отделами и совместный доступ к ресурсам без выхода в глобальную сеть, т. е. используя соединения только внутри этой сети. Соединение с глобальной сетью обычно также имеется, но рассматривается как один из разделяемых ресурсов.

Корпоративные сети, или сети масштаба предприятия, отличаются от прочих как количеством узлов и подсетей, так и территориальной удаленностью их друг от друга, вследствие чего связь

между ними может осуществляться с использованием глобальных соединений – корпоративная сеть использует существующую глобальную сеть как среду транспортировки данных. В этом случае особое значение приобретают функции управления неоднородной системой, контроля пользователей и обеспечения безопасности информационного обмена.

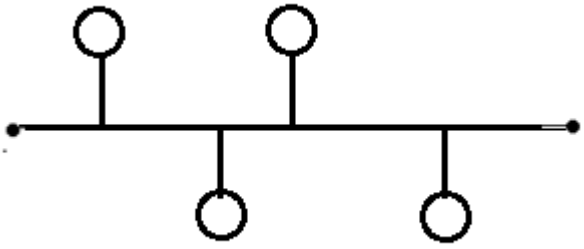
По топологии сетей

Важным квалификационным признаком сетей является топология – геометрическое строение сети, способ взаимного расположения улов и соединений между ними. Топология влияет на многие технические и эксплуатационные характеристики сети. Выделяют следующие базовые топологии:

(a) топология «***точка-точка***»

– простое соединение двух узлов. В данной системе адресация узлов не требуется, но может поддерживаться с целью унификации. К соединениям такого рода относится, например, временная связь между двумя компьютерами посредством любого подходящего интерфейса. В более сложных сетях подобное соединение может устанавливаться между маршрутизаторами опорной сети или между оконечными узлами и базовой СПД при использовании некоторых типов каналов;

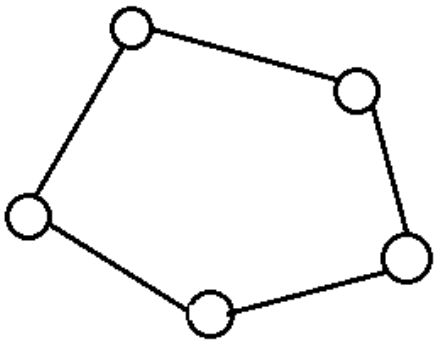
(b) ***шинная*** топология:



Все узлы однотипным образом присоединяются к единому каналу передачи данных (моноканалу), пропускная способность которого разделяется между всеми пользователями. Реализация такой сети

проста и экономична. К недостаткам следует отнести пониженные надежность (как правило, повреждение шины выводит из строя всю сеть, однако исправность отдельных узлов не сказывается на работоспособности сети) и пропускную способность (в каждый момент времени может функционировать не более одного передатчика). Как следствие, шинная топология эффективна при небольшом количестве узлов или относительно невысокой интенсивности передачи. В основном она наблюдается у сетей на коаксиальном кабеле и для беспроводных технологий (версии Ethernet);

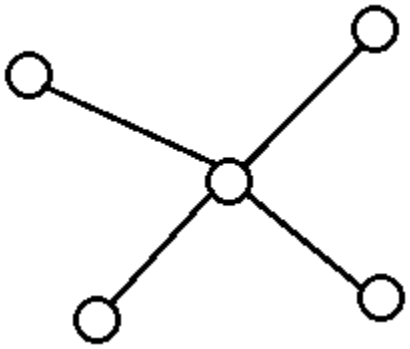
(c) ***кольцевая*** топология:



напоминает шинную, но единый канал замкнут в кольцо. Так как нормальная передача возможна только в одном направлении, соединения узлов с кольцом не могут быть пассивными: узел включается в разрыв канала и имеет вход и выход, передача данных между которыми «внутри» узла обеспечивается им самим. Для двунаправленной передачи требуется второй параллельный канал. Такой подход характерен для оптоволоконных линий связи, но может применяться и в обычных проводных. Работоспособность *кольца* зависит от состояний как канала, так и всех узлов, что сказывается на надежности, а монтаж СПД может оказаться сложным и дорогостоящим. Основным преимуществом кольцевой технологии является удобство обеспечения гарантированной пропускной способности, практически не зависящей от нагрузки на сеть, вследствие чего она эффективна для высокоскоростных магистралей, связывающих подсети. Основной технологией, базирующейся на данной топологии, является TokenRing. Для современных оптоволоконных линий связи применяется основанная на TokenRing технология FDDI, развивающая и совершенствующая

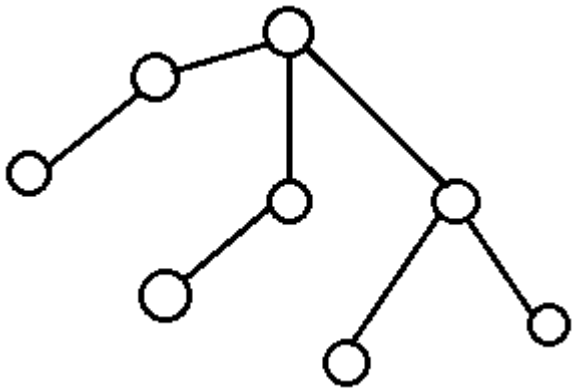
ее основные идеи. Сеть FDDI строится на основе двух оптоволоконных колец, которые образуют основной и резервный пути передачи данных между узлами сети;

(d) ***звездообразная*** (или радиальная) топология:



в сети выделяется центральный, обычно специализированный узел, к которому присоединяются все остальные узлы отдельными линиями связи. Условно можно рассматривать ее как множество соединений «точка-точка», сходящихся в одном узле. Прокладка такой сети проста и технологична, стоимость зависит от суммарной длины линий. Пропускная способность может быть высокой, так как периферийные узлы обмениваются данными с центральным независимо друг от друга. Критические факторы работоспособности сети – безотказность и производительность центрального узла, в то время как состояния периферийных узлов сказываются на ней незначительно. Существуют реализации на любых средах передачи, но наиболее характерна для обычных проводных, например на витой паре;

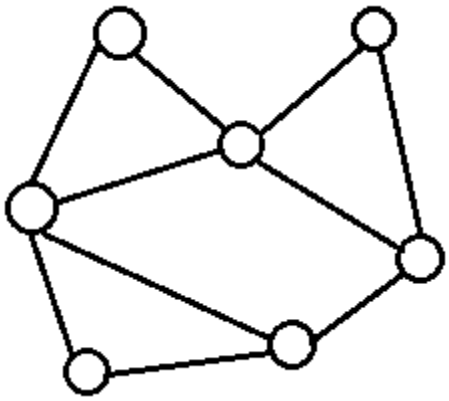
(e) ***древовидная*** топология



– развитие звездообразной, от которой отличается наличием более чем двух уровней подчиненности: любой узел может стать центральным для узлов следующего уровня иерархии. Часть узлов

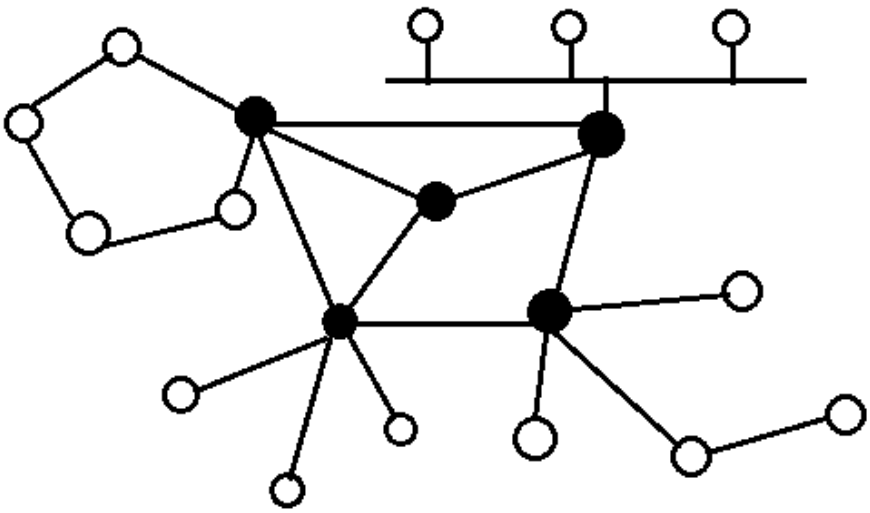
могут быть специализированными и служить исключительно для поддержания соединений (различные коммутаторы). Такая топология позволяет распределить управляющие функции и трафик, разгружая «корневой» узел, использовать каналы с различной пропускной способностью и в результате более эффективно наращивать размеры сети.

(f) ***сетевая*** топология



– дальнейшее развитие древовидной при отступлении от принципа иерархического подчинения и допущении произвольных связей между узлами. Наличие альтернативных путей доставки данных повышает пропускную способность и надежность системы за счет возможности перераспределения потоков, однако при этом возникает достаточно сложная задача маршрутизации. Как результат, сетевая топология практически не приемлема для локальных сетей, но в сложных сетях эффективна для базовой СПД, маршрутизаторы внутри которой фактически связываются попарно соединениями «точка-точка».

(g) ***смешанная*** топология



– сочетание перечисленных базовых топологий. Естественным образом возникает в глобальных сетях, объединяющих подсети различной структуры.

1.3. Принципы передачи данных

Различают два основных способа организации передачи данных в сетях: с установлением соединения и без установления соединения.

При ***передаче без установления соединения*** данные передаются в виде законченных самостоятельных блоков – дейтаграмм (datagram). Каждая дейтаграмма доставляется к пользователю по произвольному маршруту и независимо от других дейтаграмм. Подтверждение о получении (квитирование) не предусматривается, в силу чего не гарантируется ни порядок следования дейтаграмм, ни единственность доставленного экземпляра, ни сам факт доставки. Контролируются обычно только искажения каждой отдельной дейтаграммы. Такой способ передачи прост, экономичен, но для многих применений недостаточно надежен. Основное применение дейтаграммной передачи – однократные сообщения ограниченной длины при условии, что надежность доставки не слишком критична.

Передача с установлением соединения (потоковое соединение) обладает большими возможностями. Главное отличие от дейтаграммной передачи – обеспечение целостности и упорядоченности потока передаваемых данных. Независимо от способа организации потока порции данных доставляются получателю строго в том порядке, в котором они были отправлены, а прерывание потока своевременно распознается. Это достигается за счет нумерации порций данных и организации встречного потока подтверждений о получении (квитанций). Таким образом образуется виртуальный канал передачи данных, для прикладных программ близкий по своим свойствам файлу или потоку ввода–вывода. Вместе с тем потоковая передача сложнее, требует специальных процедур установления соединения и дополнительных затрат на контроль его состояния, создает дополнительную нагрузку на линии связи в виде встречного потока квитанций. Для большинства задач,

связанных с обменом значительными объемами данных, предпочтительным является взаимодействие с установлением соединения.

1.4. Методы коммутации

Если контакт между двумя узлами необходимо поддерживать в течение длительного времени, как в случае передачи с установлением соединения, возникает задача коммутации – распределения имеющихся физических каналов передачи данных для создания каналов виртуальных. Различают три основных метода коммутации: коммутация каналов, сообщений и пакетов.

Коммутация каналов (circuit switching) – предоставление виртуальному каналу цепочки физических каналов, сохраняющейся на все время жизни соединения и находящейся в монопольном владении этой парой абонентов. Такой канал в простейшем случае представляет собой отдельную линию связи, но может быть организован и с помощью временного или частотного разделения сигнала в общей для множества каналов среде передачи. Типичным примером первого служит обычная коммутируемая телефонная сеть, второго – разделение «общего» эфира между радиопередатчиками. Коммутация каналов позволяет достичь максимальной для конкретного типа канала производительности, предоставляемой отдельно взятой паре абонентов. К недостаткам следует отнести непроизводительный простой канала в случае, если обмен не является непрерывным, и длительную процедуру установления соединения или его восстановления после сбоя.

Коммутация сообщений (message switching) отличается тем, что поток данных разбивается на отдельные завершенные *сообщения* и реальное (физическое) соединение создается только на ограниченное время (для передачи одного сообщения), в остальное время те же ресурсы могут быть использованы для передачи сообщений других потоков. Неудобством такого подхода оказалась переменная и часто слишком большая длина сообщений, затрудняющая создание коммуникационного оборудования, и в настоящее время этот вид коммутации практически не используется, однако он послужил прототипом современной коммутации пакетов.

Коммутация пакетов (packet switching) – наиболее часто используемый метод. Основная идея состоит в разбиении всех передаваемых данных на порции одинакового или хотя бы ограниченного небольшого размера – *пакеты*. Каждый пакет доставляется как дейтаграмма, т. е. независимо от остальных пакетов и, возможно, по собственному маршруту, благодаря чему достигается более равномерная загрузка сети и большая устойчивость против сбоев – доставка может быть выполнена другим маршрутом, и повторить при этом придется лишь отдельный пакет. Размер пакета выбирается удобным для коммутаторов, что упрощает и удешевляет аппаратуру. К проблемам пакетной коммутации могут быть отнесены большая доля служебной информации в трафике, большая нагрузка на маршрутизаторы и необходимость сложных процедур сборки пакетов. Однако преимущества метода оказываются в большинстве случаев более существенными.

2. СТРУКТУРА СЕТЕВЫХ ПРИЛОЖЕНИЙ

В сетевых технологиях широко используются термины: «распределенная архитектура», «распределенная обработка данных», «распределенные вычисления», «распределенная система» и т. п. 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 каналы передачи данных. Процессы – это программы, работающие на компьютерах и использующие вычислительные ресурсы для решения задач.

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 одном или разных компьютерах и обменивающиеся данными.

Взаимодействие процессов внутри приложения реализуется, как правило, посредством обмена информацией между компьютерами. Организация подобных взаимодействий напрямую влияет на работоспособность приложения. В простейшем случае, когда компьютеры не объединены в сеть, это может быть сделано с помощью магнитного носителя, иначе может быть использована вычислительная сеть или другие специальные устройства.

В большинстве случаев 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. Проблема совместного использования ресурсов может возникнуть, например, уже при использовании файла данных двумя или более процессами одного или нескольких приложений. Если при этом один процесс записывает (или редактирует) данные, а другой читает, то требуются специальные механизмы для обеспечения согласованности этих операций. Аналогичная проблема возникает в многозадачных

операционных системах. Особенно остро ощущается необходимость в синхронизации обменов данными процессами приложения, работающих асинхронно (независимо друг от друга) на разных компьютерах сети.

Как результат, на практике разработчик распределенного приложения сталкивается с целым рядом проблем, основными среди которых являются:

1) обмен данными между процессами в распределенном приложении;
2) синхронизация процессов в распределенном приложении;
3) совместное использование ресурсов процессами распределенного приложения.

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

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 заключается 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.

Для определения 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.

2.1. Клиент-серверная архитектура

Как правило, компьютеры и программы, входящие в состав информационной системы, не являются равноправными. Некоторые из них владеют ресурсами (файловая система, процессор, принтер, база данных и т. д.), другие имеют возможность обращаться к этим ресурсам. Компьютер (или программу), управляющий ресурсом, называют сервером этого ресурса (файл-сервер, сервер базы данных, вычислительный сервер...). Клиент и сервер какого-либо ресурса могут находиться как на одном компьютере, так и на различных компьютерах, связанных сетью (рис. 2).

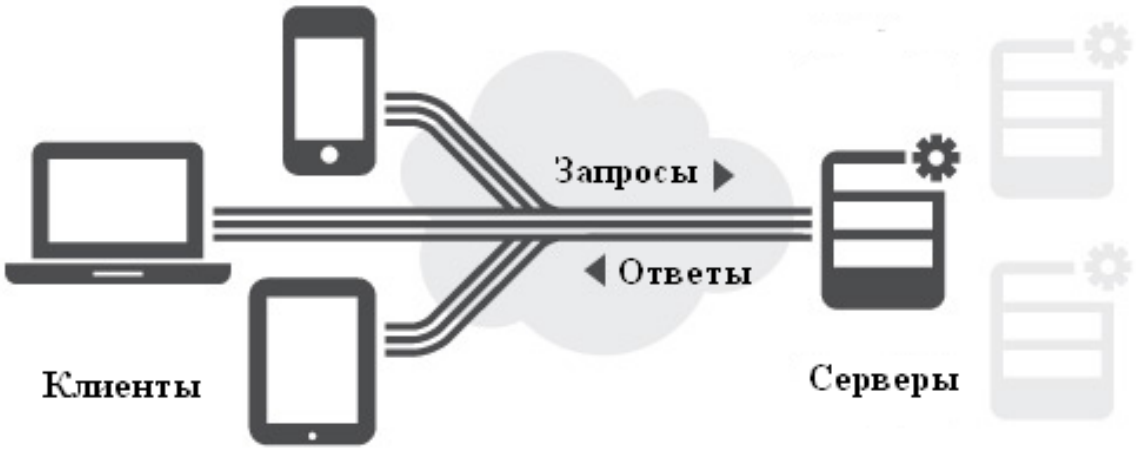


Рис. 2. Клиент-серверное взаимодействие

Анализ функционала произвольного сетевого приложения позволяет выделить три группы функций, ориентированных на решение различных подзадач:

– функции ввода и отображения данных (обеспечивают взаимодействие с пользователем);

– прикладные функции, характерные для данной предметной области;

– функции управления ресурсами (файловой системой, базой данных и т. д.).

Выполнение этих функций в основном обеспечивается программными средствами, которые можно представить в виде взаимосвязанных компонентов (рис. 3), где:

– ***компонент представления*** отвечает за пользовательский интерфейс;

– ***прикладной компонент*** реализует алгоритм решения конкретной задачи;

– ***компонент управления*** ресурсом обеспечивает доступ к необходимым ресурсам.

Рис. 3. Компоненты сетевого приложения

Автономная система (компьютер, не подключенный к сети) представляет все эти компоненты как на различных уровнях (ОС, служебное ПО и утилиты, прикладное ПО), так и на уровне приложений (нехарактерно для современных программ). Так же и сеть – она представляет все эти компоненты, но в общем случае распределенные между узлами. Задача сводится к обеспечению сетевого взаимодействия между этими компонентами.

В основе большинства современных информационных систем лежит архитектура **«клиент-сервер».** Данная архитектура распределенного программного приложения определяет общие принципы организации взаимодействия в сети, где имеются

серверы, узлы-поставщики некоторых специфичных функций (сервисов), и *клиенты*, потребители этих функций.

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, во-вторых, для обмена с клиентом данными. В большинстве случаев основная вычислительная нагрузка ложится на сервер, а клиент лишь обеспечивает интерфейсом пользователя с сервером.

По методу обслуживания серверы подразделяются на последовательные и параллельные. Параллельный сервер предназначен для обслуживания нескольких клиентов одновременно и поэтому использует специальные средства операционной системы, позволяющие распараллеливать обработку нескольких клиентских запросов.

Последовательный сервер, как правило, обслуживает запросы клиентов поочередно, заставляя клиентов ожидать своей очереди на обслуживание, или просто отказывает клиенту.

Практические реализации такой архитектуры называются **клиент-серверными технологиями**. Каждая технология определяет собственные или использует имеющиеся правила взаимодействия между клиентом и сервером, которые называются *протоколом обмена (протоколом взаимодействия)*.

Двухзвенная архитектура

В любой сети (даже одноранговой), построенной на современных сетевых технологиях, присутствуют элементы клиент-серверного взаимодействия, чаще всего на основе **двухзвенной архитектуры**. Двухзвенной (***two-tier, 2-tier***) она называется из-за необходимости распределения *трех базовых компонент* (рис.4) между *двумя узлами* (клиентом и сервером).

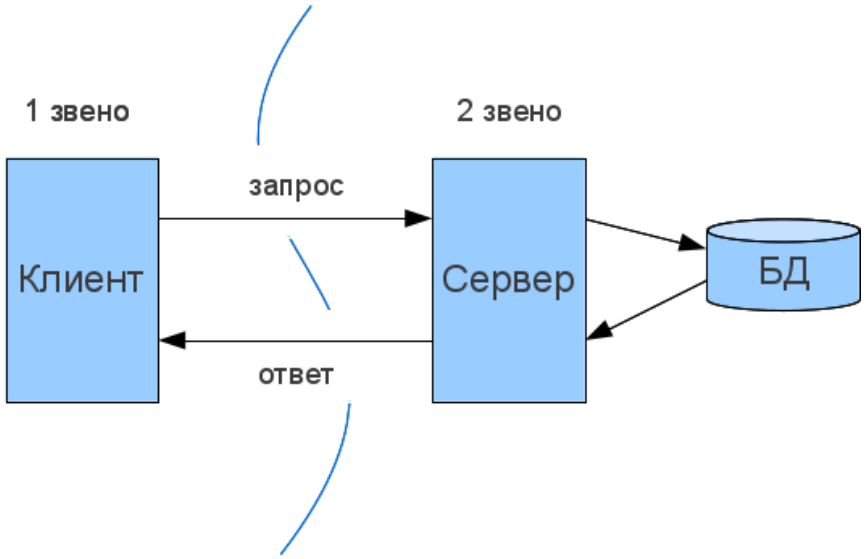


Рис. 4. Двухзвенная клиент-серверная архитектура

Двухзвенная архитектура используется в клиент-серверных системах, где сервер отвечает на клиентские запросы напрямую и в полном объеме, при этом используя только собственные ресурсы. Иначе говоря, сервер не вызывает сторонних сетевых приложений и не обращается к сторонним ресурсам для выполнения какой-либо части запроса.

Расположение отмеченных функциональных компонентов на стороне клиента или сервера задает определенную модель их взаимодействия в рамках двухзвенной архитектуры. Основные модели представлены на рис.5.

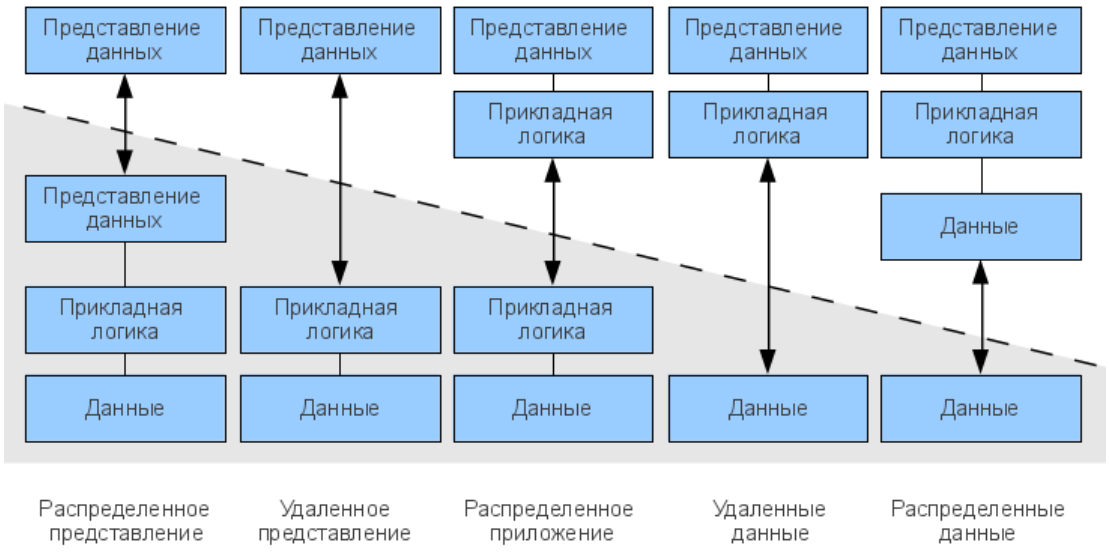


Рис. 5. Распределение функциональных компонентов
в моделях двузвенной архитектуры

Исторически первой появилась модель распределенного представления данных (модель ***сервер терминалов***). Она реализовывалась на универсальной ЭВМ (мэйнфрейме), выступавшей в роли сервера, с подключенными к ней алфавитно-цифровыми терминалами. Пользователи выполняли ввод данных с клавиатуры терминала, которые затем передавались на мэйнфрейм, где и выполнялась их обработка, включая формирование «картинки» с результатами. Эта «картинка» и возвращалась пользователю на экран терминала.

С появлением персональных компьютеров и локальных сетей была реализована модель файлового сервера, представлявшего доступ к файловым ресурсам, в том числе и к удаленной базе данных. В этом случае выделенный узел сети является файловым сервером, на котором размещены файлы базы данных. На клиентах выполняются приложения, в которых совмещены компонент представления и прикладной компонент (СУБД и прикладная программа), использующие подключенную удаленную базу как локальный файл. Протоколы обмена при этом представляют набор низкоуровневых вызовов операций файловой системы. Такая модель показала свою неэффективность ввиду того, что при активной работе с таблицами БД возникает большая нагрузка на сеть. Частичным решением является поддержка тиражирования (репликации) таблиц и запросов. В этом случае, например при изменении данных, обновляется не вся таблица, а только модифицированная ее часть.

С появлением специализированных СУБД появилась возможность реализации другой модели доступа к удаленной базе данных – модели сервера баз данных. В этом случае ядро СУБД функционирует на сервере, прикладная программа – на клиенте, а протокол обмена обеспечивается с помощью языка SQL. Такой подход по сравнению с файловым сервером ведет к уменьшению загрузки сети и унификации интерфейса «клиент–сервер». Однако сетевой трафик остается достаточно высоким, кроме того, по-прежнему невозможно удовлетворительное администрирование приложений, поскольку в одной программе совмещаются различные функции.

С разработкой и внедрением на уровне серверов баз данных механизма хранимых процедур появилась концепция *активного*

сервера БД. В этом случае часть функций прикладного компонента реализована в виде хранимых процедур, выполняемых на стороне сервера. Остальная прикладная логика выполняется на клиентской стороне. Протокол взаимодействия – соответствующий диалект языка SQL.

Реализация прикладного компонента на стороне сервера представляет следующую модель – сервер приложений. Перенос функций прикладного компонента на сервер снижает требования к конфигурации клиентов и упрощает администрирование, но представляет повышенные требования к производительности, безопасности и надежности сервера.

Трехзвенная архитектура

Еще одна тенденция в клиент-серверных технологиях связана со все большим использованием распределенных вычислений. Они реализуются на основе модели сервера приложений, где сетевое приложение разделено на две или более частей, каждая из которых может выполняться на отдельном компьютере. Выделенные части приложения взаимодействуют друг с другом, обмениваясь сообщениями в заранее согласованном формате.

В этом случае двухзвенная клиент–серверная архитектура становится ***трехзвенной (three–tier, 3–tier)*** (рис. 6):

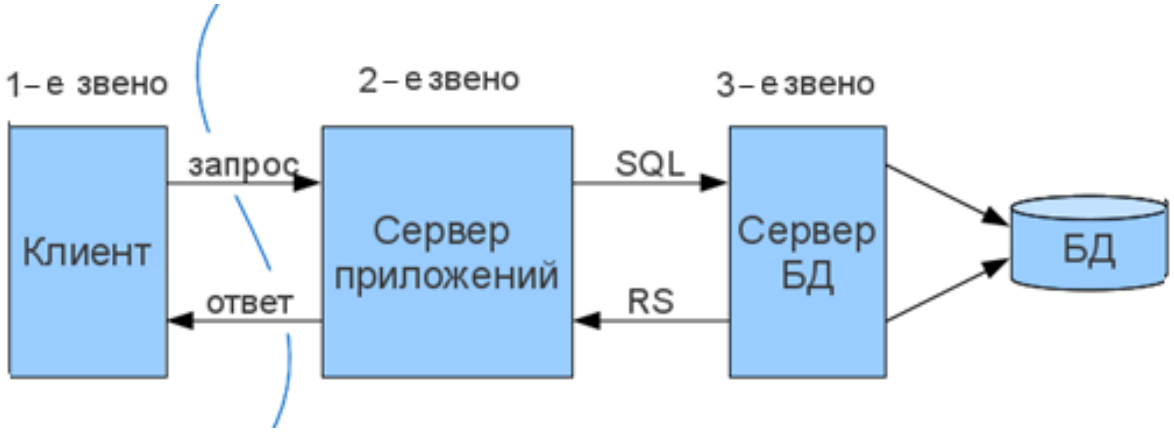


Рис. 6. Трехзвенная клиент-серверная архитектура

Как правило, третьим звеном в трехзвенной архитектуре становится сервер приложений, т. е. компоненты распределяются следующим образом:

1. Представление данных – на стороне клиента.
2. Прикладной компонент – на выделенном сервере приложений (как вариант, выполняющем функции промежуточного ПО).
3. Управление ресурсами – на сервере БД, который и представляет запрашиваемые данные.

Трехзвенная архитектура может быть расширена до ***многозвенной (N–tier, Multi–tier*)** путем выделения дополнительных серверов, каждый из которых будет представлять собственные сервисы и пользоваться услугами прочих серверов разного уровня. Абстрактный пример многозвенной модели приведен на рис. 7.

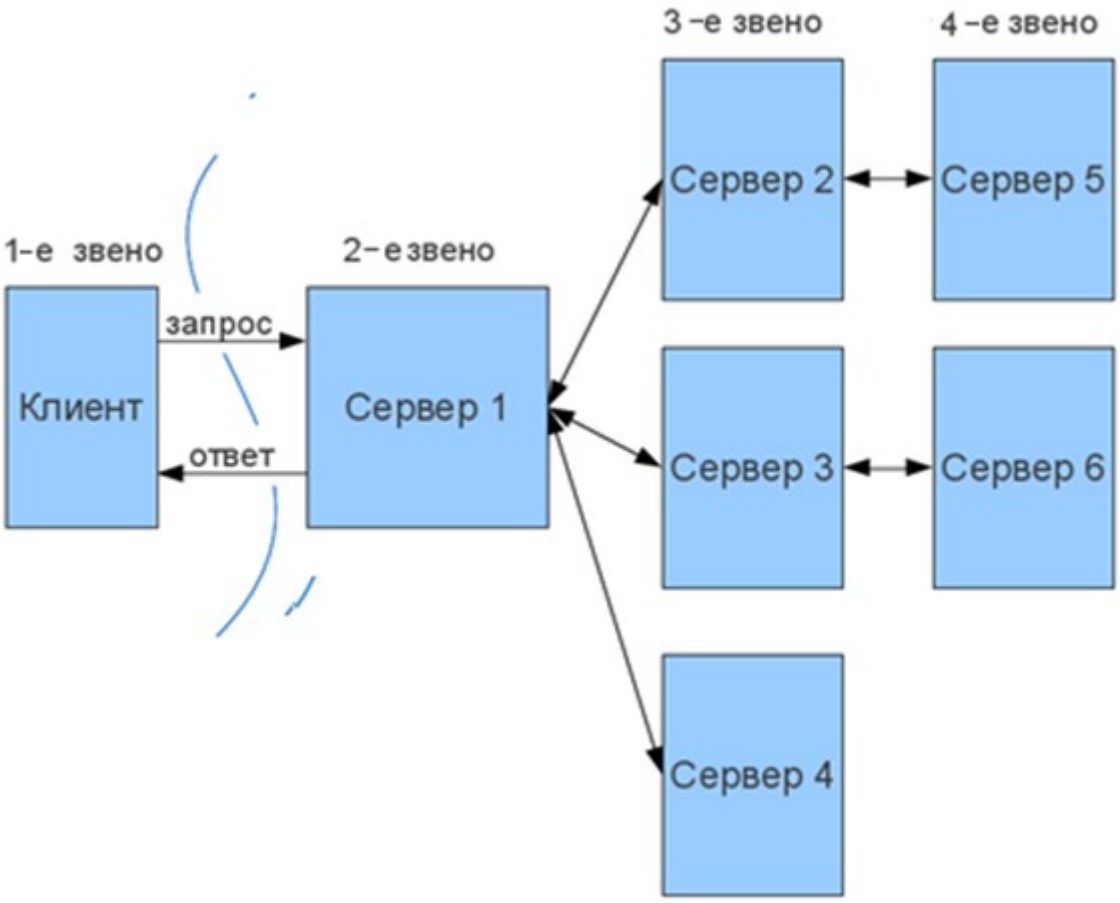


Рис. 7. Многозвенная (n-tier) клиент-серверная архитектура

Для доступа к тем или иным сетевым сервисам используются клиенты, возможности которых характеризуются понятием ***«толщины***». Оно определяет конфигурацию оборудования и программное обеспечение, имеющиеся у клиента. Рассмотрим возможные граничные значения.

«Тонкий» клиент». Этот термин определяет клиента, вычислительных ресурсов которого достаточно лишь для запуска необходимого сетевого приложения через web-интерфейс. Пользовательский интерфейс такого приложения формируется средствами *статического* HTML (выполнение JavaScript не предусматривается), вся прикладная логика выполняется на сервере. Для работы тонкого клиента достаточно лишь обеспечить возможность запуска web-браузера, в окне которого и осуществляются все действия. По этой причине Web-браузер часто называют «универсальным клиентом».

«Толстый» клиент». Таковым является рабочая станция или персональный компьютер, работающие под управлением собственной дисковой операционной системы и имеющие необходимый набор программного обеспечения. К сетевым серверам «толстые» клиенты обращаются в основном за дополнительными услугами (например, доступ к Web-серверу или корпоративной базе данных).

В последнее время все чаще используется еще один термин: ***«rich»-client***. «*Rich*»-клиент своего рода компромисс между «толстым» и «тонким» клиентом. Как и «тонкий» клиент, «*rich*»-клиент также представляет графический интерфейс, описываемый уже средствами XML и включающий некоторую функциональность толстых клиентов (например, интерфейс *drag-and-drop*, вкладки, множественные окна, выпадающие меню и т. п.). Прикладная логика «*rich*»-клиента реализована на сервере. Данные отправляются в стандартном формате обмена, на основе того же XML (протоколы SOAP, XML-RPC) и интерпретируются клиентом.

Итак, ***основная идея архитектуры «клиент-сервер» состоит в разделении сетевого приложения на несколько компонентов***, каждый из которых реализует специфический набор

сервисов. Компоненты такого приложения могут выполняться на разных компьютерах, реализуя серверные и / или клиентские функции. Это позволяет повысить надежность, безопасность и производительность сетевых приложений и сети в целом.

2.2. Пример клиент-серверного приложения с файлом в качестве общей среды

Клиент-серверные технологии могут применяться и при разработке приложений, не рассчитанных для использования в сети. В данном разделе излагаются принципы построения приложения в виде двух компонент (программы-сервера и программы-клиента), функционирующих на одном компьютере и использующих для обмена данными файлы.

Для общения клиента и сервера разработан собственный пользовательский протокол, определяющий формат запроса клиента и формат ответа сервера.

В качестве общей среды передачи информации между клиентом и сервером используются два бинарных файла: один для запросов клиента серверу (`REQUEST`), другой для ответов сервера клиенту (`ANSWER`).

Факт появления в файле новой информации определяется путем сравнения текущего и предыдущего размеров файла.

Для примера представлено приложение-сервис, реализующее «медицинский центр», основная задача которого на основе предоставляемых сведений о пациенте (имя, рост и вес) определять для него индекс массы тела.

Предлагаемые далее тексты демонстрируют логику взаимодействия программных компонент в клиент-серверном приложении.

Программа-сервер

Код программы-сервера на языке программирования C++ может быть следующим:

```
#include <iostream>
#include <fstream>
```

```
    #include <windows.h>
    using namespace std;
    struct Person
    {
        char name[25];   //имя
        int height;     //рост
        int weight;     //вес
    }B;
    int answer;
    long size_pred;

    int main()
    {
    ifstream fR;
    ofstream fA;
    setlocale(LC_ALL, "rus");
    char* nameR = "C:\REQUEST.bin";
//файл запросов клиентов
    char* nameA = "C:\ANSWER.bin";
//файл ответов сервера

    cout<< "server is working"<< endl;

// начальные установки
    fR.open(nameR, ios::binary);
//открытие файла REQUEST
    fR.seekg(0, ios::end);
    size_pred = fR.tellg();
//стартовая позиция сервера в файле REQUEST
    fR.close();

// начало работы сервера
    while (true)
    {
    fR.open(nameR, ios::binary);
//открытие файла REQUEST
```

```
    fR.seekg(0, ios::end);
//переход в конец файла REQUEST
// есть новые запросы от клиентов?
    while (size_pred >= fR.tellg())
    { Sleep(100); fR.seekg(0, ios::end);}
// получен новый запрос
    fR.seekg(size_pred, ios::beg);
//переход к началу полученного запроса
    fR.read((char*)&B, sizeof(B));
//считывание данных клиента
    size_pred = fR.tellg();
// на конец обработанных данных
    fR.close();
// определение индекса массы
    double IMT = B.weight / (0.01*B.height)
    /(0.01*B.height);
    if (18.5 <= IMT && IMT < 25) answer = 1;
    if (18.5 > IMT) answer = 0;
    if (IMT >=25)answer = 2;
// передача ответа клиенту
    fA.open(nameA, ios::binary | ios::app);
//открытие файла ANSWER
    fA.write( (char*)&answer, sizeof(answer));
//запись ответа клиенту
    fA.close();
    }
    }
```

Из текста программы видно, что работа сервера представляет собой бесконечный цикл, в рамках каждой итерации которого выполняются три основных действия:

1. Проверка получения нового запроса в файле REQUEST путем сравнения позиций файловой переменной во внутреннем цикле. При положительном исходе происходит считывание данных запроса.

2. Обработка запроса и получение результата (в нашем примере индекса массы тела).

3. Запись ответа сервера в конец файла ANSWER.

Программа-клиент

Для программы-клиента можно предложить следующий код на языке C++:

```
    #include <iostream>
    #include <fstream>
    #include <windows.h>
    using namespace std;
// структура данных запроса клиента
    struct Person
    {
        char name[25];  //имя
        int height;     //рост
        int weight;     //вес
    } A;

    void main()
    {
    setlocale(LC_ALL, "rus");
    char* nameR = "C:\REQUEST.bin";
//файл для запросов клиентов
    char* nameA = "C:\ANSWER.bin";
//файл для ответов сервера
        ofstream f_REQ;
        ifstream f_ANS;
        long pred_size;
    int answer;

    while (true)
    {
// передача данных от клиента серверу
cout << "Введите запрос: Фамилия Рост Вес"<<endl;
        cin >> A.name>>A.height>>A.weight;
```

```
        cout<< A.name << A.height<<A.weight;
    f_REQ.open( nameR, ios::app |ios::binary);
//открытие файла REQUEST
    f_REQ.write((char*)&A, sizeof(A));
//запись запроса в файл REQUEST
    f_REQ.close();

// поступил ответ от сервера?
        f_ANS.open(nameA, ios::binary);
//открытие файла ANSWER
        f_ANS.seekg(0, ios::end);
//переход в конец файла ANSWER
        pred_size = f_ANS.tellg();
    while (pred_size >= f_ANS.tellg())
    {   Sleep(100);
// ждем и переходим в конец файла ANSWER
        f_ANS.seekg(0, ios::end);   }
// получение ответа от сервера
    f_ANS.seekg(pred_size, ios::beg);
// на начало нового ответа
    f_ANS.read  ((char*)&answer,  sizeof(answer))  ;
//считывание ответа
    f_ANS.close();

// расшифровка ответа
        switch (answer) {
    case 0: {cout << "Недостаток веса\n";break;}
    case 1: {cout << "Норма веса\n";break;}
    case 2: {cout << "Избыток веса\n";break;}
                    }
        }
    }
```

Из приведенного примера видно, что код программы-клиента в некотором смысле «*зеркально*» симметричен коду программы-

сервера. Так клиент в бесконечном цикле выполняет следующие действия:

1. Запись запроса в конец файла REQUEST.
2. Ожидание ответа от сервера. Для проверки также используются значения файлового указателя. При положительном исходе чтение ответа из файла ANSWER.
3. Вывод на консоль полученных от сервера данных.

Представленное в качестве примера клиент-серверное приложение работает корректно и в случае, когда конечное количество клиентов обращается с запросами к серверу. Ограничение на количество клиентов задает только операционная система компьютера, на котором запущено на выполнение данное приложение.

2.3. Модель взаимодействий открытых систем

Основные понятия

Необходимость унификации построения разнородных систем и сетей и взаимодействия их друг с другом привела к переносу на них концепции открытой системы.

Открытой является система (компьютер, вычислительная сеть, ОС, программный пакет, другие аппаратные и программные продукты), которая построена в соответствии с открытыми спецификациями.

Под термином ≪***спецификация***≫ понимают формализованное описание аппаратных или программных компонентов, способов их функционирования, взаимодействия с другими компонентами, условий эксплуатации, особых характеристик. Под открытыми спецификациями понимаются опубликованные, общедоступные спецификации, соответствующие стандартам.

Модель OSI касается только одного аспекта открытости, а именно открытости средств взаимодействия устройств, связанных в компьютерную сеть. Здесь под открытой системой понимается сетевое устройство, готовое взаимодействовать с другими сетевыми устройствами по стандартным правилам, определяющим формат, содержание и значение передаваемых сообщений.

Модель OSI описывает только системные средства взаимодействия, реализуемые операционной системой, системными утилитами, системными аппаратными средствами. Модель не включает средства взаимодействия приложений конечных пользователей. Спецификации ISO/OSI используются производителями аппаратного и программного обеспечений.

В модели OSI средства взаимодействия делятся на ***семь уровней: прикладной, представления, сеансовый, транспортный, сетевой, канальный и физический***. Каждый уровень связан с совершенно определенным аспектом взаимодействия сетевых устройств. Задача каждого уровня – предоставление услуг вышестоящему уровню таким образом, чтобы детали реализации этих услуг были скрыты. **Протоколы** определяют правила взаимодействия модулей одного уровня в разных узлах, а **интерфейсы** – правила взаимодействия модулей соседних уровней в одном узле. Иерархически организованный набор протоколов, достаточный для организации взаимодействия узлов в сети, называется ***стеком протоколов*.**

В соответствии с идеальной схемой модели OSI приложение может обращаться с запросами только к самому верхнему уровню – ***прикладному***, однако на практике многие стеки коммуникационных протоколов предоставляют возможность программистам напрямую обращаться к сервисам, или службам нижележащих уровней.

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.

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), так как 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 определяет 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 задает 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. 8 изображена схема взаимодействия двух систем с точки зрения модели OSI. Движение данных между системами

проходит 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 рис. 8 это взаимодействие обозначено линиями со стрелками между уровнями двух систем).

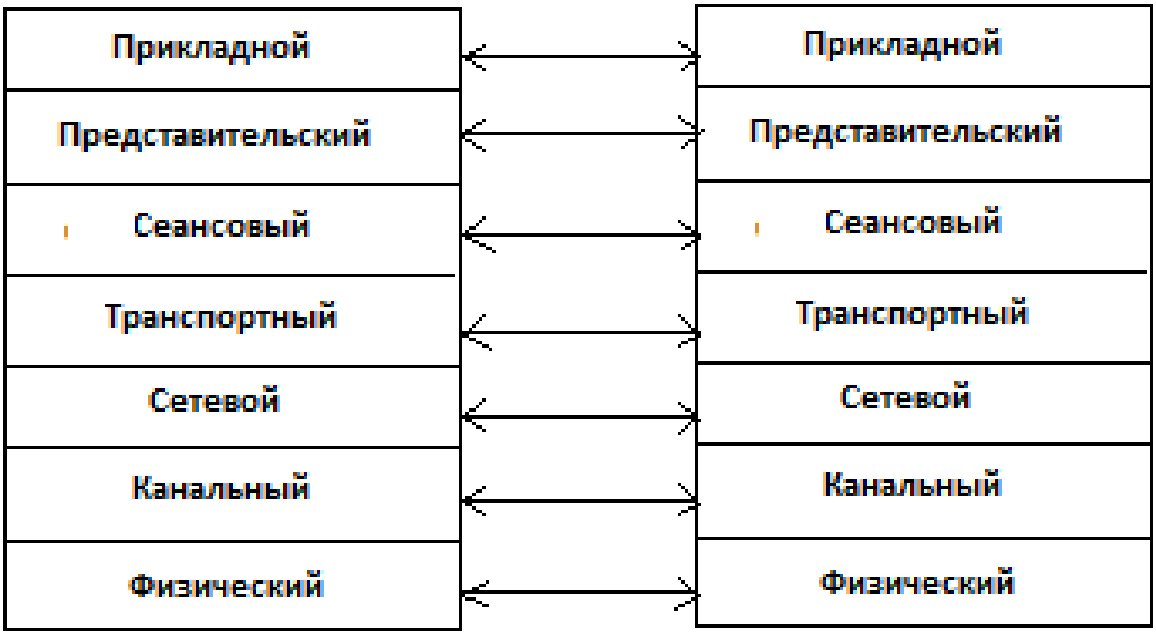


Рис. 8. Схема взаимодействия открытых систем

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

На рис. 9 изображены два процесса (с именами A и B), функционирующие на разных компьютерах в среде соответствующих операционных систем.

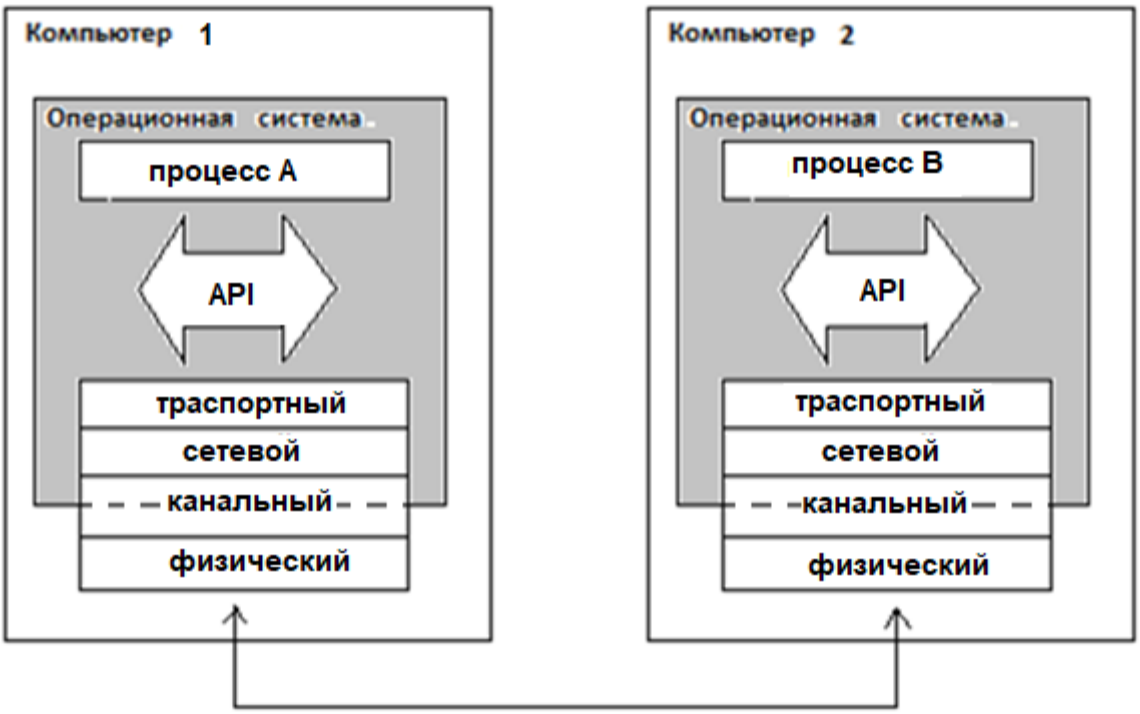


Рис. 9. Схема взаимодействия процессов в сетевом приложении

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 рис. 10 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. 9 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.

3. СТЕК ПРОТОКОЛОВ TCP/IP

Семейство протоколов **TCP/IP** стало промышленным стандартом де-факто для обмена данными между процессами распределенного приложения и поддерживается всеми без исключения операционными системами общего назначения.

Основу обширной коллекции сетевых протоколов и служб стека TCP/IP составляют два важнейших протокола, давших ей имя: **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 этой главе рассматриваются основные сведения о компонентах стека протоколов TCP/IP, необходимые для разработки распределенного сетевого приложения.

Структура TCP/IP

Архитектура TCP/IP была разработана задолго до модели ISO/OSI, в силу чего конструкция TCP/IP несколько отличается от эталонной модели. На рис. 10, где указаны уровни обеих моделей, видно, что они похожи, но не идентичны.

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 OSI берет на себя транспортный уровень TCP/IP. Содержимое сетевого уровня модели 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 OSI, а уровень доступа к сети – совокупности физического и канального уровней.

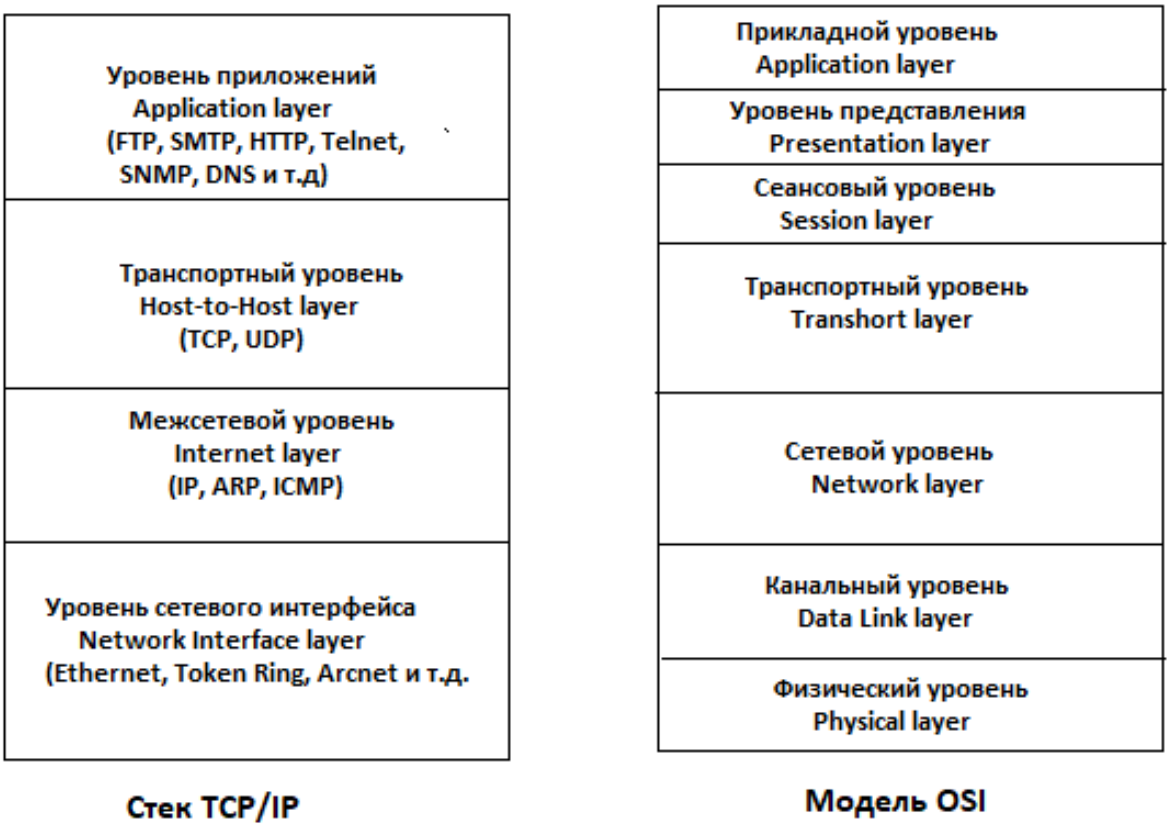


Рис. 10. Сопоставление моделей TCP/IP и OSI

3.1. Протоколы межсетевого уровня

IP-протокол (`Internet Protocol`). В семействе протоколов `TCP/IP` этому протоколу отводится центральная роль. Его основной задачей является доставка *дейтаграмм* (*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.

Пара, состоящая из номера сети и номера узла, является *сетевым адресом*, или в терминологии TCP/IP – IP-*адресом*. IP-адрес (*Internet Protocol Address*) – это уникальный сетевой адрес узла в компьютерной сети, построенной по протоколу IP.

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 хостами (host).

Наиболее распространенной на настоящий момент версией IP протокола является IPv4. Этот протокол оказался самым удачным сетевым протоколом из когда-либо созданных. Поэтому IPv4 быстро превратился в стандарт.

IPv4-адрес представляет собой последовательность из 32 битов. Причем старшие (левые) биты этой последовательности отводятся для адреса сети (Network ID), а младшие (правые) – для адреса хоста в этой сети (Host ID).

При записи IPv4-адреса, как правило, используется числовой формат: 4 десятичных числа, разделенных точкой. Каждое из них является десятичным представлением 8 битов (1 байт) адреса. Например: 128.64.08.16.

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.

Для сети класса **A** старший бит адреса всегда равен 0:

Старший байт адреса этого класса используется для номера сети (Network ID), следующие за ним – номер хоста (Host ID).

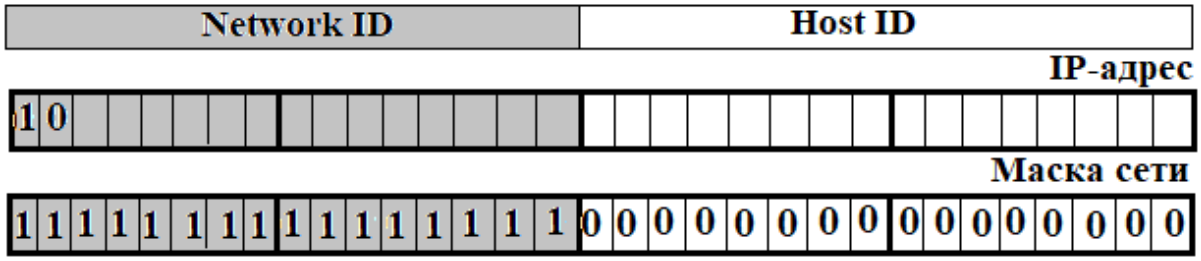
Например:

10.0.0.0 – адрес сети класса A, так как все биты адреса узла Host ID равны 0.

10.0.1.0 – адрес узла этой сети класса A.

10.255.255.255 – широковещательный адрес этой сети, потому что все биты адреса узла Network ID равны 1.

Для сети класса **B** старшие биты адреса всегда равны 10:



Два старших байта адреса этого класса используются для номера сети, следующие за ним – номер хоста.

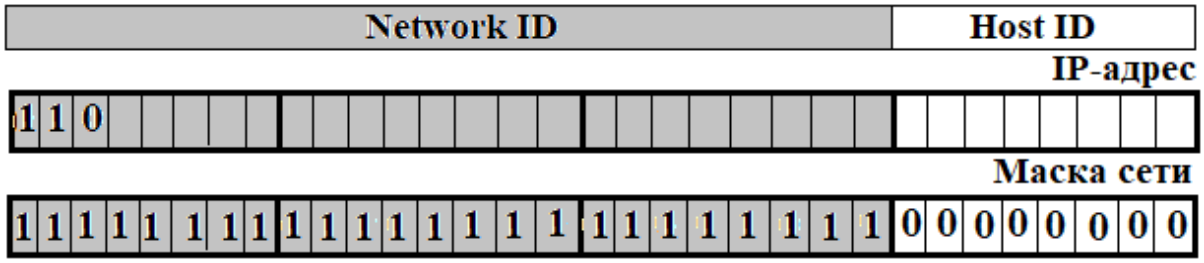
Например:

172.17.0.0 – адрес сети класса B

172.17.0.1 – адрес узла этой сети

172.17.255.255 – широковещательный адрес этой сети.

Для сети класса **C** старшие биты адреса всегда равны 110:



Три старших байта адреса этого класса используются для номера сети, следующие за ним – номер хоста.

Например:

192.168.3.0 – адрес сети класса C

192.168.3.42 – адрес узла этой сети

192.168.3.255 – широковещательный адрес этой сети.

Класс **D**. IP-адреса этого класса используются для групповых адресов (multicast-adress). Четыре старших бита всегда равны 1110. Оставшиеся биты адреса используются для назначения группового адреса.

Класс E. Пять старших бита всегда равны 11110. Адреса этого класса зарезервированы для будущего использования.

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 каждой из сетей сопоставляется 32-битовая маска, которую часто называют маской сети (net mask) или маской подсети (subnet mask), которая по длине равна IP-адресу. Старшие биты маски подсети, состоящие из 1, определяют, какие разряды IP-адреса относятся к идентификатору сети. Младшие биты маски, состоящие из 0, определяют, какие разряды IP-адреса относятся к идентификатору узла. 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.

IPv6. Протокол IPv4 стал жертвой собственной популярности, так как предлагаемое полезное пространство адресов практически исчерпано. Решением возникшей проблемы явилась разработка нового сетевого протокола IPv6, в котором дополнительно были реализованы и другие возможности.

Главным отличительным признаком новой усовершенствованной версии IP-протокола IPv6 является 128-битный адрес (16 байт), позволяющий увеличить адресное пространство более чем на 20 порядков.

В числовом формате адреса IPv6 используется шестнадцатеричное представление, каждые два байта отделяются двоеточием, например так:

2001:0db8:c9d2:aee5:73e3:934a:a5ae:9551

Так как довольно часто встречаются IP–адреса с многими нулями, принято пропускать 4 нуля между двоеточиями и писать просто «::». Также можно отбрасывать ведущие нули для каждой пары байт.

Для примера укажем пары эквивалентных адресов:

2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00:::::

0000:0000:0000:0000:0000:0000:0000:0001
::1

Последняя указанная пара задает loopback–адрес. Он всегда присутствует на каждой машине. Напомним, что в IPv4 такой адрес – 127.0.0.1.

На первоначальном этапе внедрения IPv6 предполагается совместное использование обеих версий IP–протокола, так называемых IPv4-совместимых, или IPv4–преобразованных адресов. Например, адрес IPv4 «192.0.2.33» в формате IPv6 имеет вид «::ffff:192.0.2.33».

При использовании модели *бесклассовой междоменной маршрутизации* для указания маски подсети в IPv6 обычно принято использовать сокращенный формат: в конце записи IP-адреса после знака ‘/’указывается количество старших бит маски для номера сети). Например:

2001:db8::/32 ;
2001:db8:5413:4028::9db9/64 .

Другой интересной особенностью IPv6 является возможность автоконфигурации. Автоконфигурация – 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 сложностью администрирования современных сетей с большим количеством хостов и необходимостью поддерживать мобильных (перемещающихся) пользователей. Большое внимание в новой версии протокола уделяется вопросам безопасности.

IP-маршрутизация. Протокол IP обеспечивает доставку дейтаграмм в пределах всей составной IP-сети. Составной IP-сетью называется 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. 11 изображен пример составной IP-сети.

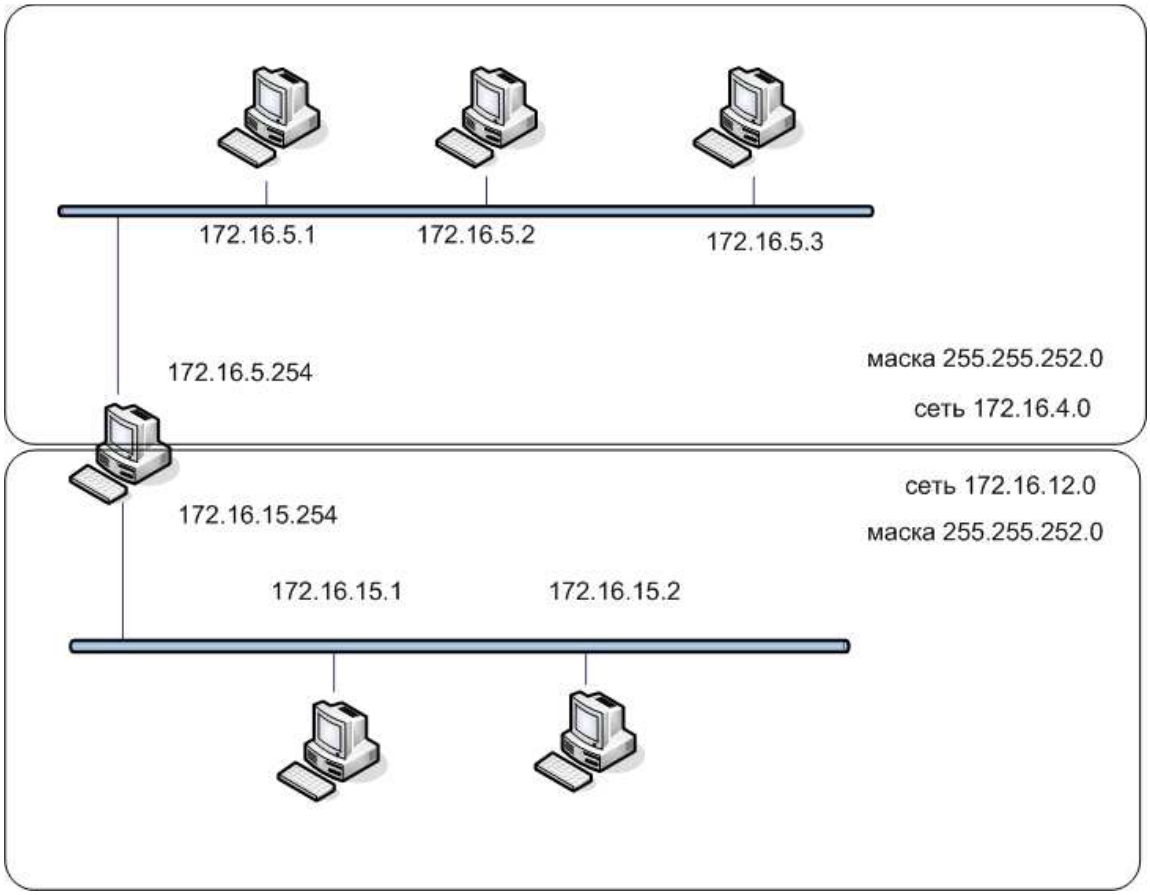


Рис. 11. Двухсегментная составная сеть

Шлюз имеет два интерфейса: один принадлежит сети 172.16.5.0, другой – сети 172.16.15.0. 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.

Протокол ICMP (*Internet Control Message Protocol*) 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. С помощью протокола ICMP осуществляется деятельность утилит достижимости (ping, traceroute), регулируется частота отправки IP-дейтаграмм, оптимизируется MTU для маршрута передачи IP-дейтаграмм, доставляется хостам, маршрутизаторам и шлюзам всевозможная служебная информация, осуществляются поиск и переадресация маршрутизаторов, оптимизируются маршруты, диагностируются ошибки и оповещаются узлы IP-сети.

Протокол ARP (*Address Resolution Protocol*). IP-адреса воспринимаются только на сетевом уровне и вышестоящих уровнях TCP/IP. На канальном уровне всегда действует другая схема адресации, которая зависит от используемого протокола этого уровня. Для установки соответствия между IP-адресами и теми или иными MAC-адресами (Media Access Control-address), действующими на канальном уровне, применяется механизм привязки адресов по протоколу ARP. 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. 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 применяется для получения IP-адреса по MAC-адресу. В настоящее время он заменен на протокол прикладного уровня DHCP, предлагающий более гибкий метод присвоение адресов.

На межсетевом уровне TCP/IP используются и другие важные протоколы. Можно упомянуть, например, основной дистанционно-векторный протокол маршрутизации **RIP** (*Routing Information Protocol*) или протокол динамической маршрутизации **OSPF** (*Open Shortest Path First*), основное назначение которых – сбор актуальной маршрутной информации, необходимой для корректной и эффективной работы сетевого протокола IP.

3.2. 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.

В компьютерных сетях существуют два принципиально разных способа передачи данных. ***Первый*** из них предполагает посылку пакетов данных от одного узла другому (или сразу нескольким узлам) без получения подтверждения о доставке и даже без гарантии того, что передаваемые пакеты будут получены в правильной последовательности. Примером такого протокола может служить протокол UDP. Основные преимущества дейтаграмных протоколов заключаются в высоком быстродействии и возможности широковещательной передачи данных, когда один узел

отправляет сообщения, а другие их получают, причем все одновременно.

Второй способ передачи данных предполагает создание виртуального соединения между двумя различными узлами сети. При этом соединение создается средствами дейтаграммных протоколов, однако доставка пакетов в этом случае является гарантированной. Пакеты всегда доходят в целостности и сохранности, причем в правильном порядке, хотя быстродействие получается в среднем ниже за счет посылки подтверждений. Примером протокола, использующего соединение, может служить протокол TCP.

Процесс, получающий или отправляющий данные с помощью транспортного уровня, идентифицируется номером, который называется *номером порта*. Таким образом, адресат в сети TCP/IP полностью определяется тройкой: IP-адресом, номером порта и типом протокола транспортного уровня (UDP или TCP). Основным отличием протоколов UDP и TCP является то, что UDP – протокол без установления соединения (ориентирован на сообщения), а TCP – протокол на основе соединения (ориентирован на поток). На рис. 12 изображен стек протоколов TCP в двух разрезах и названия, принятые для обозначения блоков данных в протоколах TCP и UDP.

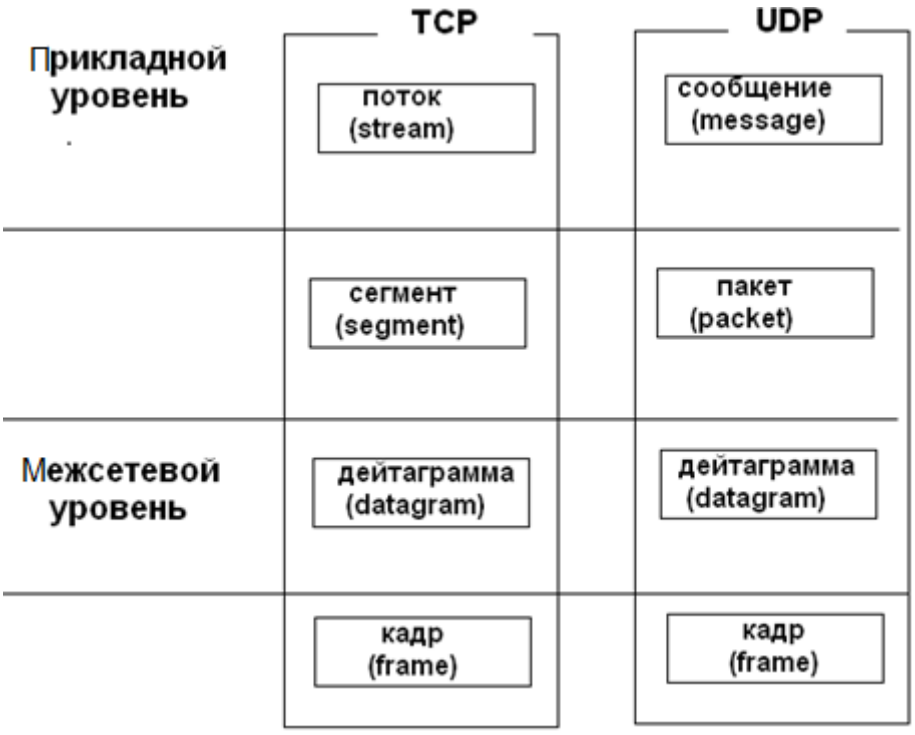


Рис. 12. Протоколы TCP и UDP в стеке TCP/IP

Движение информации с верхних уровней на нижние сопровождается *инкапсуляцией* данных, 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.

Номера портов, используемые для идентификации прикладных процессов, 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. 1.

Таблица 1

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

Хорошо известные номера портов присваиваются базовым системным службам (core services), имеющим системные привилегии. Зарегистрированные номера портов присваиваются промышленным приложениям и процессам. Распределение некоторых хорошо известных и зарегистрированных номеров портов приведено в табл. 2. Динамическое выделение прикладным процессам номеров портов, как правило, производится специализированной службой операционной системы. Некоторые системы TCP/IP применяют диапазон значений от 1024 до 5000 для назначения динамических номеров портов.

Таблица 2

Номер порта	Протокол	Описание
20	TCP	FTP-DATA для передачи данных FTP
21	TCP	FTP для передачи команд FTP
22	TCP, UDP	SSH (*Secure Shell*) – криптографический сетевой протокол для безопасной передачи данных
23	TCP, UDP	Telnet – применяется для передачи текстовых сообщений в незашифрованном виде

25	TCP, UDP	SMTP (*Simple Mail Transfer Protocol*) – применяется для пересылки почтовых сообщений в виде незашифрованного текста
53	TCP, UDP	DNS(*Domain Name Server*)
67	TCP, UDP	*Dynamic Host Configuration Protocol* (DHCP) Server
68	TCP, UDP	*Dynamic Host Configuration Protocol* (DHCP) Client
80	TCP, UDP	WWW(*World Wide Web*)
88	TCP,UDP	Система аутентификации Kerberos
110	TCP, UDP	POP3(*Post Office Protocol, Version 3*)
161	TCP, UDP	SNMP (*Simple Network Management Protocol*)
220	TCP, UDP	IMAP3 (*Interactive Mail Access Protocol, version 3*)
443	TCP, UDP	HTTPS (*HyperText Transfer Protocol Secure*) – HTTP с шифрованием по SSL или TLS

UDP – «протокол пользовательских дейтаграмм», наиболее простой транспорт в стеке TCP/IP. Он обеспечивает передачу данных без установления соединения. 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 суммы 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.

Фактически UDP – это тонкая прослойка интерфейса, обеспечивающая доступ процессов прикладного уровня непосредственно к протоколу IP. Он представляет собой простой и удобный способ передачи небольших объемов информации, когда требования к надежности невелики. Полезным свойством UDP является возможность посылки дейтограммы на широковещательный или групповые адреса.

TCP – более сложный и обладающий более богатыми возможностями «протокол управления передачей», обеспечивает надежную связь между абонентами посредством виртуального канала, используя метод передачи с установлением соединения. В связи с этим протокол TCP имеет несколько характерных особенностей:

1. Для каждой пары взаимодействующих абонентов создается отдельное соединение. Порядок функционирования включает: запрос соединения, подтверждение соединения, обмен данными через него, затем разрыв соединения. На всех стадиях существенно участие обеих взаимодействующих сторон.

2. Так как поток данных предполагается непрерывным, а порция, передаваемая одним обращением к протоколу, конечна, поток представляется состоящим из сегментов – блоков данных соответствующего формата, вкладываемых в IP-пакеты.

3. Надежность передачи обеспечивается наличием подтверждений приема данных, которые отсылаются (тоже как поток) в обратном направлении – от приемника к источнику. По этой причине соединение на самом деле является двунаправленным, даже если прикладной уровень использует его как однонаправленное. В случае дуплексного канала подтверждения могут совмещаться с встречными данными.

4. Потоки рассматриваются как непрерывные, а составляющие их сегменты могут подвергаться перекомпоновке, поэтому как для идентификации сегментов данных, так и для подтверждений их приема используется не номер сегмента, а позиция в потоке.

Протокол TCP обеспечивает одновременно нескольких соединений. Каждый процесс прикладного уровня идентифицируется номером порта. Заголовок TCP-сегмента содержит номера портов отправителя и получателя.

На рис. 13 ломаными прямыми линиями изображены каналы между процессами прикладного уровня A, B, C, D и E.

Процесс D работает на хосте с IP-адресом 172.18.5.2 и использует порты 2777 и 2888 для связи с двумя процессами A (порт 2000) и B (порт 2500), функционирующими на хосте с IP-адресом 172.18.15.1. Процессы C и E образуют канал между хостами 172.18.5.1 и 172.15.5.3 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.

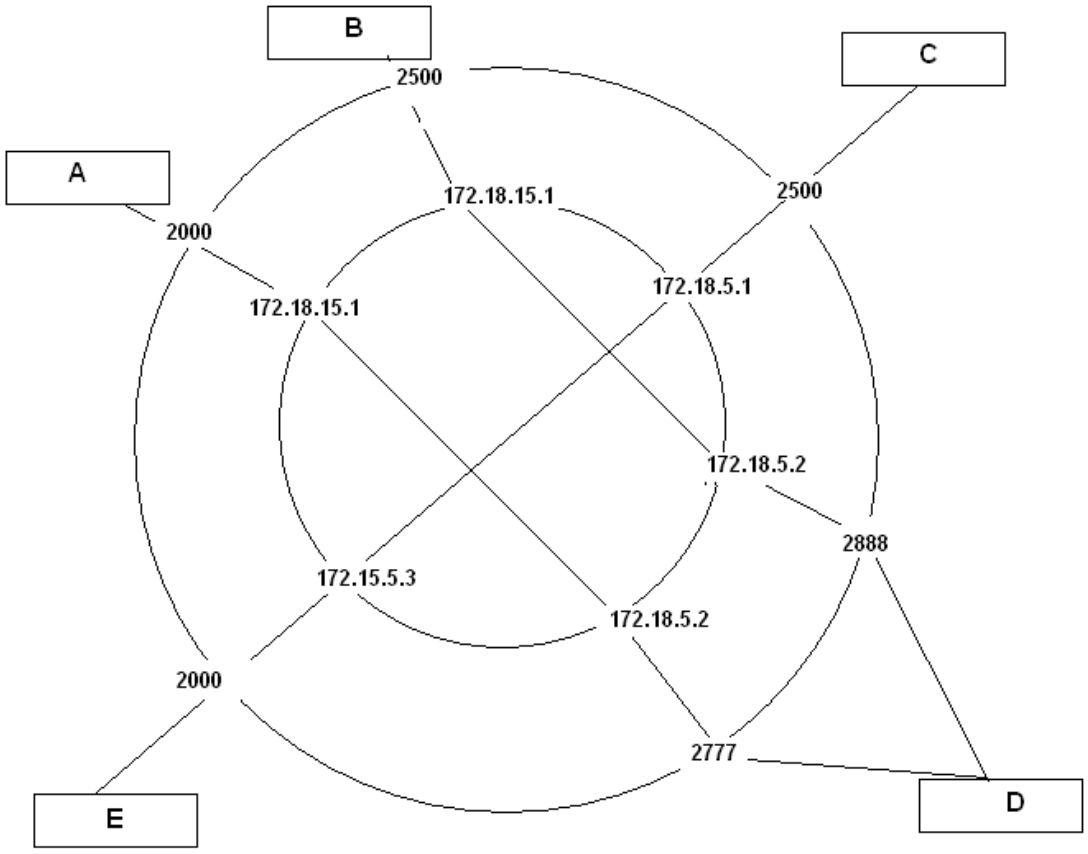


Рис. 13. Разделение каналов в сети TCP/IP

3.3 Интерфейс внутренней петли

Большинство реализаций TCP/IP поддерживает интерфейс внутренней петли (loopback interface), который позволяет двум прикладным процессам, находящимся на одном хосте, обмениваться данными посредством протокола TCP/IP.

При этом, как обычно, формируются дейтаграммы, но они не покидают пределы одного хоста. Для интерфейса внутренней петли, как уже упоминалось, 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*. В системном файле hosts можно обнаружить следующую запись для loopback:

```
127.0.0.1       localhost
```

На рис. 14 приведена упрощенная схема обработки данных интерфейсом внутренней петли.

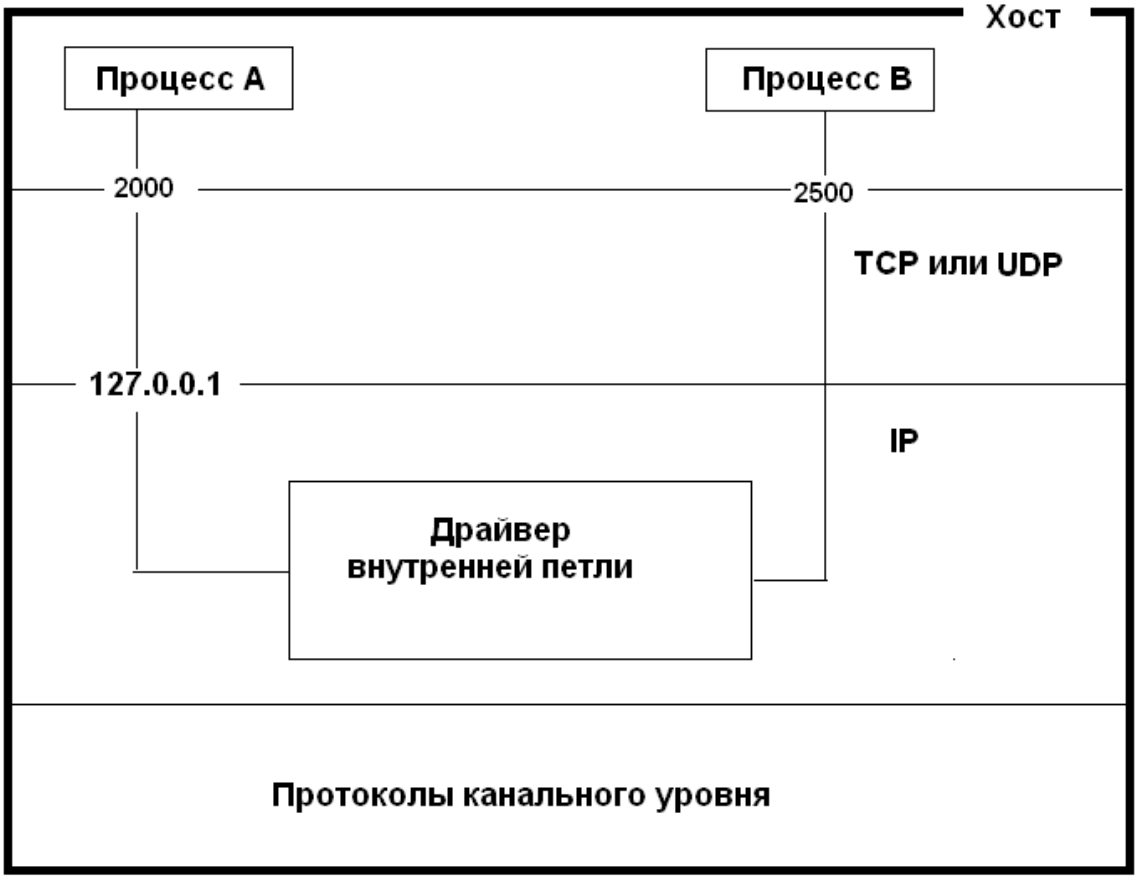


Рис. 14. Схема работы интерфейса внутренней петли

Прикладной процесс A, используя номер порта 2000, отправляет данные процессу B. Указав в параметрах сокета процесса B сетевой адрес 127.0.0.1, процесс A обеспечил обработку посылаемых дейтаграмм на межсетевом уровне драйвером внутренней петли, который направляет эти дейтаграммы во входную очередь модуля IP. IP-модуль, следуя обыкновенной логике своей работы, доставляет данные на транспортный уровень. Далее протокол транспортного уровня в соответствии с номером порта 2500 в заголовке сегмента (или пакета) направляет данные процессу B.

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

3.4. Системные службы

Программную реализацию протоколов прикладного уровня TCP/IP принято называть *системными службами*. С точки зрения пользователей, компьютерные сети представляют собой набор служб (сервисов), таких как электронная почта, WWW, интернет-телефония и интернет-телевидение.

Транспортные функции сети, обеспечивающие работу этих служб, скрыты от пользователей, хотя иногда и влияют на некоторые детали предоставления службы, например, недостаточно высокая надежность доступа в Интернет по телефонным каналам потребовала коротких TCP-сеансов в службе WWW при передаче содержания веб-страниц.

Помимо служб, ориентированных на конечных пользователей, существуют службы, ориентированные на сетевых администраторов, решающие задачи конфигурирования и управления сетевыми устройствами. В эту категорию входят службы FTP, Telnet и SNMP. Дополняют общую картину службы, помогающие компьютерам и сетевым устройствам организовать свою работу, такие как службы DNS и DHCP.

Служба DNS (*Domain Name System*) – одна из важнейших служб TCP/IP, появление которой в 1980-х гг. дало мощный

толчок развитию TCP/IP и всемирной сети Internet. DNS обеспечивает преобразование 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.

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 имена доменов).

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 подразделяются на ***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.

Служба 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.

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 динамическое. 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.

Служба TELNET создавалась для подключения пользователей к удаленному компьютеру с целью производить на нем вычисления, работать с базами данных, подготавливать документы и т. д. Служба также может быть использована для управления работой удаленного компьютера, настройки и диагностирования сетевого оборудования.

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

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 также носит служба, реализующая этот протокол. Архитектура службы FTP представлена на рис. 15.

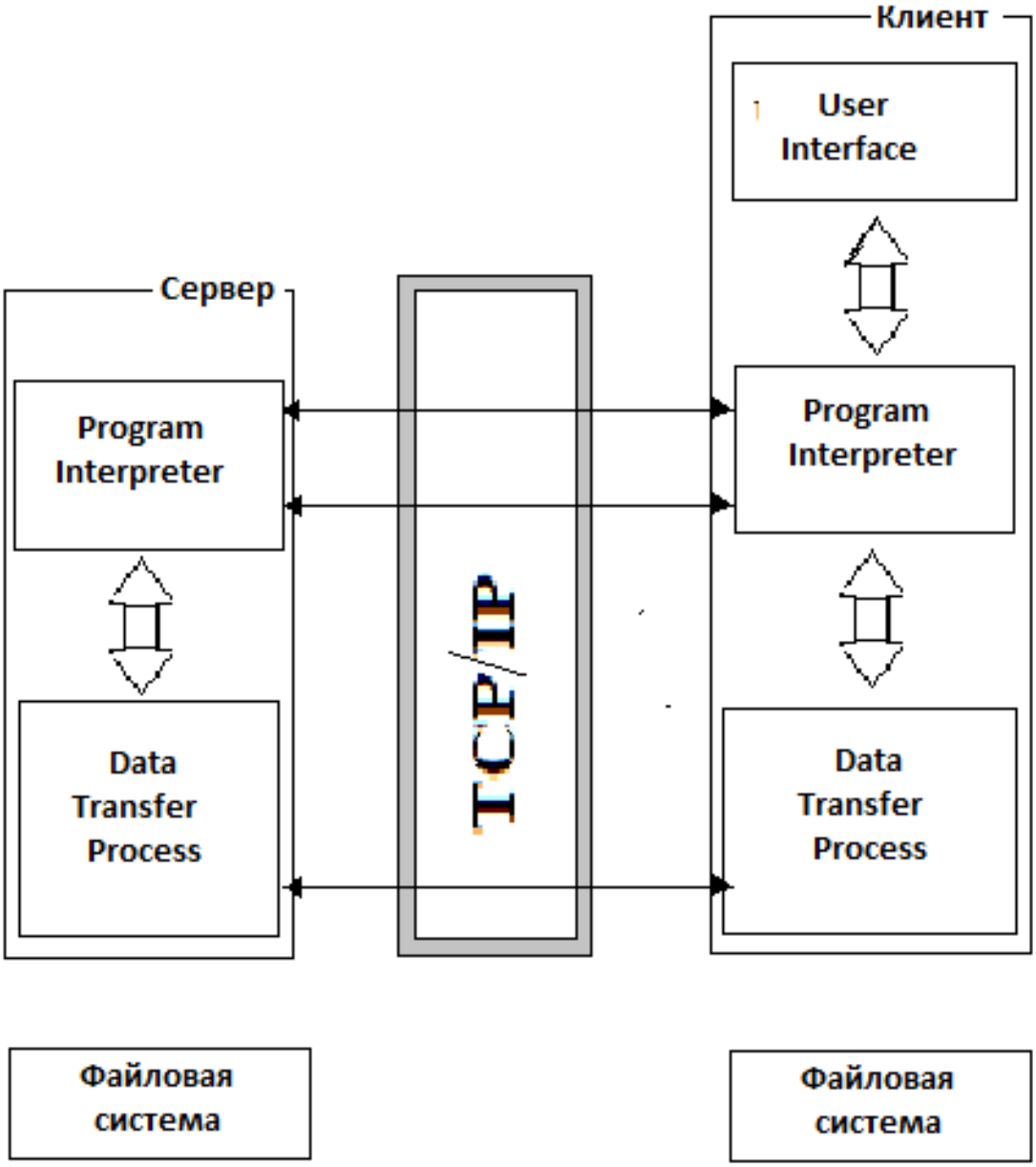


Рис. 15. Взаимодействие основных компонентов службы FTP

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 клиент, 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.

Электронная почта. Современная электронная почта основывается на протоколах прикладного уровня **SMTP** (*Simple Mail Transport Protocol*), **POP3** (*Post Office Protocol*) и **IMAP4** (*Internet Message Access Protocol*).

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. 16 изображена схема взаимодействия этих 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.

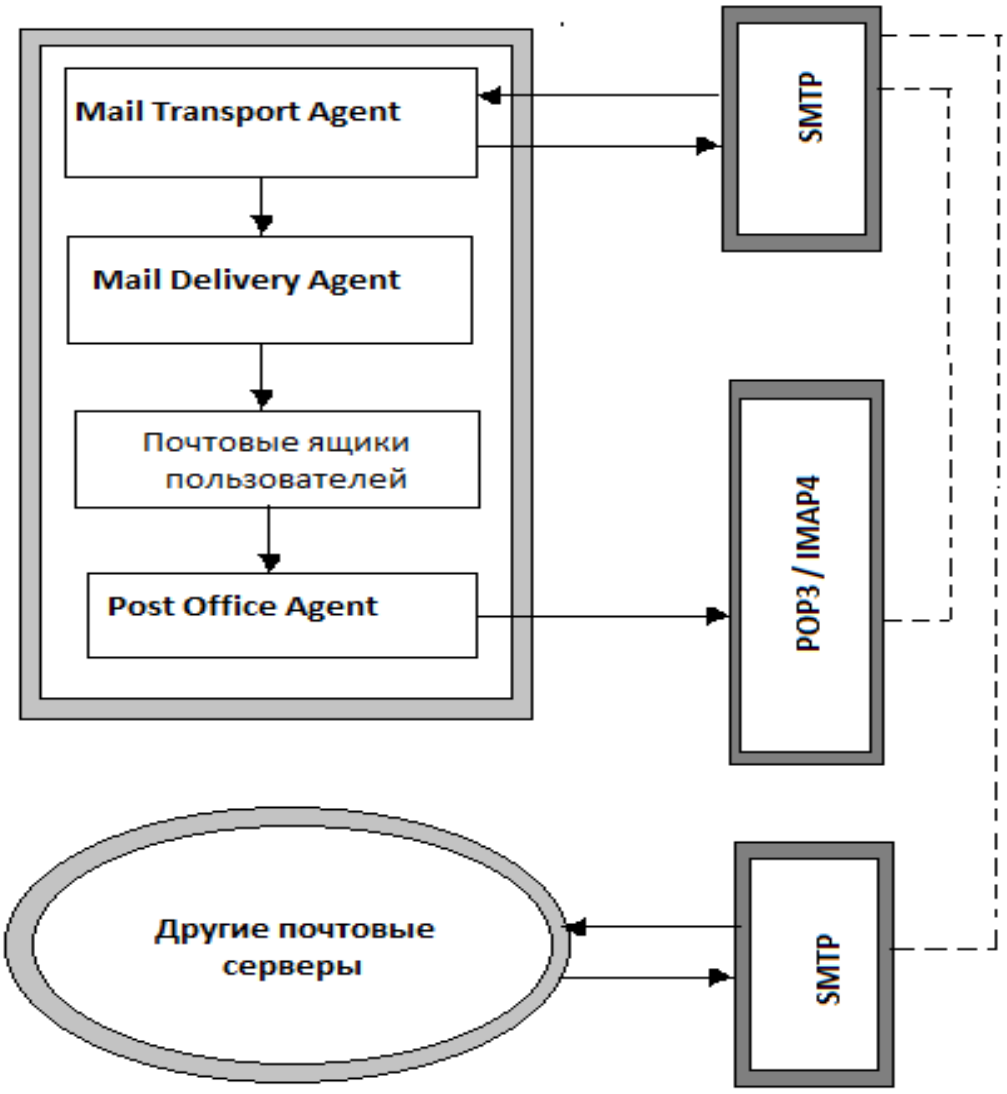


Рис. 16. Схема взаимодействия компонентов электронной почты

Под почтовым сервером обычно подразумевают совокупность серверов MTA, POA, программу MDA, систему хранения почтовых сообщений (почтовые ящики) и ряд дополнительных программ, обеспечивающих безопасность и дополнительный сервис, расположенных на отдельном компьютере с TCP/IP-интерфейсом.

Почтовый клиент представляет собой программу, устанавливаемую на пользовательском компьютере и взаимодействующую с почтовым сервером с помощью TPC/IP-соединения.

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. В Internet протокол HTTP применяется с 1990 г.

HTTP часто используется как протокол передачи информации для других протоколов прикладного уровня. В таком случае говорят, что протокол HTTP используется как «транспорт». API многих программных продуктов также подразумевает использование HTTP для передачи данных – сами данные при этом могут иметь любой формат, например, XML или JSON.

HTTP предназначен для построения систем архитектуры «клиент-сервер». Как правило, передача данных по протоколу HTTP осуществляется через TCP/IP-соединения. Серверное программное обеспечение по умолчанию использует TCP-порт 80, хотя может использовать и любой другой.

На данный момент именно благодаря протоколу HTTP обеспечивается работа Всемирной паутины WWW (*World Wide Web*).

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

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». Серверная и клиентская части службы (обычно называемые веб-сервером и веб-браузером) взаимодействуют друг с другом с помощью протокола HTTP.

3.5. Сетевые утилиты

Утилиты представляют собой внешние команды операционной системы и предназначаются для диагностики сети. В табл. 3 перечислены популярные сетевые утилиты, поддерживаемые операционной системой Windows.

Таблица 3

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
netstat	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
net	Управление сетью

Утилита nslookup, проверяющая правильность функционирования DNS-серверов, 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). На рис. 17 приведен скрин, демонстрирующий результаты работы этой утилиты. Запуск утилиты в интерактивном режиме осуществляется вызовом команды nslookup без параметров.

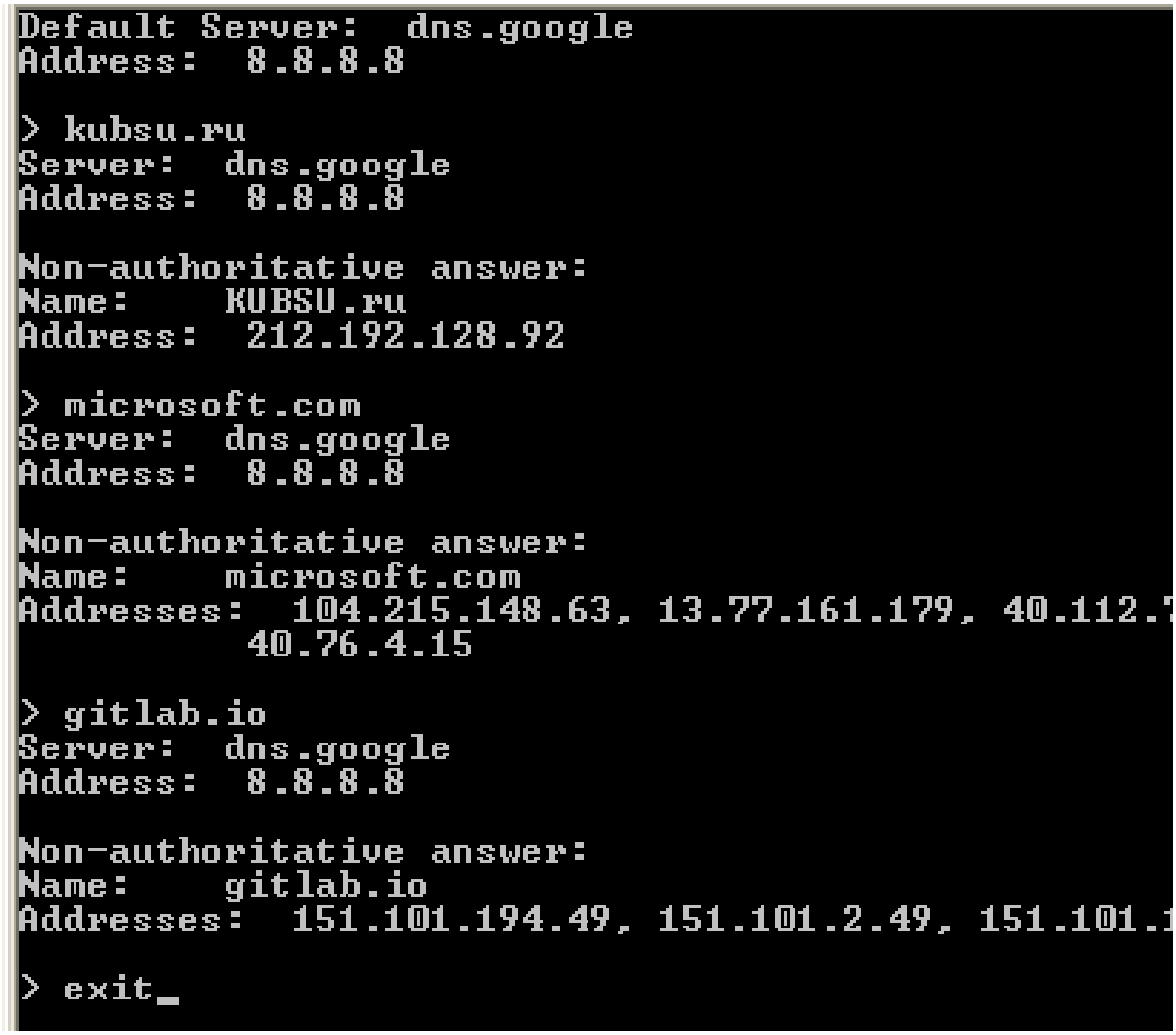


Рис. 17. Результат выполнения команды nslookup

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 kubsu.ru, microsoft.com, gitlab.io и для завершения работы утилиты была введена команда exit.

Наиболее востребованной сетевой утилитой является утилита ipconfig. 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. Короткий отчет о конфигурации TCP/IP можно получить, выдав команду ipconfig без параметров. Для получения полного отчета нужно использовать ключ /all.

4. ИНТЕРФЕЙС СОКЕТОВ

Сокеты Беркли. В процессе сетевого соединения два компьютера или процесса обмениваются данными. Сетевые профессионалы называют каждую сторону такого соединения конечной точкой (endpoint). Сокет является выражением абстракции конечной точки сетевого соединения. Концепция сокетов дает программистам понятную и удобную модель для разработки сетевых приложений. В 1980-х гг. в Калифорнийском университете в г. Беркли группа исследователей-программистов на основе парадигмы (модели) сокетов разработала интерфейс прикладного программирования для сетевых приложений TCP/IP, который часто называется *сокетами Беркли* (Berkeley sockets). Интерфейс Беркли – всего лишь одна (хотя и чаще всего используемая) реализация интерфейса прикладного программирования, основанная на модели сокетов. Интерфейс Беркли дает инструмент для создания как весьма сложных серверных приложений, так и устойчиво работающих приложений-клиентов. Сокеты позволяют работать со множеством протоколов и являются удобным средством межпроцессорного взаимодействия. Однако в данной работе ограничимся рассмотрением сокетов семейства протоколов TCP/IP, использующихся для обмена данными между узлами сети Интернет.

Интерфейс Winsock

Сеть Интернет исторически основывается на протоколах TCP/IP. Спецификация Windows Socket описывает стандарт, по которому программы Windows обязаны общаться с сетями на базе TCP/IP. Цель разработки – создание единого интерфейса прикладного программирования (API), который бы использовался в равной мере как разработчиками, так и продавцами сетевого программного обеспечения и служил бы стандартом работы с TCP/IP для Windows.

Сокеты Windows (часто называемые «Winsock») – интерфейс прикладного программирования, разработанный на основе модели сокетов. Тогда как сокеты Беркли используются на разных операционных системах, Winsock предназначен исключительно для

семейства Windows. Модель программирования Winsock тесно связана с моделью сокетов Беркли, поскольку разработка Winsock базировалась на модели интерфейса Беркли.

Интерфейс сокетов Windows не входит в состав системы Windows, а реализован в виде динамически загружаемой библиотеки (DLL) (рис.18). Модуль WINSOCK.DLL находится между стеком протоколов TCP/IP и клиентскими приложениями. Другими словами, он управляет интерфейсом к стеку TCP/IP. В качестве протоколов высокого уровня на рисунке указаны RPC (*удаленный вызов процедур*) протоколы, предназначенные для организации распределённых вычислений и создания распределенных клиент-серверных информационных систем.

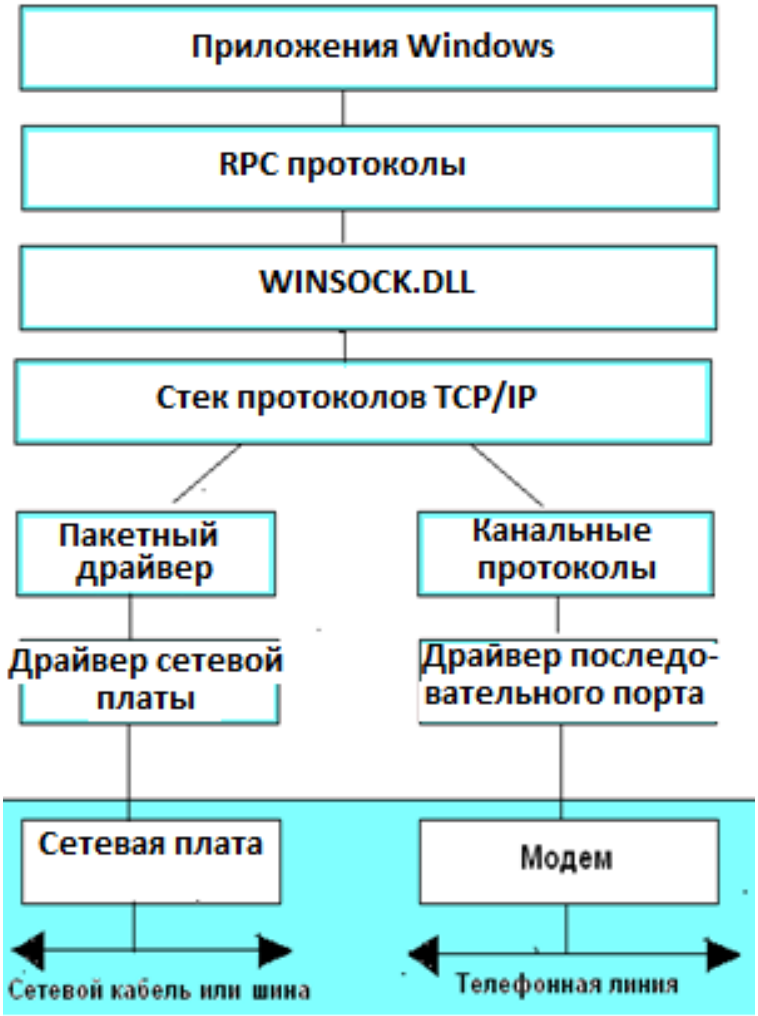


Рис.18. Место Winsock в операционной среде Windows

Интерфейс прикладного программирования Winsock API представляет собой набор функций, используемых программистами для написания приложений в определенной сетевой операционной среде. В состав Winsock входит множество функций из интерфейса Беркли, изначально разработанных для Unix – операционной системы, для которой и предназначались сокеты Беркли. Кроме того, имеется набор специфических для Windows функций, позволяющих программистам пользоваться преимуществами интерфейса Windows, основанного на передаче сообщений.

4.1. Библиотека функций Winsock

Интерфейс прикладного программирования Winsock содержит набор (библиотеку) функций общего пользования, требуемых для решения определенного класса задач. Спецификация Winsock разделяет всю библиотеку на три группы:

– функции сокетов в стиле Беркли, включенные в состав Winsock API;

– функции для получения информации об именах доменов, коммуникационных службах и протоколах.

– специфические функции Windows, расширяющие набор функций интерфейса сокетов Беркли.

Библиотека Winsock поддерживает два вида сокетов – ***синхронные*** (*блокируемые*) и ***асинхронные*** (*неблокируемые*). Синхронные сокеты задерживают управление на время выполнения операции, а асинхронные возвращают его немедленно, продолжая выполнение в фоновом режиме, и, закончив работу, уведомляют об этом вызывающий код. Как правило, все функции сетевого ввода-вывода сокетов в стиле Беркли – блокирующие. На деле блокирование проявляется задержкой в выполнении программы до окончания передачи-приема сетевых данных.

В табл. 4 приведены функции сокетов в стиле Беркли, которые могут блокировать выполнение операций в Winsock API. По приведенным в таблице описаниям функций сокетов можно заметить, что все эти функции либо производят операции ввода-вывода, либо ждут окончания сетевого ввода-вывода до того, как

завершить выполнение. Отсюда можно сделать вывод, что любая функция, так или иначе связанная с операциями ввода-вывода, может блокировать выполнение остальных функций `Winsock API`.

Таблица 4

Функция	Описание
`accept`	Подтверждает запрос на установление соединения, образует новый сокет и соединяет его с удаленным сетевым компьютером, запрашивающим соединение. Исходный сокет возвращается в состояние приема входящих запросов.
`closesocket`	Закрывает одну сторону в соединении сокетов.
`connect`	Устанавливает соединение на указанном сокете.
`recv`	Принимает данные из соединенного сокета.
`recvfrom`	Принимает данные из соединенного или не соединенного сокета.
`select`	Выполняет синхронные мультиплексные операции ввода-вывода путем наблюдения за состоянием нескольких сокетов.
`send`	Передает данные через соединенный сокет.
`sendto`	Передает данные через не соединенный или соединенный сокет.

С другой стороны, функции, приведенные в табл. 5, не выполняют операций ввода-вывода в процессе работы. Они либо преобразуют информацию, либо имеют дело с локальным сокетом сетевого компьютера. Другими словами, их деятельность не связана с удаленными сетевыми устройствами. Поэтому, несмотря на то что они также являются функциями в стиле Беркли, ни одна из них не блокирует операции прикладной программы.

Таблица 5

Функция	Описание
`bind`	Присваивает имя неинициализированному (новому) сокету.
`getpeername`	Получает имя удаленного процесса, связанного с указанным сокетом (в дальнейшем вы узнаете, что Winsock API хранит эту информацию в локальной структуре данных,

	поэтому вызов данной функции не связан с операциями сетевого ввода-вывода).
getsockname	Возвращает имя указанного местного (локального) сокета.
getsockopt	Возвращает статус (опции) указанного сокета.
htonl	Преобразует порядок байтов в 32-разрядном числе из машинно-зависимого в сетевой.
htons	Преобразует порядок байтов в 16-разрядном числе из машинно-зависимого в сетевой.
inet_addr	Преобразует строку с IP-адресом в формате десятичное с точкой в 32-разрядное двоичное число (с сетевым порядком байтов).
inet_ntoa	Преобразует IP-адрес в формат десятичное с точкой.
ioctlsocket	Управляет параметрами сокета, относящимися к обработке операций сетевого ввода-вывода.
listen	Переводит указанный сокет в состояние прослушивания запросов на входное соединение (функция переводит сокет в режим прослушивания, однако сама по себе не производит никаких операций сетевого ввода-вывода).
ntohl	Преобразует порядок байтов 32-разрядного числа из сетевого в машинно-зависимый (порядок хоста).
ntohs	Преобразует порядок байтов 16-разрядного числа из сетевого в машинно-зависимый (порядок хоста).
setsockopt	Устанавливает режим (опции) работы сокета.
shutdown	Закрывает одну сторону дуплексного соединения (только для местного компьютера).
socket	Образует точку сетевого соединения и возвращает дескриптор сокета.

4.2. Модель сокетов

Сетевое соединение – это процесс передачи данных по сети между двумя компьютерами или процессами. Важным понятием в сетевом программировании является **сокет**. Сокет можно рассматривать как конечный пункт передачи данных по сети. Другими словами, когда программы используют сокет, для них он является абстракцией, представляющий одно из окончаний сетевого соединения. Для установления соединения в абстрактной модели

сокетов необходимо, чтобы каждая из сетевых программ имела собственный сокет. С помощью сокетов разработчик может создавать свои, отличные от стандартных, прикладные протоколы, определяющие формат сообщений при передаче данных.

На основе модели **сокетов** создается программный интерфейс для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одном компьютере, так и на различных, связанных между собой сетью. Сетевые компьютеры идентифицируются на сетевом уровне **IP** при помощи сетевого адреса. Это значит, что каждый компьютер, подключенный к сети, должен иметь уникальный сетевой адрес.

Сокеты существуют внутри коммуникационных доменов. **Домены** – это абстракции, которые подразумевают конкретную структуру адресации и множество протоколов, что определяет различные типы сокетов внутри домена. Примерами коммуникационных доменов могут быть: Unix домен, Internet домен и т. д.

Каждый сокет имеет **тип** и ассоциированный с ним **процесс.**

В Internet домене **сокет** – это комбинация **IP адреса** и номера **порта**, которая однозначно определяет отдельный сетевой процесс во всей глобальной сети Internet. Два сокета, один для хоста-получателя, другой для хоста-отправителя, определяют соединение для протоколов, ориентированных на установление связи, таких, как TCP.

Тип сокета определяется режимом сетевого соединения: потоковый, дейтаграммный и сырой сокет. В потоковых (ориентированных на соединение) протоколах перемещение данных представляется как единый, последовательный поток байтов без какого-либо деления на блоки. В дейтаграммных (неориентированных на соединение) протоколах сетевые данные перемещаются в виде отдельных пакетов, называемых датаграммами. Сокеты могут работать как с неориентированными (**UDP**), так и с ориентированными на соединение протоколами (**TCP**). TCP гарантирует доставку пакетов, их очередность, автоматически разбивает данные на пакеты и контролирует их передачу, в отличие от UDP. Но при этом TCP работает медленнее за счет повторной передачи потерянных пакетов и большего количества выполняемых операций над пакетами. Поэтому там, где требуется гарантированная доставка

(WWW, Telnet, e-mail), используется TCP, если же требуется передавать данные в реальном времени (многопользовательские игры, видео, звук), используют UDP. В табл. 6 представлены характеристики основных типов сокетов: потоковых (SOCK_STREAM) и датаграммных (SOCK_DGRAM).

Таблица 6

Потоковый (SOCK_STREAM)	Дейтаграммный (SOCK_DGRAM)
Устанавливает соединение	Нет
Гарантирует доставку данных	Нет
Гарантирует порядок доставки пакетов	Нет
Гарантирует целостность пакетов	Тоже
Разбивает сообщение на пакеты	Нет
Контролирует поток данных	Нет

Интерфейс сокетов позволяет указать и третий тип соединения, соответствующий *«**сырому**»* сокету (*raw socket*). «Сырой» сокет – разновидность сокетов Беркли, позволяющий собирать TCP/IP-пакеты, контролируя каждый бит заголовка и отправляя в сеть нестандартные пакеты. «Сырой» сокет позволяет программе использовать напрямую те низкоуровневые протоколы, которые обычно используются сетевыми протоколами более высокого уровня. Это достигается путем «ручного» формирования TCP\IP-пакетов, равно как и полного доступа к содержимому заголовков полученных TCP\IP-пакетов, что необходимо многим сетевым сканерам, FireWall-ам, брандмаузерам и, разумеется, атакующим программам, например, устанавливающим в поле «адрес отправителя» адрес самого получателя. Например, программа ***Ping*** создает «сырой» сокет, чтобы напрямую использовать протокол управляющих сообщений Интернет (ICMP). Обычно ICMP служит для передачи сообщений о различных сетевых ошибках. Как

правило, прикладные программы не используют ICMP. Они предоставляют самой сети решать проблемы, связанные с ошибками. Транспортные протоколы Интернет самостоятельно доставят сообщение об ошибке, адресованное прикладной программе. Однако прикладная программа может напрямую обратиться к уровню IP или ICMP. Для этого ей придется использовать «сырой» сокет.

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

Большинство сетевых программ основано на **клиент-серверной** технологии. По этой технологии каждое сетевое соединение характеризуется своими двумя конечными точками. Технология клиент-сервер делает эти две точки неравноправными. Одна должна выполнять функции сервера, а другая – клиента. Конечная точка-клиент инициирует запрос к сетевым службам и представлена программой-клиентом или процессом-клиентом. Конечная точка, отвечающая на запрос, представлена программой или процессом-сервером. Этим типам конечных точек соединения в модели соответствуют различные ***клиентские*** и ***серверные*** сокеты. ***Клиентские*** можно сравнить с оконечными аппаратами телефонной сети, а ***серверные*** – с коммутаторами. Клиентское приложение (например, браузер) использует только клиентские сокеты, а серверные (например, веб-сервер) – как клиентские, так и серверные сокеты. Для их реализации интерфейс сокетов предлагает различные наборы функций. Так, например, табл. 7 демонстрирует применение основных функций Winsock клиентскими и серверными потоковыми сокетами.

Таблица 7

Общие	
Socket	Создать новый сокет и вернуть файловый дескриптор
Send	Отправить данные по сети
Recv	Получить данные из сети

Closesocket	Закрыть соединение
Серверные	
Bind	Связать сокет с локальным IP-адресом и портом
Listen	Объявить о желании принимать соединения. Слушает порт и ждет, когда будет установлено соединение
Accept	Принять запрос на установку соединения
Клиентские	
Connect	Установить соединение с удаленным узлом

4.3. Инициализация Winsock

Для работы с библиотекой WINSOCK в исходный код программы необходимо включить директиву компилятора

```
#include <winsock.h> или
#include <winsock2.h>,
```

в зависимости от того, какую версию Winsock вы будете использовать.

Также в сетевой проект должны быть включены все соответствующие библиотечные lib-файлы: Ws2_32.lib или Wsock32.lib. В среде разработки MicroSoft Visual Studio C++ для этого достаточно нажать <Alt-f7>, перейти к закладке "Link" и в списке библиотек, перечисленных в строке "Object/Library modules", в командной строке линкера указать "Ws2_32.lib", отделив ее от остальных символом пробела. Подключение библиотеки можно также выполнить с помощью директивы компилятора:

```
#pragma comment (lib,"Ws2_32.lib").
```

В процессе инициализации приложение должно зарегистрировать себя в этой библиотеке. Для инициализации необходимо

вызвать функцию WSAStartup, определенную следующим образом:

```
int WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData),
```

Параметр wVersionRequested задает версию интерфейса Windows Sockets, необходимую для работы сетевого приложения. Старший байт параметра указывает младший номер версии (minor version), младший байт – старший номер версии (major version). Возможные версии: 1.0, 1.1, 2.0, 2.2... Для «сборки» этого параметра можно использовать макрос MAKEWORD. Например, для определения версии 1.1 используем MAKEWORD (1, 1) или напрямую указываем шестнадцатиричное значение (для нашего примера 0x0101).

Перед вызовом функции WSAStartup параметр lpWSADATA должен содержать указатель на структуру типа WSADATA, в которую будут записаны сведения о конкретной реализации интерфейса Winsock. Размер памяти, выделенной для структуры WSADATA, должен быть не менее 1 Кб.

Примером может служить следующий код:

```
    WSADATA ws;
//...
    if (WSAStartup (MAKEWORD( 1, 1 ), &ws) )
    {       // Error...
        error = WSAGetLastError();
//…}
```

Конкретно говоря, инициализация заключается в сопоставлении номера версии и реально существующей DLL в системе. Фактически, при вызове WSAStartup происходит диалог между программой и модулем WINSOCK.DLL.

Первое: функция WSAStartup позволяет сетевой программе указать требуемую версию Winsock (в примере это 1.1). Она

также позволяет программе получить информацию в структуре типа WSADATA о конкретной реализации Winsock.

Второе: предусмотрена также возможность определять наиболее эффективную версию Winsock для того, чтобы обеспечить разработку программ, использующих более новые и не работающих со старыми версиями Winsock. Обычно программа указывает минимальную версию, с которой она еще может работать, а Winsock – наиболее высокую версию из тех, что он может обеспечить (также в WSADATA). Далее сетевая программа решает, как ей лучше поступить.

Если инициализация состоялась, то вернется нулевое значение, иначе – ненулевое.

Прикладная программа может вызывать WSAStartup несколько раз за время работы. Это может понадобиться, например, если в разных частях программы требуется узнать версию WINSOCK.DLL.Реализация Winsock (WINSOCK.DLL) отводит внутренние статические области памяти для зарегистрировавшейся в ней программы. Каждая программа получает собственные области памяти, к которым можно обращаться только через Winsock API. Области памяти являются внутренними для Winsock точно так же, как и структуры данных сокета.

Перед тем как завершить свою работу, приложение должно освободить ресурсы, полученные у операционной системы для работы с WINSOCK. Для выполнения этой задачи приложение должно вызвать функцию WSACleanup:

```
int WSACleanup (void);
```

Эта функция может возвратить нулевое значение при успехе или значение SOCKET_ERROR в случае ошибки. Для получения кода ошибки можно воспользоваться функцией с именем WSAGetLastError.

Функция WSACleanup органично дополняет WSAStartup. На каждый вызов функции WSAStartup должен приходиться вызов WSACleanup. Как только WSACleanup вызвана в последний раз, Winsock отсоединяет все оставшиеся сокеты, ориентированные на

соединение. Тем не менее все данные, оставшиеся в выходных очередях, отправляются. Прикладной программе необходимо тщательно отслеживать последовательность вызовов `WSAStartup` и `WSACleanup`. Особенно это относится к аварийному завершению программы. Даже в этом случае следует предусмотреть корректное окончание работы с `WINSOCK.DLL`, чтобы сбой в программе не отразился на обслуживающих ее сетевых модулях.

4.4. Создание сокета

После инициализации интерфейса Windows Sockets приложение должно создать один или несколько сокетов, которые будут использованы для передачи данных. В сетевом интерфейсе процессы создания сокета и соединения сокета с компьютером-получателем происходят раздельно.

Для создания сокета используется функция `socket`:

```
SOCKET socket(int domain, int socket_type, int protocol);
```

Эта функция имеет три параметра: вид коммуникационного домена, тип сокета и сам протокол.

Первый параметр задает домен или семейство, к которому принадлежит протокол, например семейство TCP/IP, и форматы адресов. Интерфейс сокетов может обслуживать несколько различных типов сетей одновременно. С помощью этого параметра могут быть выбраны различные семейства протоколов: **`AF_INET`** для сетевого протокола `IPv4`, **`AF_INET6`** для `IPv6` или **`AF_UNIX`** для `Unix`

Режим соединения (датаграммный (`SOCK_DGRAM`), потоковый (`SOCK_STREAM`) или «сырой» сокет (`SOCK_RAW`)) задается вторым параметром – «типом сокета».

Параметр «протокол» определяет протокол, с которым будет работать сокет. Нулевое значение соответствует выбору по умолчанию: `TCP` – для потоковых сокетов и UDP для дейтаграммных. В большинстве случаев нет никакого смысла задавать протокол вручную и обычно полагаются на автоматический выбор по умолчанию.

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

Каждый раз, когда программа вызывает функцию `socket`, реализация сокетов отводит машинную память для новой структуры данных для указания в ней семейства адресов, типа сокета и протокола. В таблице дескрипторов размещается указатель на эту структуру. Она возвращает дескриптор сокета, подобный дескриптору файла. Дескриптор сокета указывает на таблицу, содержащую описание свойств и структуры сокета. Дескриптор, полученный вашей программой от функции `socket`, является индексом (порядковым номером) в таблице дескрипторов.

Структура данных сокета в упрощенном виде показана на рис. 19:

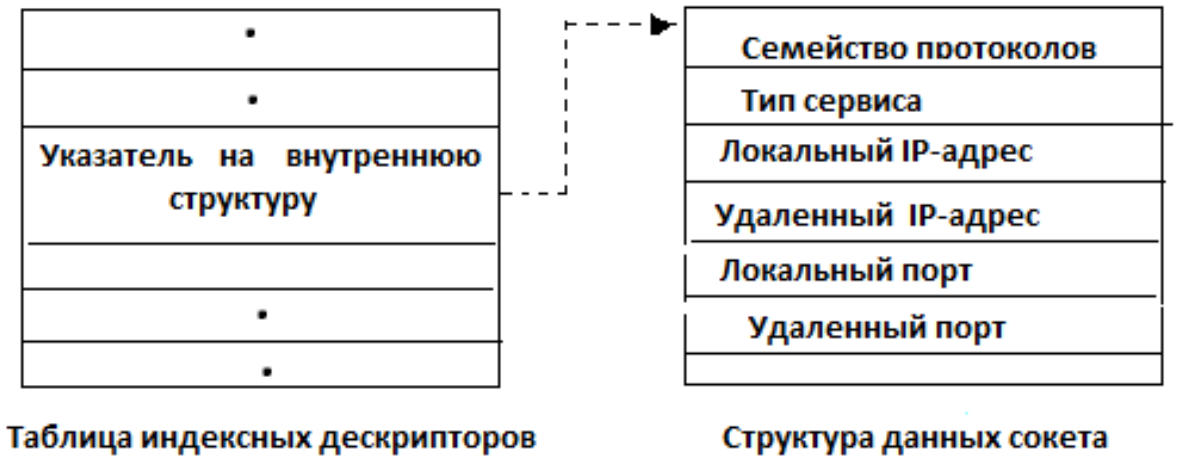


Рис. 19. Упрощенная структура данных сокета

Как видно из рисунка, структура данных сокета содержит поля, куда прежде всего заносятся значения аргументов функции `socket`. Кроме того, структура сокета содержит поля для размещения адресной информации: `IP`-адрес локального компьютера, `IP`-адрес удаленного компьютера, номера локального и удаленного портов.

Пример:

```
    if(INVALID_SOCKET==
(s=socket(AF_INET,SOCK_STREAM,0)))
{
```

```
// Error...
error = WSAGetLastError();
    }
```

При ошибке функция возвращает `INVALID_SOCKET`. В таком случае можно получить расширенную информацию об ошибке, используя вызов `WSAGetLastError()`.

4.5. Закрытие сокета

Для освобождения ресурсов приложение должно закрывать сокеты, которые ему больше не нужны. Процедура закрытия активного соединения происходит с помощью функций `closesocket` и `shutdown`.

Для закрытия соединения и уничтожения сокета предназначена функция

```
int closesocket (socket s),
```

которая в случае удачного завершения операции возвращает нулевое значение. В таком случае соединение разрывается моментально. Вызов `closesocket` имеет мгновенный эффект. После вызова `closesocket` сокет уже недоступен.

Протокол `TCP` позволяет выборочно закрывать соединение любой из сторон, оставляя другую сторону активной. Например, клиент может сообщить серверу, что не будет больше передавать ему никаких данных и закрывает соединение «***клиент > сервер***», однако готов продолжать принимать от него данные до тех пор, пока сервер будет их посылать, т.е. хочет оставить соединение «***клиент < сервер***» открытым. Для этого необходимо вызвать функцию:

```
int shutdown (socket s,int how),
```

передав в аргументе `how` одно из следующих значений:

`SD_RECEIVE` для закрытия канала «***сервер < клиент***»,

SD_SEND для закрытия канала «***клиент < сервер***»,

SD_BOTH для закрытия ***обоих каналов***.

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

4.6. Конфигурация сокета

Сокет является реализацией абстрактной модели конечной точки сетевого соединения. Структура данных сокета, как видно на рис. 19, содержит все элементы, необходимые конечной точке сетевого соединения, что значительно упрощает процесс сетевого соединения. Когда одна программа желает установить связь с другой, программа-передатчик просто отдает свою информацию сокету, а интерфейс сокетов в свою очередь передает ее дальше стеку сетевых протоколов TCP/IP.

Термин «***адрес сокета***» относится не к самому сокету, а к адресам портов и компьютеров, размещенных внутри его структуры данных. Важно, чтобы структура данных была подготовлена к приему как номера порта, так и сетевого адреса компьютера. Вызывая функцию socket, мы вместе с протоколом указываем тип сетевой службы (ориентированный или неориентированный на соединение), но не указываем ни номер порта, ни сетевой адрес. Заполнение адресных полей структуры сокета зависит как от типа сетевой службы по доставке данных, так и от назначения программы. Назначение программы – выполнять функции клиента или сервера. Несмотря на то что одна программа может быть и тем и другим, в один момент времени она может выполнять функцию либо только клиента, либо только сервера.

Итак, каждая сетевая программа вначале создает сокет, вызывая функцию socket. Далее при помощи других функций

сетевого интерфейса сокет конфигурируется или настраивается так, как это нужно сетевой программе.

Параметры сокета

Инициализация сокета заключается в определении значений всех полей структуры сокета. Предварительно все параметры сокета необходимо поместить в структуру типа sockaddr:

```
struct sockaddr
{
u_short sa_family; // семейство протоколов
// (для IPv4 значение AF_INET)
char sa_data[14]; // IP-адрес узла и порт
};
```

Так как для представления IP-адреса используются форматы разного вида и длины в Winsock 2.x, на смену этой структуре пришла структура sockaddr_in, определенная следующим образом:

```
struct sockaddr_in
  {
short  sin_family;  // адресное пространство
                        // (AF_INET для IPv4)
u_short  sin_port;          // порт
struct  in_addr sin_addr;   // IP-адрес
char  sin_zero[8];          // хвост
  };
```

При переопределении адресной структуры произошла замена беззнакового короткого целого на знаковое короткое целое для представления семейства протоколов, адрес узла теперь представлен в виде трех полей – sin_port (номера порта), sin_addr (IP-адреса узла) и «хвоста» из восьми нулевых байт, который остался от четырнадцатисимвольного массива sa_data. Дополнительные байты дают возможность структуре sockaddr не привязываться к конкретному сетевому протоколу и работать с различными

сетями. Адреса же некоторых сетей требуют для своего представления гораздо больше четырех байтов (например, в сетях IPv6 адрес занимает 12 байтов).

Поле sin_port определяет номер порта, который будет использоваться для передачи данных. Заметим, что порт – это просто идентификатор программы, выполняющей обмен на сети. На одном узле может одновременно работать несколько программ, использующих разные порты. Особенностью поля sin_port является использование так называемого сетевого формата данных. Универсальный сетевой формат данных удобен при организации глобальных сетей, так как в узлах такой сети могут использоваться компьютеры с различной архитектурой.

Для выполнения преобразований из обычного формата в сетевой и обратно в интерфейсе Winsock предусмотрен специальный набор функций. В частности, для заполнения поля sin_port можно использовать функцию htons, выполняющую преобразование 16-разрядных данных из формата Intel в сетевой формат.

Например:

```
#define serv_port 5000
sockaddr_in sin;
sin.sin_port = htons(serv_port);
```

Внутренняя структура in_addr определяется следующим образом:

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

Как видно, она состоит из одного IP-адреса, записанного в трех вариантах: четырехбайтовой последовательности (S_un_b), пары двухбайтовых слов (S_un_w) и одного длинного целого (S_addr).

При инициализации сокета в этой структуре необходимо указать IP-адрес, с которым будет работать данный сокет. Если сокет будет работать с любым адресом (например, в случае приложения-сервера, доступного из узлов с любым адресом), адрес для сокета можно указать следующим образом:

```
sockaddr_in sin;
sin.sin_addr.s_addr = INADDR_ANY;
```

В последнем примере использовалось существующее в файле Winsock2.h следующее определение:

```
#define s_addr S_un.S_addr,
```

отмечающее, что s_addr – эквивалент S_addr, т. е. IP-адресу, записанному в виде длинного целого. Этот факт позволяет в сетевых программах использовать более удобное краткое обращение к IP-адресу в структуре in_addr.

Дейтаграммный протокол UDP позволяет посылать пакеты данных одновременно всем рабочим станциям в широковещательном режиме. Для этого необходимо указать адрес как INADDR_BROADCAST.

В том случае, если сокет будет работать с определенным адресом IP (например, для клиента, который будет обращаться к серверу с конкретным адресом), в указанную структуру необходимо записать реальный адрес.

Для преобразования IP-адреса, записанного в виде текстовой строки (наподобие «198.64.8.1»), в четырехбайтовый сетевой формат предназначена функция:

```
unsigned long inet_addr (const char far * cp ).
```

Она принимает указатель на текстовую строку и в случае успешной операции преобразует эту строку в четырехбайтовый IP адрес или –1, если это невозможно. Возвращенный функцией результат можно присвоить элементу структуры sockaddr_in следующим образом:

```
sockaddr_in dest_addr;
        …
dest_addr.sin_addr.s_addr=
          inet_addr("195.161.42.222");
```

При использовании структуры sockaddr это будет выглядеть так:

```
struct sockaddr demo_addr;
((unsigned int *)(&demo_addr.sa_data[0]+2))[0] =
inet_addr("195.161.42.222");
```

В случае ошибки функция возвращает значение INADDR_NONE. К ошибке приводит, например, попытка передать inet_addr доменное имя узла, что можно использовать для проверки.

Однако чаще всего пользователь работает с доменными именами, используя сервер DNS или файл hosts. В этом случае вначале необходимо воспользоваться функцией gethostbyname для получения IP-адреса. Если же указанный узел найден в базе DNS или в файле hosts, функция gethostbyname возвращает указатель на структуру hostent, описанную ниже:

```
struct  hostent
{
char  far * h_name;      // имя узла
char  far * far * h_aliases;
// список альтернативных имен
short h_addr type;           // тип адреса узла
short h_length;              // длина адреса
char far * far * h_addr_list; // список адресов
};
```

```
#define h_addr   h_addr_list[0]   // адрес
```

Искомый адрес находится в первом элементе списка h_addr_list[0], на который также можно ссылаться при помощи h_addr. Длина поля адреса находится в поле h_length.

В качестве примера используем данную функцию для получения адреса узла с именем "fpm.kubsu.ru":

```
hostent * ph;   sockaddr_in demo_sin;
ph = gethostbyname ("fpm.kubsu.ru");
if(ph == NULL)
{
closesocket  (demo_socket);
cout<< "gethostbyname  error";
return 1;
}
((unsigned long *)&demo_sin.sin_addr)[0]=
((unsigned long **)ph->h_addr_list)[0][0];
```

В случае ошибки функция gethostbyname возвращает NULL. При этом причину ошибки можно выяснить, проверив код возврата функции WSAGetLastError.

Известно, что с некоторыми доменными именами связано сразу несколько IP-адресов. В случае неработоспособности одного узла клиент может попробовать подключиться к другому или просто выбрать узел с наибольшей скоростью обмена. Использование поля h_addr, очевидно, позволяет использовать только первый IP-адрес в списке и игнорировать все остальные! Конечно, более эффективным будет наличие в сетевых программах возможности подключения к остальным IP-адресам при невозможности установить соединение с первым.

Еще раз отметим, что функция gethostbyname ожидает на входе ***только*** доменные имена, но не цифровые IP-адреса. Поэтому произвольную строку с адресной информацией правильнее

вначале передать на обработку функции inet_addr, если эта функция возвращает ошибку, то вызываем gethostbyname.

Пример заполнения структуры sockaddr_in:

```
        sockaddr_in dest_addr;
        dest_addr.sin_family=AF_INET;
        dest_addr.sin_port=htons(port);
        hostent *hst;
/* преобразование IP адреса из символьного в сетевой
формат */
        if (inet_addr(serveraddr)!=INADDR_NONE)
dest_addr.sin_addr.s_addr=inet_addr(serveraddr);
        else
/* попытка получить IP адрес по доменному имени сер-
вера */
    if (hst=gethostbyname(serveraddr))
/* hst->h_addr_list является массивом указателей на
адреса */
        ((unsigned long *)&dest_addr.sin_addr)[0]=
        ((unsigned long **)hst->h_addr_list)[0][0];
    else
        {
        cout<< "invalid address \n" << serveraddr;
            closesocket(my_sock);
            WSACleanup();
            cin.get();
            return -1;
        }
```

Задача определения имени узла по его адресу возникает, например, у серверов, желающих «в лицо» знать своих клиентов. Для решения этой обратной задачи – определения доменного имени по IP-адресу – предусмотрена функция:

```
    struct hostent far * gethostbyaddr
(const char far * addr, int len, int type),
```

которая во всем аналогична gethostbyname, за исключением, что ее аргументом является не указатель на строку, содержащую имя, а указатель на поле с сетевым форматом IP-адреса. Еще два аргумента задают его длину и тип сетевого формата (соответственно **4** и AF_INET для адресов IPv4).

Для преобразования IP-адреса, записанного в сетевом формате, в текстовую строку с числовым адресным форматом предусмотрена функция inet_ntoa:

```
char far * inet_ntoa (struct in_addr in),
```

которая принимает на вход структуру in_addr, а возвращает указатель на строку, если преобразование выполнено успешно и ноль в противном случае.

Приведем пример получения доменного имени и числовой адресной строки на основе сетевого адреса, предварительно размещенного в адресной структуре cl_addr:

```
    sockaddr_in cl_addr;
    hostent *hst;
    …
hst=gethostbyaddr((char*)&cl_addr.sin_addr.s_addr,
                  4,AF_INET);
    if(hst) cout<< hst->h_name ; //доменное имя
    else    cout<< "";
cout<<inet_ntoa(cl_addr.sin_addr); //числовой адрес
```

На практике можно с одинаковым успехом пользовать исходную структуру sockaddr и модифицированную sockaddr_in. Однако, поскольку прототипы остальных функций не изменились, при использовании sockaddr_in придется постоянно выполнять явные преобразования типов переменных. Например,

```
    sockaddr_in dest_addr;
    // заполнение структуры dest_addr
    …
```

```
    connect (mysocket, (sockaddr*) &dest_addr,
sizeof(dest_addr)).
    …
```

4.7. Сетевой формат данных

Среди производителей процессоров нет единого мнения насчет порядка следования младших и старших байт. Так, у микропроцессоров **Intel** младшие байты располагаются по меньшим адресам, а у микропроцессоров **Motorola** – наоборот. Этот факт вызывает проблемы при межсетевом взаимодействии, поэтому был введен специальный ***сетевой порядок*** *байт*, предписывающий старший байт передавать первым.

Для преобразований чисел из сетевого формата в формат локального хоста и наоборот предусмотрены четыре функции – первые две манипулируют короткими целыми (16-битными словами), а две последние – длинными (32-битными двойными словами):

```
u_short ntohs (u_short netshort);
u_short htons (u_short hostshort );
u_long ntohl (u_long netlong );
u_long htonl (u_long hostlong);
```

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

Отметим, что все значения, возвращенные socket-функциями, уже находятся в сетевом формате и "вручную" их преобразовывать ***нельзя***, так как это преобразование исказит результат и приведет к неработоспособности.

Чаще всего к вызовам этих функций прибегают для преобразования номера порта согласно сетевому порядку. Например:

```
dest_addr.sin_port = htons(110).
```

4.8. Привязка адреса к сокету

После заполнения структуры sockaddr параметрами сокета, следует выполнить привязку адреса к сокету, т. е. скопировать информацию в структуру сокета.

Связывание осуществляется вызовом функции:

```
int bind (socket s, const struct sockaddr far* name, int namelen);
```

Первым аргументом передается дескриптор сокета, возвращенный функцией socket, за ним следуют указатель на структуру sockaddr и ее длина.

В случае ошибки функция bind возвращает значение SOCKET_ERROR. Анализ причин ошибки следует выполнять при помощи функции WSAGetLastError.

Реализация процесса привязки адресной информации на стороне сервера может отличаться от реализации ее на стороне клиента.

Программа-сервер ожидает появления запроса от клиента. Транспортный уровень TCP/IP для общения с приложениями (клиентами и серверами) использует порт протокола. Прежде чем сервер сможет использовать сокет, он должен связать его с локальным адресом. Локальный, как, впрочем, и любой другой адрес Интернета, состоит из IP-адреса узла и номера порта. Если сервер имеет несколько IP-адресов, то сокет может быть связан как со всеми ними сразу (для этого вместо IP-адреса следует указать константу INADDR_ANY), так и с каким-то конкретным одним. Используемый для передачи данных порт сервера должен быть определен и известен всем клиентам. Поэтому ему приходится осуществлять связывание «вручную», используя для этого функцию bind.

Клиент при инициализации сокета также должен связывать его с локальным адресом. Но это может быть реализовано по-разному.

В случае ориентированного на соединение взаимодействия клиент так же, как и сервер, может выполнить связывание с помощью функции bind, задав при этом конкретные значения IP-адреса и номера порта клиента. Однако за клиента это может сделать

и функция connect, ассоциируя сокет с одним из портов, наугад выбранных из диапазона 1024-5000.

Неориентированный на соединение клиент также должен прослушивать порт протокола. Программы, пользующиеся неориентированными на соединение протоколами, не устанавливают прямого сетевого соединения. Из этого следует, что неориентированный на соединение клиент также должен прослушивать порт протокола на предмет появления ответных дейтаграмм. Как и программы-серверы, клиенты, не ориентированные на соединение, должны использовать функцию bind, чтобы зарегистрировать порт протокола в интерфейсе сокетов. Другими словами, неориентированный на соединение клиент, так же, как и сервер, указывает интерфейсу сокетов тот порт, который он будет использовать для доставки данных. Реализация сокетов организует в свою очередь интерфейс между таким клиентом и программным модулем UDP транспортного уровня.

4.9. Соединение сокета

Сокет представляет абстракцию, пользуясь которой можно настраивать и программировать конечные точки сетевого соединения. Ориентированные на соединение протоколы организуют между конечными точками виртуальную цепь. То есть в этом случае соединение между конечными точками теоретически не отличается от выделенного двухточечного соединения. Протокол TCP (ориентированный на соединение) транспортного уровня стека TCP/IP обслуживает виртуальную цепь (держит соединение открытым), обмениваясь сообщениями-подтверждениями о доставке данных между двумя конечными точками. В результате ориентированной на соединение программе-клиенту в сети TCP/IP все равно, с которого локального номера порта передаются ее данные. Программа-клиент может принимать данные на любом порту протокола. Поэтому в большинстве случаев ориентированные на соединение программы-клиенты используют динамический номер локального порта протокола, т. е. номер порта выбирается операционной системой автоматически.

Ориентированная на соединение программа-клиент вызывает функцию connect, чтобы настроить сокет на сетевое соединение с сервером. Функция connect размещает информацию о локальной и удаленной конечных точках соединения в структуре данных сокета. Функция connect требует, чтобы были указаны: дескриптор сокета клиента, указатель на структуру, содержащую информацию об удаленном компьютере, и длина этой структуры.

Прототип функции connect:

```
int connect (SOCKET socket_handle,
const struct sockaddr FAR *remote_socket_address, int address_length);
```

Первый параметр функции socket_handle – дескриптор сокета, полученный ранее от функции socket. Функция socket всегда вызывается до того, как устанавливается соединение. Дескриптор сокета указывает программному обеспечению, какая именно структура данных в таблице дескрипторов имеется в виду. Дескриптор также сообщает о том, куда нужно записать информацию об адресах удаленного участника соединения.

Второй параметр функции задает адрес удаленного сокета, является указателем на структуру адресных данных сокета специального вида. Информация об адресе, хранящаяся в структуре, зависит от конкретной сети, т. е. от семейства протоколов, которое мы используем. Адресная структура данных сокета содержит семейство адресов, порт протокола и адрес сетевого компьютера. Функция connect записывает эту информацию в таблицу дескрипторов сокетов, на которую указывает соответствующий дескриптор сокета (первый параметр функции connect).

До того, как вызвать функцию connect, информацию об адресах удаленного компьютера нужно занести в структуру адресных данных сокета типа sockaddr. Другими словами, функция connect должна знать сетевой адрес и номер порта удаленного компьютера. Местный IP-адрес в явном виде можно не указывать: реализация сокетов операционной системы может самостоятельно разместить адрес вашего компьютера.

Третий параметр функции сообщает интерфейсу длину структуры адресных данных удаленного сокета (второй параметр), измеренную в байтах. Содержимое (и длина) этой структуры зависит от конкретной сети. Зная длину структуры, интерфейс сокетов представляет, сколько памяти отведено для хранения этой структуры. Когда реализация сокетов выполняет функцию `connect`, она извлекает количество байтов, указанное третьим параметром из буфера данных, на который указывает параметр `remote_socket_address` (*адрес удаленного сокета*). Функция `connect` устанавливает прямое соединение с удаленным сетевым компьютером. Это необходимо, если используется ориентированный на соединение протокол. Если к моменту выполнения `connect` используемый им сокет не был привязан к адресу посредством `bind`, то такая привязка будет выполнена автоматически.

Если протокол не ориентирован на соединение, он никогда не устанавливает его напрямую. Неориентированный на соединение протокол передает данные в дейтаграммах — он никогда не передает их потоком байтов. Точно так же программа-сервер никогда не начинает соединение первой. Можно создать программу-сервер, работающую по неориентированному на соединение протоколу, однако и в этом случае она будет пассивно прослушивать порт протокола, ожидая появления запроса от клиента. Другими словами, прямое соединение инициируется клиентом, а не сервером.

Ключевое сходство между любой неориентированной на соединение программой и программой-сервером, ориентированной на соединение, состоит в том, что обе они прослушивают порт протокола. Например, ориентированные и неориентированные на соединение программы-серверы должны ждать появления запроса клиента на порту протокола. Точно так же, поскольку неориентированные на соединение клиенты не устанавливают соединения напрямую с удаленным компьютером, они должны ожидать появления дейтаграммы-ответа на собственный запрос на порту протокола. Во всех этих случаях заранее нельзя определить адресные данные удаленного компьютера.

В качестве примера приведем фрагмент кода для соединения клиента с сервером:

```
// Объявим структуру для хранения адреса сервера
    sockaddr_in s_addr;
// Заполним ее информацией
    ZeroMemory (&s_addr, sizeof (s_addr));
// вначале очистим структуру
s_addr.sin_family = AF_INET;   // тип адреса IPv4
/* адрес сервера переводим из десятичного формата в
сетевой */
            s_addr.sin_addr.s_addr =
        inet_addr ("193.108.128.226");
// Номер порта представляем в сетевом формате
        s_addr.sin_port = htons (1111);
// выполняем соединение с сервером
    if(SOCKET_ERROR==
(connect (s,(sockaddr *)&s_addr, sizeof(s_addr))))
    {   // Error...
            error = WSAGetLastError();
    }
```

Connect возвращает 0, если вызов прошел успешно. При ошибке функция connect возвращает SOCKET_ERROR= –1. При положительном исходе сокет клиента s связан с удаленным процессом-сервером и может посылать / принимать данные только с него.

Загруженный сервер может отвергнуть попытку соединения, поэтому в некоторых видах программ необходимо предусмотреть повторные попытки соединения.

В режиме взаимодействия без установления соединения необходимости в выполнении системного вызова connect нет. Однако его выполнение в таком режиме не является ошибкой – просто меняется смысл выполняемых при этом действий: устанавливается адрес «по умолчанию» для всех последующих посылок дейтаграмм, что позволяет дейтаграмному сокету пользоваться для обмена данными не только функциями sendto, recvfrom, но и более удобными и компактными send и recv.

4.10. Серверные сокеты

Для установления связи «клиент-сервер» ориентированная на соединение программа-сервер вызывает функции `listen` и `accept`. Первая переводит сервер в режим пассивного ожидания запроса. Функция `accept`, наоборот, заставляет сокет программы установить соединение.

Функция `listen`. Что происходит, когда появляется очередной запрос от клиента, а сервер еще не закончил обработку предыдущего. Например, когда программа-клиент пытается послать еще один запрос в то время, как последовательный сервер еще не закончил обработку предыдущего, или когда клиент посылает запрос до того момента, как параллельный сервер закончил создание нового процесса – обработчика предыдущего запроса. В этом случае сервер может отвергнуть или игнорировать поступивший запрос. Для того чтобы обработка запросов была эффективнее, существует функция:

```
int listen(SOCKET s_handle, int q_length);
```

Функция `listen` имеет два параметра: дескриптор сокета сервера `s_handle` и длина очереди клиентов `q_length`. Длина очереди обозначает максимальное количество запросов, которое может поместиться в ней.

Функция `listen` не только переводит сокет в пассивный режим ожидания, но и подготавливает его к обработке множества одновременно поступающих запросов. Другими словами, в системе организуется очередь поступивших запросов, и все запросы, ожидающие обработки сервером, помещаются в нее, пока освободившийся сервер не выберет его. Размер очереди ограничивает количество одновременно обрабатываемых соединений, поэтому к его выбору следует подходить «с умом». Если очередь полностью заполнена, очередной клиент при попытке установить соединение получит отказ (`TCP` пакет с установленным флагом `RST`). Максимально разумное количество подключений определяется производительностью сервера, объемом оперативной памяти и т. д.

Дейтаграммные серверы не вызывают функцию listen, так как работают без установки соединения и сразу же после выполнения связывания могут вызывать recvfrom для чтения входящих сообщений.

Пример использования функции listen:

```
…
if (listen(listen_socket, SOMAXCONN) ==
SOCKET_ERROR)
{
cerr << "listen failed with error: " <<
WSAGetLastError() << "\n";
closesocket(listen_socket);
WSACleanup();
return 1;
}
…
```

В этом примере при вызове функции в качестве второго параметра использовалась системная константа SOMAXCONN, значением которой является максимально возможное число одновременных TCP-соединений. Это ограничение работает на уровне ядра ОС.

Функция accept. Извлечение запросов на соединение из очереди осуществляется функцией

```
SOCKET accept (SOCKET s, struct sockaddr far*
addr, int far* addrlen)
```

При вызове accept требуется указывать три параметра: дескриптор сокета, указатель на адресную структуру и указатель на длину этой структуры. Дескриптор сокета указывает на сокет, который прослушивается сервером. Когда на сокете, контролируемом функцией accept, появляется очередной запрос клиента, программное обеспечение-реализация сокетов автоматически создает новый сокет и немедленно соединяет его с процессом-клиентом. Функция accept возвращает дескриптор нового сокета, а в структуру sockaddr заносит сведения о подключившемся клиенте (IP-адрес и порт). Сокет, на который поступил запрос, освобождается

и продолжает работу в режиме ожидания запросов от любого сетевого компьютера.

Если в момент вызова `accept` очередь пуста, функция не возвращает управление до тех пор, пока с сервером не будет установлено хотя бы одно соединение. В случае возникновения ошибки функция возвращает отрицательное значение.

4.11. Передача данных

После конфигурирования сокет можно использовать для сетевого соединения. Процесс сетевого соединения подразумевает посылку и прием информации. Интерфейс сокетов обеспечивает четыре функции для передачи данных через сокет. Двум из них требуется указывать адрес назначения в качестве аргумента, а двум остальным – нет. Основное различие между этими двумя группами состоит в их ориентированности на соединение. Все четыре функции из интерфейса сокетов описаны в табл. 8.

Таблица 8

Функция интерфейса сокетов	Описание
`send`	Передает данные через соединенный сокет. Использует некоторые флаги для управления поведением сокета.
`resv`	Прием данных через соединенный сокет. Использует некоторые флаги для управления поведением сокета.
`sendto`	Передает данные через несоединенный сокет. Использует буфер данных.
`resvfrom`	Прием данных через несоединенный сокет. Использует буфер данных.

Передача данных через потоковый сокет

Виртуальные цепи образуются ориентированными на соединение протоколами. При этом соединение между конечными точками сети выглядит как выделенное двухточечное соединение.

После того как программное обеспечение установило соединение, прикладная программа может обмениваться данными в простом последовательном потоке байтов. Функции интерфейса сокетов, осуществляющие ориентированную на соединение передачу данных, не требуют от прикладных программ указывать адрес назначения в качестве аргумента. Вся информация об адресах назначения (сетевой адрес и номер порта) заносится в структуру сокета на стадии его конфигурации. В течение сеанса связи через ориентированный на соединение сокет все заботы, связанные с адресами удаленного сокета и управления интерфейсом с транспортным уровнем, берет на себя реализация сокетов.

После того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными при помощи функций

```
int send (socket s, const char far * buf,
    int len,int flags)
```

и

```
int recv (socket s, char far* buf, int len,
    int flags)
```

(для посылки и приема данных соответственно).

Аргументы функций аналогичны. В качестве первого аргумента задается дескриптор сокета, откуда принимаются или куда передаются данные. Второй параметр – буфер сообщения, указывает на буфер, т. е. область памяти, в которой расположены предназначенные для передачи данные. Прикладная программа должна предварительно отвести память для этого буфера, а затем либо при передаче заполнить его данными, либо при приеме извлекать информацию из указанного буфера. Третий параметр вызова функции обозначает длину буфера, т. е. количество данных для передачи. Функции `send`, `resv` предназначены только для потоковых сокетов – они не требуют указания адреса назначения.

Функция `send` возвращает управление сразу же после ее выполнения независимо от того, получила ли принимающая сторона наши данные или нет. При успешном завершении функция возвращает количество *передаваемых* (*не переданных!*) данных – т. е. успешное завершение еще не свидетельствует об успешной

доставке! В общем-то протокол TCP (на который опираются потоковые сокеты) гарантирует успешную доставку данных получателю, но лишь при условии, что соединение не будет преждевременно разорвано. Если связь прервется до окончания пересылки, данные останутся непереданными, но вызывающий код не получит об этом никакого уведомления! А ошибка возвращается лишь в том случае, если соединение разорвано ***до*** вызова функции send!

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

Winsock протоколы (TCP / UDP) могут «отправить» пакет и максимального размера MAX_PACKET_SIZE (65535 байт). Однако обычно реальный размер IP-пакета меньше объема передаваемых данных для конкретного типа сети, поэтому данные фрагментируются. При этом фрагментированные данные могут идти с задержками.

Например, если у удалённого Web-сервера запросить файл достаточно большого размера, то вначале будет получена только первая порция данных от сервера. Остальная часть данных будет, скорее всего, утеряна. Выход из данного положения очень прост. Он основан на знании принципа работы HTTP и TCP/IP протоколов, плюс некоторых особенностей Winsock. Как правило, Web-сервер, передав последнюю порцию данных, закроет соединение. А функция recv в таком случае, после получения последнего фрагмента данных, возвращает ноль. Вывод напрашивается сам: необходимо читать входящие данные до тех пор, пока функция recv не вернет ноль.

Вот возможный вариант кода:

```
    int len;
    do
{
If (SOCKET_ERROR==(len = recv(s,(char*)&buff,
            MAX_PACKET_SIZE,0)))
    return -1;
    buff[len]='0';
    cout<<buff;
} while (len!=0);
```

При такой организации считывания данных потерь не будет.

Работой обеих функций можно управлять с помощью *флагов*, передаваемых в одной переменной типа `int` третьим слева аргументом `flags`.

Флаг `MSG_PEEK` заставляет функцию `recv` просматривать данные вместо их чтения. Просмотр, в отличие от чтения, не уничтожает просматриваемых данных.

Флаг `MSG_OOB` предназначен для передачи и приема *срочных* (*out of band*) данных. `TCP/IP` имеет режим передачи данных вне основной полосы пропускания (данных для неотложной обработки). Эти данные имеют приоритет выше, чем у остальных. Если данные передавались функцией `send` с установленным флагом `MSG_OOB`, для их чтения флаг `MSG_OOB` функции `recv` также должен быть установлен.

Как отмечалось, при использовании протокола `TCP` (сокеты типа `SOCK_STREAM`) есть вероятность получить меньше данных, чем было передано, так как ещё не все данные были переданы. В этом случае нужно либо дождаться, когда функция `recv` возвратит 0 байт (как в предыдущем примере), либо выставить флаг `MSG_WAITALL` для функции `recv`, что заставит её дождаться окончания передачи. Для остальных типов сокетов флаг `MSG_WAITALL` ничего не меняет (например, в `UDP` весь пакет – это сообщение целиком).

Передача данных через датаграммный сокет

У дейтаграммного сокета для передачи данных есть свои функции:

```
    int sendto (socket s, const char far * buf,
int len,int flags, const struct sockaddr far * to,
int tolen)
```

и

```
    int recvfrom (socket s, char far* buf, int len,
int flags, struct sockaddr far* from,
int far* fromlen ).
```

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

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

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

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

4.12. Процесс целиком

На рис. 20 изображены системные вызовы интерфейса сокетов, как правило, используемые программами, ориентированными на соединение. Левая часть диаграммы показывает вызовы на стороне сервера, а правая – на стороне клиента. Линии и стрелки между модулями сервера и клиента изображают поток сетевых сообщений между двумя программами.

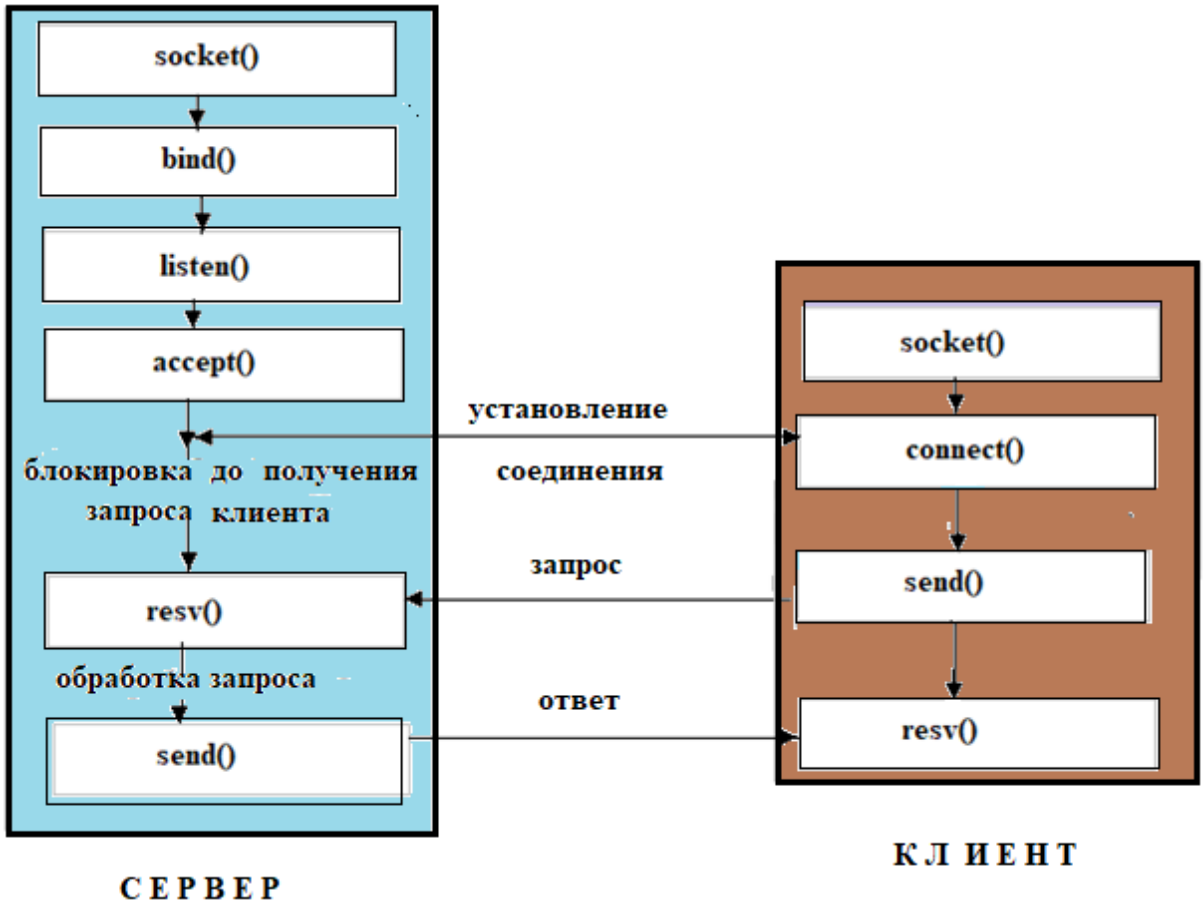


Рис. 20. Потоковые сокеты при сетевом взаимодействии

Программа-сервер создает сокет, вызывая функцию `socket`. Строго говоря, программа сервер запрашивает у интерфейса сокетов отвести структуру данных сокета и возвратить дескриптор

сокета, который будет использован для дальнейших вызовов сетевых функций. Далее сервер привязывает сокет к локальному номеру порта протокола.

Вызов `listen` требует, чтобы сокет прослушивал порт на предмет входящих соединений и подтверждений о доставке. То есть вызов `listen` переводит сокет в режим пассивного ожидания соединения. Ожидающий соединения сокет высылает каждому передатчику сообщение-подтверждение о том, что сетевой компьютер принял запрос на установление соединения.

Однако на самом деле это не означает, что пассивный сокет принял запрос. Чтобы действительно принять запрос и установить соединение, программа должна вызвать функцию `accept`.

Как показано на рис. 20, программа-клиент также создает сокет, вызывая функцию `socket`. Однако в отличие от сервера ориентированной на соединение программе-клиенту нет дела до номера порта протокола, который она получит. То есть ей незачем явно вызывать функцию `bind` и указывать конкретный номер порта. Вместо этого программа-клиент, ориентированная на соединение, вызывает функцию `connect`, которая связывает сокет клиента с автоматически подобранным произвольным номером порта и устанавливает соединение с сервером. После установления соединения передача данных происходит при помощи функций `send` и `recv`.

Рисунок 21 иллюстрирует системные вызовы функций, предназначенных для неориентированных на соединение протоколов. Дейтаграммный сервер вызывает функции **`socket`** и **`bind`** так же, как и сервер, ориентированный на соединение. Заметим, что при дейтаграммном взаимодействии программа-клиент также вызывает функцию `bind`, но не вызывает `connect`. Поскольку образованный сокет не соединен, для чтения данных программа-клиент использует функцию `recvfrom` вместо `recv`. Функция `recvfrom` не ожидает соединения. Вместо этого она обрабатывает любые данные, появившиеся на соответствующем (связанном с ней) порту протокола. Получив дейтаграмму из сокета, функция `recvfrom` записывает как ее содержимое, так и сетевой адрес, с которого она получена.

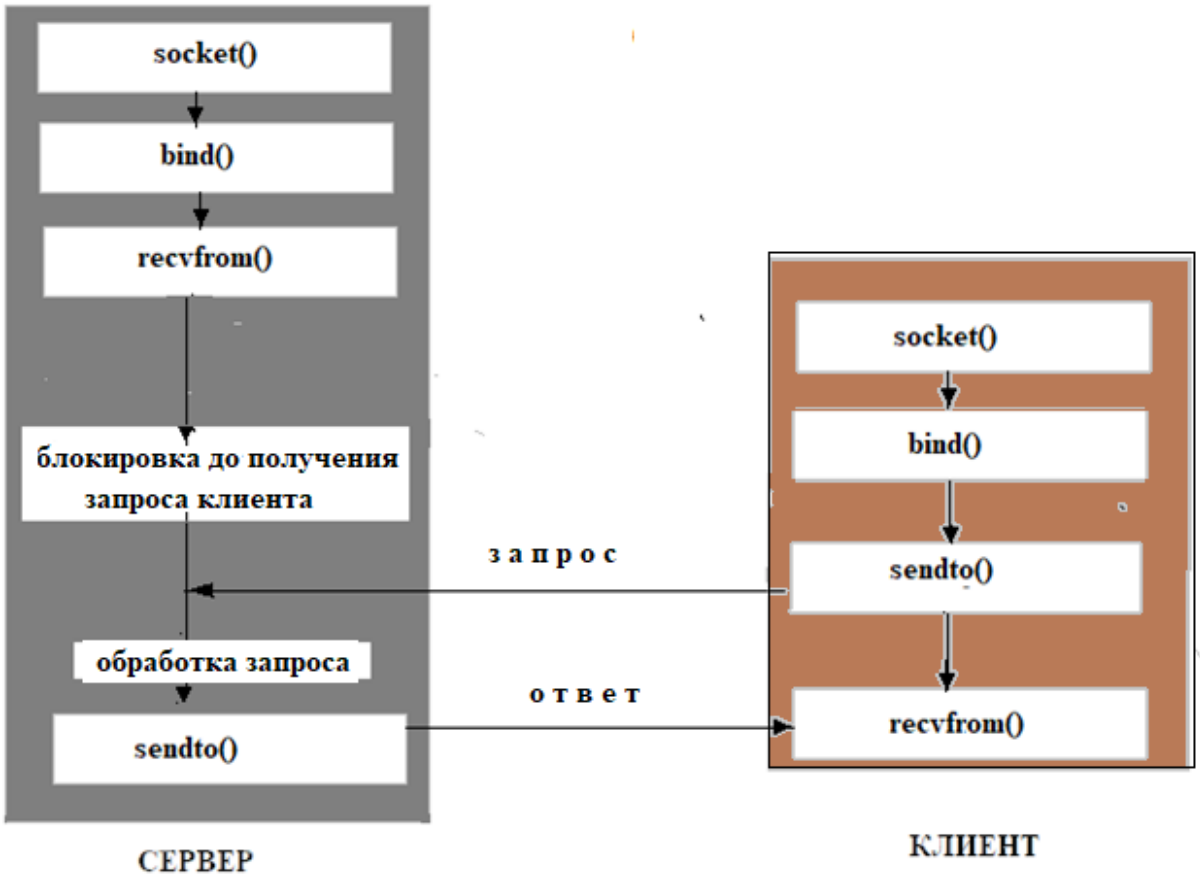


Рис. 21. Дейтаграммные сокеты при сетевом взаимодействии

Для передачи дейтаграмм используется функция `sendto`, требующая от программы указать адрес назначения сообщения в качестве одного из аргументов.

Программы серверы и клиенты используют сетевой адрес для идентификации процесса передатчика или приемника дейтаграммы. Ответную дейтаграмму, как правило, сервер посылает по адресу, ранее извлеченному функцией `recvfrom` из пришедшей дейтаграммы.

Применение интерфейса внутренней петли

При отладке распределенных приложений удобно использовать *интерфейс внутренней петли*, что позволяет имитировать сетевой обмен данными между процессами распределенного приложения на одном компьютере.

Как уже отмечалось, для интерфейса внутренней петли зарезервирован IP-адрес 127.0.0.1. В формате TCP/IP этот адрес можно также задать с помощью определенной в Winsock2.h константы INADDR_LOOPBACK. Напомним также, что символическое имя localhost является зарезервированным и предназначено для обозначения собственного имени компьютера. Если с помощью функции gethostbyname получить адрес компьютера с именем localhost, то это будет собственный IP-адрес компьютера. Напомним, что для получения действительного собственного имени компьютера (DNS-имени) можно использовать функцию gethostname.

Используя интерфейс внутренней петли работу всех приведенных в данном издании приложений, выполняющих обмен данными в сети TCP/IP, можно протестировать на одном компьютере (используя loopback-адрес в десятичном формате 127.0.0.1 или символьное имя localhost).

4.13. Примеры сетевых взаимодействий на основе socket-интерфейса

В данном параграфе рассматриваются два примера использования socket-интерфейса: взаимодействие с установлением логического соединения (TCP-сокеты) и дейтаграммное взаимодействие (UDP-сокеты).

Предлагаемые тексты модельных программ предназначены для иллюстрации логики взаимодействия программ через сеть, в силу чего в программах значительный акцент на содержательную часть программ не производится.

Потоковые TCP-сокеты. Представленный модельный пример демонстрирует создание виртуального соединения между клиентом и сервером, используя протокол TCP.

Логика взаимодействия проста и сводится к следующему: сервер, приняв запрос на соединение, передает клиенту классический вопрос «Who are you?». Клиент, получив сообщение от сервера, выводит его в консоль и направляет серверу свое ответное

сообщение, далее процесс повторяется. Диалог продолжается до тех пор, пока клиент не передаст серверу сообщение «Bye».

В примере задействован интерфейс внутренней петли, что позволяет тестировать работу распределенного приложения на одном компьютере. Для функционирования приложения в условиях реальной сети надо в коде программ клиента и сервера адрес внутренней петли, указанный в примере для сервера, заменить на адрес конкретного узла сети, на котором планируется установить сервер.

В текстах программ смысл указанных команд кратко поясняется комментариями. За более полным пониманием обращайтесь к теоретическому материалу по интерфейсу WinSock, изложенному ранее.

Код TCP-клиента:

```
#include <iostream>
#include <winsock2.h>
#include <string>
#include <windows.h>
#pragma comment (lib, "Ws2_32.lib")
using namespace std;
#define SRV_HOST "localhost" //имя сервера
#define SRV_PORT 1234        // порт сервера
#define CLNT_PORT 1235       // порт клиента
#define BUF_SIZE 64         // размер буфера

int main()
{
// инициализация интерфейса
char buff[1024];
if (WSAStartup(0x0202,(WSADATA *) &buff[0]))
{    // Ошибка!
cout<<"Error WSAStartup\n"<<WSAGetLastError();
return -1;
}
// создание сокета клиента
SOCKET s;
```

```
    int from_len;
    char buf[BUF_SIZE]={0};
    hostent * hp;
    sockaddr_in clnt_sin, srv_sin;
// создание сокета клиента
    s = socket (AF_INET, SOCK_STREAM, 0);
// определение параметров клиента
    clnt_sin.sin_family = AF_INET;
    clnt_sin.sin_addr.s_addr = 0;
    clnt_sin.sin_port = htons(CLNT_PORT);
// связывание сокета с адресом клиента
    bind(s,(sockaddr*)&clnt_sin, sizeof(clnt_sin));
// определение параметров сервера
    hp=gethostbyname(SRV_HOST);
    srv_sin.sin_port=htons(SRV_PORT);
    srv_sin.sin_family = AF_INET;
    ((unsigned long *)&srv_sin.sin_addr)[0]=
    ((unsigned long **)hp->h_addr_list)[0][0];
    // соединение с сервером
    connect(s,(sockaddr*)&srv_sin,sizeof(srv_sin));
    string mst;
// диалог с сервером, пока клиент не передаст "Bye"
    do
    {
// получение сообщения от сервера
    from_len = recv(s,(char *)&buf, BUF_SIZE,0);
    buf[from_len]=0;
    cout << buf <<endl;
// отправка сообщения серверу
    getline(cin, mst);
    int msg_size = mst.size();
    send (s, (char *)& mst[0], msg_size,0);
    }
    while (mst!="Bye");
    // разрыв соединения
    cout << "exit"<< endl;
```

```
    closesocket(s);
    return 0;
    }
```

В представленной программе задается конкретный порт для клиента, а связывание сокета клиента с адресом локального узла происходит «*вручную*» путем использования функции `bind`. Такое поведение возможно, но не типично для клиента. Как указывалось, для клиента, как правило, операционная система динамически выделяет номер порта, а связывание локального адреса с сокетом клиента выполняет функция `connect` при соединении с сервером. Именно это мы увидим в большинстве примеров клиент-серверных приложений, представленных в данной работе.

Код TCP-сервера:

```
    #include <iostream>
    #include <winsock2.h>
    #include <windows.h>
    #include <string>
    #pragma comment (lib, "Ws2_32.lib")
    using namespace std;
    #define SRV_PORT 1234
    #define BUF_SIZE 64
    const string QUEST ="Who are you?\n" ;
    int main()
    {
// инициализация
    char buff[1024];
    if (WSAStartup(0x0202, (WSADATA *)&buff[0]))
    {
    cout<<"Error WSAStartup \n"<<WSAGetLastError();
    // Ошибка!
    return -1;
    }
        SOCKET s, s_new;
```

```
        int from_len;
        char buf[BUF_SIZE] = { 0 };
        sockaddr_in sin, from_sin;
// создание слушающего сокета сервера
        s = socket(AF_INET, SOCK_STREAM, 0);
// задание параметров сокета сервера
        sin.sin_family = AF_INET;
        sin.sin_addr.s_addr = 0;
        sin.sin_port = htons(SRV_PORT);
        bind(s, (sockaddr *)&sin, sizeof(sin));
        string msg, msg1;
// создание очереди сокетов
        listen(s, 10);
    while (1)       // работа сервера с клиентами
    {
    from_len = sizeof(from_sin);
//выделение из очереди сокета очередного клиента
    s_new=accept(s,(sockaddr*)&from_sin,&from_len);
        cout << "new connected client! "<<endl;
        msg=QUEST;
        while (1)   //общение с выделенным клиентом
    {
//отправка сообщения
    send(s_new, (char *)&msg[0], msg.size(), 0);
//прием сообщения
    from_len =recv(s_new, (char *)buf, BUF_SIZE,0);
    buf[from_len]=0;
    msg1=(string)buf;
    cout << msg1 << endl;;
        if (msg1=="Bye") break;
        getline(cin, msg);
    }
// разрыв соединения с клиентом
    cout << "client is lost";
    closesocket(s_new);
    }
```

```
        return 0;
    }
```

Заметим, что данный сервер может общаться и более чем с одним клиентом (в нашем случае не более 10). Информация о сокетах присоединенных клиентов будет сохранена в очереди сервера. Представленный пример демонстрирует *последовательное* обслуживание клиентов сервером: пока сервер не разорвет соединение с очередным клиентом, он не может взаимодействовать с остальными.

Дейтаграммные UDP-сокеты. Клиент-серверное взаимодействие без установления соединения демонстрирует приложение эхо-обмена, когда сервер обратно передает клиенту полученное от него сообщение.

Код UDP-клиента:

```
    #define _WINSOCK_DEPRECATED_NO_WARNINGS
    #include <winsock2.h>
    #include <string>
    #include <windows.h>
    #include <iostream>
    #pragma comment(lib, "ws2_32.lib")
    #pragma warning(disable: 4996)
    using namespace std;
//  задание адреса и порта сервера
    #define PORT 999    // порт
    #define SERVERADDR "127.0.0.1" // IP-адрес

    int main()
    {
    char buff[10*1014];
// память для информации интерфейса Winsock
    cout<< "Use quit to exit \n";
// инициализация библиотеки Winsocks
```

```
        if (WSAStartup(0x202,(WSADATA *)&buff))
    {
cout<<"WSAStartup error:"<<WSAGetLastError()<<"\n";
        return -1;      } // ошибка инициализации
// создание дейтаграммного сокета в домене IPv4
    SOCKET my_sock=socket(AF_INET, SOCK_DGRAM, 0);
        if (my_sock==INVALID_SOCKET)
        {                       // ошибка
cout<<"SOCKET() ERROR: "<<WSAGetLastError()<< "\n";
        WSACleanup();
        return -1;
        }
//заполнение адресной структуры параметрами сервера
        sockaddr_in Daddr;
// адресная структура для сервера
        Daddr.sin_family=AF_INET;
// указание домена (IPv4)
        Daddr.sin_port=htons(PORT);
// порт в сетевом формате
// определение сетевого формата IP-адреса сервера
        hostent *hst;
        if (inet_addr(SERVERADDR))
// адрес в числовом формате?
    Daddr.sin_addr.s_addr=inet_addr(SERVERADDR);
        else
        if (hst=gethostbyname(SERVERADDR))
// получение IP-адреса по символьному имени по DNS
        Daddr.sin_addr.s_addr=
        ((unsigned long **)hst->h_addr_list)[0][0];
        else
// адрес получить не удалось
{cout<<"Unknown Host: "<< WSAGetLastError()<< "\n";
        closesocket(my_sock);
        WSACleanup();
        return -1;        }
// обмен сообщениями с сервером
```

```
    while(1)
    {
// чтение сообщения с клавиатуры
        cout<<"S<=C:";
        string SS;
        getline(cin,SS);
    if (SS == "quit") break;
// окончание взаимодействия
// передача сообщений на сервер
        sendto(my_sock,(char*)&SS[0],SS.size(),0,
        (sockaddr*)&Daddr,sizeof(Daddr));
        sockaddr_in SRaddr; // адресная структура
        int SRaddr_size=sizeof(SRaddr);
// прием сообщения с сервера
        int n=recvfrom(my_sock,buff,sizeof(buff),0,
        (sockaddr *) &SRaddr, &SRaddr_size);
        if (n==SOCKET_ERROR)  {
cout<<"recvfrom() error:"<<WSAGetLastError()<<"\n";
        closesocket(my_sock);
        WSACleanup();  return -1;  }
// вывод принятого с сервера сообщения на экран
        buff[n]='\0';
        cout << "S=>C:"<< buff<< "\n";
    }
// выход
        closesocket(my_sock);
        WSACleanup();
        return 0;
}
```

Код UDP-эхо сервера:

```
    #define _WINSOCK_DEPRECATED_NO_WARNINGS
    #include <winsock2.h>
    #include <windows.h>
    #include <string>
    #include <iostream>
```

```
    #pragma comment(lib, "ws2_32.lib")
    #pragma warning(disable: 4996)
    using namespace std;
// данные сервера
    #define PORT 999    // порт сервера
    #define sHELLO "Hello, student!\n"
    int main()     {
    char buff[1024];
// инициализация Winsock
    if (WSAStartup(0x202,(WSADATA *) &buff[0]))
        {    // ошибка
cout<< "WSAStartup error: "<< WSAGetLastError();
    return -1;     }
// создание сокета сервера
    SOCKET Lsock;
    Lsock=socket(AF_INET,SOCK_DGRAM,0);
    if (Lsock==INVALID_SOCKET)
    {
    cout<<"socket() error: "<< WSAGetLastError();
    WSACleanup();
    return -1;
    }
// задание параметров сокета в адресной структуре
    sockaddr_in Laddr;
    Laddr.sin_family=AF_INET; // адресная структура
    Laddr.sin_addr.s_addr=INADDR_ANY;
    // или 0 (любой IP адрес)
    Laddr.sin_port=htons(PORT);
// связывание сокета с локальным адресом
if (bind(Lsock,(sockaddr *) &Laddr,sizeof(Laddr)))
    { cout<< "BIND ERROR:"<< WSAGetLastError ();
    closesocket(Lsock);
    WSACleanup();
    return -1;
    }
```

```
// обработка пакетов, присланных клиентами
    while(1)
    {
/*структура для параметров сокета клиента, прислав-
шего дейтаграмму серверу */
    sockaddr_in Caddr;
    int Caddr_size = sizeof(Caddr);
// прием данных от клиента
int bsize=recvfrom(Lsock,&buff[0],sizeof(buff)-1,0,
    (sockaddr *) &Caddr, &Caddr_size);
    if (bsize==SOCKET_ERROR) //ошибка передачи
cout<< "recvfrom() error:"<< WSAGetLastError ();
/* определение параметров клиента: IP-адреса, сим-
вольного имени, порта  */
    hostent *hst;
    hst=gethostbyaddr((char*)&Caddr.sin_addr,
    4,AF_INET);
    cout<< "NEW DATAGRAM!\n"<<
    ((hst)?hst->h_name:"Unknown host")<<"/n"<<
    inet_ntoa(Caddr.sin_addr)<<"/n"<<
    ntohs(Caddr.sin_port)<< '\n';
//вывод принятого сообщения в консоли
buff[bsize]='\0'; // добавление завершающего нуля
cout << "C=>S:" << buff<<'\n' ; // Вывод на экран
// посылка дейтаграммы клиенту
sendto(Lsock,&buff[0],bsize,0,(sockaddr*)&Caddr,
sizeof(Caddr));
    }
    return 0;
    }
```

Заметим, что дейтаграммные клиент и сервер используют для сетевых взаимодействий одни и те же Winsock-функции. Разница лишь в том, что клиент является инициатором обращения к серверу, а сервер, как правило, пассивно ожидает обращений от клиентов. Отсутствие логических соединений сервера с клиентами

позволяет серверу обмениваться информацией с клиентами в произвольном порядке. Такой режим взаимодействия создает иллюзию параллельной работы сервера с клиентами. Дейтаграммное взаимодействие более оперативно, нежели взаимодействие с установлением соединения. Однако, как уже было сказано, оно не дает полной гарантии доставки данных адресату и может не сохранить порядок передачи и приема данных.

Представленное в данном примере модельное дейтаграммное сетевое приложение может использоваться и для взаимодействия с сервером большого числа клиентов, функционирующих на разных узлах сети TCP/IP.

5. РЕЖИМЫ СОКЕТОВ

Windows сокеты могут выполнять операции ввода-вывода в двух режимах: блокирующем и неблокирующем. В блокирующем режиме вызовы Winsock функций, которые выполняют операции ввода-вывода, такие как send и recv – ждут, пока операция завершится, прежде чем отдать управление приложению. В неблокирующем режиме Winsock функции отдают управление приложению сразу.

Блокирующие сокеты создают неудобства, потому что вызов определенных Winsock API функций блокируют выполнение на некоторое время. Большинство Winsock приложений следуют модели «*производитель – потребитель*», в которой приложение считывает либо записывает определенное количество байт и выполняет их обработку.

Следующий отрывок кода иллюстрирует эту модель:

Код:

```
...
SOCKET sock;
char    buffer[256];
int     err;

while(true)
{
// прием данных
    err = recv(sock, buffer, sizeof (buffer));

if (err == SOCKET_ERROR)     // ошибка приема
{cout<<"recv failed with error" <<WSAGetLastError()
<<endl;
return -1; }

// обработка данных
}
...
```

Проблема в приведенном коде в том, что функция **recv** может никогда не отдать управление приложению, если на данный сокет не придут какие-то данные.

Эта особенность не очень хорошо подходит для программ, которые кроме получения / отправки данных должны выполнять еще множество других действий (отслеживание состояния системы меню, вывод информации, опрос других устройств ввода / вывода). Избежать этого можно многими способами. Можно обойтись средствами мультизадачности, и процедуру обмена данными «повесить» на отдельную ветвь. А можно решить проблему средствами Winsock. В каждом конкретном случае выбор метода остаётся за программистом. Исследуем механизм блокировки TCP-сокетов, имеющийся в Winsock. Рассмотрим присущие ему два метода устранения проблемы блокировки:

5.1. Функция ioctlsocket

Функция ioctlsocket позволяет менять / получать режим ввода / вывода конкретного сокета.

Прототип функции:

```
int ioctlsocket (SOCKET s,       // Сокет [in]
long cmd,                        // Команда [in]
u_long FAR *argp);  //Параметр/значение [in/out]
```

Для перевода сокета в неблокируемое состояние (nonblocking mode) используется команда FIONBIO. Параметр argp должен содержать ненулевое значение.

Пример фрагмента кода:

```
BOOL lu = TRUE;
If (SOCKET_ERROR ==
ioctlsocket(s, FIONBIO, (unsigned long*)&lu))
{           // Error
    int res = WSAGetLastError ();
    return -1;}
```

Кроме команды FIONBIO существуют команды FIONREAD и SIOCATMARK. Если коротко, то FIONREAD позволяет получить количество байт информации, поступившей в буфер на данный момент операции чтения, а SIOCATMARK – используется при работе с OOB данными.

В предыдущем фрагменте кода используется команда FIONBIO. После вызова ioctlsocket сокет s стал неблокируемым, т. е. Winsock-функции для этого сокета не дожидаются окончания операций ввода / вывода, что в свою очередь не вызывает нежелательных пауз в работе программы. Однако не всё так просто. Например, возврат из функции recv может произойти до момента получения данных. Как определить, что текущая операция ввода / вывода полностью завершена? По коду ошибки **WSAEWOULDBLOCK**, который возвращают Winsock-функции для nonblocked сокета, если текущая операция не завершена. То есть если функция recv вернула этот код ошибки, значит, данные еще не готовы для чтения, и операцию придется повторить позже. В таком случае ваша программа может выполнять другие действия, попутно проверяя, не завершена ли текущая операция ввода / вывода.

Описанный механизм деблокирования можно использовать, например, в программе-клиента, которая обращается к Web-серверу с запросом на скачивание определенного файла. Так как размер запрашиваемого файла может оказаться большим, можно не прерывать работу программы на все время скачивания, а продолжать выполнять некоторую другую работу, периодически проверяя окончание передачи файла. В качестве примера приведем код программы Web-клиента, где используется функция ioctlsocket:

```
    #include "winsock2.h"
char * request  ="GET /ru-ru.html HTTP/1.1\r\nHost:
www.anotherd.com\r\n\r\n"           //HTML запрос.
    int  MAX_PACKET_SIZE=   65535
    int main()
    {
```

```
        WSADATA     ws;
        SOCKET      s;
        sockaddr_in adr;
        hostent*hn;
        char    buff [MAX_PACKET_SIZE];
// инициализация
        if (WSAStartup (0x0101, &ws) != 0)
        {       // Error
            return -1;  }
// создание сокета
        if (INVALID_SOCKET == (s = socket (AF_INET,
SOCK_STREAM, 0)))
        { return -1;  } // Error
/* Заполняем  поля структуры adr перед соединением
с сервером */
// Определяем адрес
if(NULL==(hn = gethostbyname ("www.anotherd.com")))
        {   return -1; }// Error
    adr.sin_family  = AF_INET; // домен для IPv4
adr.sin_addr.s_addr=*(DWORD* ) hn->h_addr_list[0];
    adr.sin_port= htons (80);

// Устанавливаем соединение с сервером
    if(SOCKET_ERROR==connect(s,(sockaddr*)&adr,
        sizeof (adr) ))
    {int res = WSAGetLastError ();
    return -1; }   // Error

// Посылаем запрос серверу
    if(SOCKET_ERROR==
        send(s,&request,sizeof(request),0))
    {       int res = WSAGetLastError ();
        return -1; }            // Error
// Устанавливаем nonblocked mode
        BOOL l = TRUE;
    if (SOCKET_ERROR == ioctlsocket(s,FIONBIO,
```

```
        (unsigned long*)&l))
    {int res = WSAGetLastError ();
        return -1; }   // Error

// Получаем данные...
    int len = 0;
    do
    {
    if(SOCKET_ERROR==(len=recv(s,(char*)&buff,
MAX_PACKET_SIZE, 0)))
            {
        int res = WSAGetLastError ();
        if (res!=WSAEWOULDBLOCK) return -1;
            else len = 1;
            }
        else
        for (int i = 0; i<len; i++)
            cout<<buff [i];
/* Если recv вернул ошибку WSAEWOULDBLOCK, то можем
продолжать работу.Делаем все необходимые нам дей-
ствия.и опять переходим к вызову recv (...) */
    } while (len!=0);
// закрываем соединение
        if (SOCKET_ERROR == closesocket (s))
        { return -1; }          // Error

        return 1;
}
```

5.2. Функция `select`

Как правило, функция *`select`* используется в программах-серверах, однако ее также можно встретить и в программах-клиентах со сложных функционалом. С помощью данной функции можно проверять конечное множество сокетов на готовность к

считыванию / отсылке данных, выполнения `connect`, на предмет входящих соединений, наличия OOB сообщений и т. п.

Функция `select` позволяет установить для сокета следующий статус:

Сокет для чтения (readable socket) содержит принятые данные, которые извлекаются программой при помощи стандартных вызовов recv или recvfrom.

Сокет для записи (writable socket) – это сокет, установивший соединение. Через него программа может передавать данные, используя стандартные функции типа send или sendto.

Ошибочный сокет (exception socket) – это сокет, в котором произошла сетевая ошибка. Ситуация ошибки требует дальнейшей программной обработки.

В качестве примера рассмотрим ситуацию, когда в сетевой программе задействованы три сокета и есть одна процедура, обслуживающая операции чтения, другая процедура, осуществляющая запись, и третья, обрабатывающая ошибочные ситуации. С помощью функции `select` можно одновременно получить информацию о состоянии всех трех сокетов. Предположим, что в результате вызова `select` определяется, что один сокет может считывать данные, второй может записывать, а третий содержит информацию об ошибке. В этом случае разумнее всего будет вызвать процедуру обработки ошибок. Получив статус всех контролируемых сокетов, можно выбрать то действие, которое наиболее разумно предпринять.

Для определения множества сокетов, за которыми будет установлено наблюдение, интерфейс сокетов использует битовые маски. Функция `select` определяет состояние только тех сокетов, которые были предварительно отмечены в битовой маске множества одной из трех описанных категорий. При выходе в вызывающую программу `select` возвращает количество сокетов, готовых к операциям ввода-вывода. Дополнительно `select` модифицирует битовые маски множеств таким образом, что каждый сокет оказывается принадлежащим множеству определенной категории. До того, как вызвать функцию `select`, программа должна поместить сокет во множестве определенной категории, т. е. установить биты в маске так, чтобы они указывали на сокеты, информацию о

которых необходимо получить. Функция *select* сбросит биты, указывающие на любой сокет, не готовый к определенной операции ввода-вывода. После завершения функции *select* прикладная программа может проанализировать содержимое битовой маски. Если бит, идентифицирующий определенный сокет, окажется установленным, это значит, что сокет готов к операции ввода-вывода (чтению, записи или сообщению об ошибке).

Прототип функции `select`*:*

```
    int select (
    int nfds,
/* общее количество сокетов для наблюдения, может
быть опущен */
    fd_set FAR *readfds,
/* множество сокетов, проверяемых на готовность к
чтению */
    fd_set FAR *writefds,
/* множество сокетов, проверяемых на готовность к
отсылке */
    fd_set FAR *exceptfds,
/* множество сокетов, проверяемых на ошибку/OOB
данные */
    const struct timeval FAR *timeout
// Таймаут проверки
            );
```

Каждый из параметров `readfds, writefds, exceptfds, timeout` является необязательным и может быть проигнорирован (установлен в `NULL`).

В случае `readfds / writefds / exceptfds == NULL` проверка на опущенные типы сокетов просто не будет производиться. В случае `timeout == NULL` функция `select` вызовет блокировку (до первого готового к вводу / выводу сокета). Функция возвращает общее количество сокетов (во всех заданных

множествах readfds / writefds / exceptfds), готовых к операциям ввода / вывода.

Параметры readfds / writefds / exceptfds – указатели на тип fd_set (битовая маска, представляющая собой множество сокетов). Для работы с этим типом данных объявлены такие макросы:

FD_CLR (s, *set) – удаляет дескриптор s из set.
FD_ISSET(s, *set) – возвращает ненулевое значение, если s присутствует в set. Иначе, возвращает ноль.
FD_SET(s, *set) – добавляет s к set.
FD_ZERO(*set) – очищает множество set.

Параметр timeout – указатель на структуру timeval, позволяет задать таймаут, в течение которого сокеты будут проверяться на готовность:

```
struct timeval {
    long tv_sec;// секунды
    long tv_usec;   // микросекунды
};
```

В качестве примера приведем исходный код программы Web-клиента, проверяющей готовность сокета к чтению. Для этого поместим данный сокет во множество, на которое будет указывать readfds (в примере это read_s), зададим timeout и воспользуемся функцией select. Web-сервер располагается по адресу: www.hugedomains.com

```
#include "winsock2.h"
#define request "GET/ HTTP/1.1\r\n
Host: www.hugedomains.com \r\n\r\n" //HTML запрос.
#define MAX_PACKET_SIZE 65535
int main()
{
    WSADATA ws;
```

```
        SOCKET  s;
        sockaddr_in adr;
        hostent*hn;
        charbuff [MAX_PACKET_SIZE];
// Init
        if (WSAStartup (0x0101, &ws) != 0)
        {    return -1; }    // Error
// Создаём сокет
    if(INVALID_SOCKET ==
(s=socket(AF_INET,SOCK_STREAM,0)))
        {   return -1; }  // Error
// Получаем адрес
    if (NULL ==
(hn=gethostbyname("www.hugedomains.com")))
        {   return -1; }            // Error
/* Заполняем поля структуры adr для использования
ее в connect */
        adr.sin_family= AF_INET;
adr.sin_addr.s_addr=*(DWORD* )hn->h_addr_list[0];
        adr.sin_port= htons (80);
// Устанавливаем соединение с сервером
    if(SOCKET_ERROR ==
    connect(s,(sockaddr*)&adr,sizeof(adr)))
{ int res=WSAGetLastError (); return -1;} // Error
// Посылаем запрос серверу
    if (SOCKET_ERROR ==
    send (s, &request, sizeof (request), 0) )
    { int res=WSAGetLastError();
    return -1; } // Error
// Ждём ответа
    int len, res;
fd_set  read_s;     // Множество readable sockets
    timeval time_out;       // Таймаут
    time_out.tv_sec = 0;
    time_out.tv_usec = 500000; // Таймаут 0.5 sec.
```

```
    do
    {
FD_ZERO (&read_s);       // Обнуляем множество
FD_SET (s, &read_s);     // Заносим в него наш сокет
    if (SOCKET_ERROR ==
    (res = select(0,&read_s,NULL,NULL, &time_out)))
    return -1;           //ошибка
// здесь для примера используется FD_ISSET!
    if ((res!=0) && (FD_ISSET (s, &read_s)) )
    {       // Данные готовы к чтению...
if (SOCKET_ERROR == (len = recv(s,(char
*)&buff,MAX_PACKET_SIZE,0)))
    { int res = WSAGetLastError (); return -1; }
// Выводим их на экран.
        buff[len]=0;
        cout<< buff;
    }
// Делаем все необходимые нам действия...
    }   while (len!=0);

// закрываем соединение
        if (SOCKET_ERROR == closesocket (s))
        { return -1; }              // Error

    return 1;
    }
```

6. СОВРЕМЕННЫЕ НАДСТРОЙКИ WINSOCK

Расширение применения интерфейса сокетов на сетевые домены, использующие разные сетевые протоколы и, как следствие, разные адресные форматы, потребовало модернизации библиотеки интерфейсных функций для обеспечения единого стиля при сетевом программировании. Особенно остро этот вопрос встал в то время, когда на смену сетевому протоколу IPv4 пришел протокол IPv6. Отметим, что для использования при программировании новых интерфейсных функций необходимо именно в таком порядке подключить два заголовочных файла:

```
#include <WinSock2.h>
#include <WS2tcpip.h>
```

Далее укажем наиболее важные изменения интерфейса Winsock.

6.1. Функции getaddrinfo() и freeaddrinfo()

В первую очередь нужно отметить появление новой функции getaddrinfo(), позволившей оптимизировать процесс заполнения структуры сокета адресной информацией. Напомним, что ранее для DNS-запросов необходимо было использовать функцию под названием gethostbyname(). Затем информацию нужно было вручную заносить в struct sockaddr_in и только затем использовать её. Теперь в этом нет необходимости (это даже нежелательно, если вы хотите использовать не только IPv4, но и IPv6).

Функция getaddrinfo() при необходимости выполнит обращение к DNS, осуществит перевод IP-адресов и номеров портов в сетевой формат и заполнит адресные структуры полученными данными.

Прототип функции getaddrinfo:

```
int getaddrinfo (
```

```
    const char *node,
// например, "www.example.com" или IP-адрес
    const char *service,
// например, "http" или номер порта
    const struct addrinfo *hints,
    struct addrinfo **res
                        );
```

Определение внутренней структуры `addrinfo`:

```
    struct addrinfo {
        int     ai_flags;
        int     ai_family;
        int     ai_socktype;
        int     ai_protocol;
        size_t  ai_addrlen;
        struct sockaddr*    ai_addr;
        char*               ai_canonname;
        struct addrinfo*    ai_next;
                    };
```

Функция `getaddrinfo`() принимает три входных параметра и выдаёт указатель `res` на связанный список результатов. Параметр `hints` – указатель на структуру `addrinfo`, в которую предварительно необходимо внести некоторую информацию для указания на предпочтительный тип сокета или протокол. Так, нулевое значение `hints` указывает, что для работы возможен любой сетевой адрес или протокол. Если этот параметр не равен `NULL`, то он является указателем на структуру `addrinfo`, значения полей `ai_family`, `ai_socktype` и `ai_protocol` которой определяют предпочтительный тип сокета. Так значение `AF_UNSPEC` (или `PF_UNSPEC`) в `ai_family` указывает на произвольное семейство протоколов (например, `IPv4` либо `IPv6`). Нулевое значение в `ai_socktype` или `ai_protocol` указывает, что для работы возможен любой тип сокета и любой протокол соответственно. Поле `ai_flags` указывает на дополнительные опции. Несколько флагов

указываются путем их логического сложения (or). Все другие поля структуры, на которую указывает hints, должны содержать либо 0, либо указатель NULL.

Функция getaddrinfo() создает связанный список структур addrinfo, по одной на каждый сетевой адрес, с учетом ограничений, наложенных параметром hints. Если в поле ai_flags структуры hints выставлен флаг AI_CANONNAME, то в структуре результата в поле ai_canonname устанавливается указатель официального имени машины. Параметры ai_family, ai_socktype и ai_protocol указывают на параметры создания сокета. Указатель на структуру сокета помещается в поле ai_addr, а размер структуры сокета (в байтах) – в поле ai_addrlen.

Нулевое значение может принимать либо параметр node, либо service, но не оба одновременно. Ненулевое значение node указывает на сетевой адрес или в числовом виде (десятично-точечный формат для IPv4, шестнадцатеричный для IPv6), или в виде символьного имени машины, поиск и разрешение адреса которой будут затем произведены. Если поле ai_flag в структуре с указателем hints содержит флаг AI_NUMERICHOST, то параметр node должен быть числовым сетевым адресом.

Если node равно NULL, то сетевой адрес в каждой сокетной структуре инициализируется в соответствии с флагом AI_PASSIVE, устанавливаемым в поле ai_flags параметра hints. Если установлен флаг AI_PASSIVE, то сетевой адрес каждой структуры не будет указан. Это используется серверными приложениями, предназначенными для приема соединений клиентов, имеющих любой сетевой адрес. Если флаг AI_PASSIVE сброшен, то сетевой адрес будет вписан в адрес интерфейса обратной петли. Это используется клиентскими приложениями, желающими установить соединение с сервером, запущенным на той же машине.

Параметр service вписывает номер порта в сетевой адрес каждой сокетной структуры. Если service равно NULL, то номер порта останется неинициализированным.

При нормальном завершении работы функция getaddrinfo возвращает 0, в противном случае можно при помощи

`gai_strerror(status)` получить описание кода ошибки `status` этой функции.

Пример вызова данной функции в случае сервера, слушающего сеть на порту `3490`:

```
    int status;
    struct addrinfo hints;
    struct addrinfo *servinfo;
// указатель на результаты

    ZeroMemory(&hints, sizeof(hints));
// очищение исходной структуры
//предварительно заполняем структуру hints
hints.ai_family = AF_UNSPEC;    // любая версия IP
    hints.ai_socktype = SOCK_STREAM;
// TCP STREAM-SOCKETS
    hints.ai_flags = AI_PASSIVE;
// заполнение IP-адреса системой

if ((status = getaddrinfo(NULL, "3490", &hints,
&servinfo)) != 0)
{ cerr << "getaddrinfo failed: " << status << "\n";
WSACleanup();  // выгрузка библиотеки Ws2_32.dll
    return 1; }

/* servinfo теперь указывает на связанный список на
одну или больше структуру addrinfo */
//Делаем что-то, где используем структуру addrinfo
....
// и затем освобождаем память под связанный список
    freeaddrinfo(servinfo);
```

Указанное в примере значение `AF_UNSPEC` поля `ai_family` позволяет использовать различные типы адресов. Можно также

задать в этом поле значение AF_INET/PF_INET для домена IPv4 или AF_INET6/ AF_INET6 для IPv6.

Используемое в примере значение флага AI_PASSIVE дает возможность функции getaddrinfo() автоматически назначать сокету адрес данного хоста. Это позволяет не указывать адрес данного компьютера вручную. В противном случае можно указать свой IP в качестве первого параметра getaddrinfo() (в данном примере при указанном флаге AI_PASSIVE первый параметр равен NULL).

Если есть ошибки (getaddrinfo() вернула не ноль), можно напечатать код ошибки. Если всё работает правильно, servinfo будет указывать на связанный список структур addrinfo.

После завершения всех необходимых действий со связанными списками, созданными getaddrinfo(), динамически выделенная под список память освобождается вызовом freeaddrinfo().

В следующем примере создаются необходимые адресные структуры для подключения клиента к хосту "www.example.net" и порту 3490.

```
    int status;
    struct addrinfo hints;
    struct addrinfo *servinfo;

/* предварительное заполнение адресной структуры
hints */
    ZeroMemory(&hints, sizeof(hints);
    hints.ai_family = AF_UNSPEC;  //  IPv4 или IPv6
    hints.ai_socktype = SOCK_STREAM;

// готовимся к соединению
    status = getaddrinfo ("www.example.net",
"3490", &hints, &servinfo);
```

```
/* теперь servinfo – указатель на список структур
addrinfo */
//
```

6.2. Функции inet_pton и inet_ntop

Познакомимся с новыми функциями `inet_pton` и `inet_ntop`, предназначенными для преобразования адресов Интернета из числовых строк `ASCII` (удобных для человеческого восприятия) в двоичные значения с сетевым порядком байтов (эти значения хранятся в структурах адресов сокетов). Эти функции появились с `IPv6` и работают как с адресами `IPv4`, так и с адресами `IPv6`. Символы `p` и `n` указывают соответственно на формат текстового представления и на сетевой двоичный формат адреса.

Первая функция:

```
int inet_pton(int ai_family, const char *strptr,
void *addrptr);
```

пытается преобразовать текстовую строку, на которую указывает `strptr`, и сохранить двоичный результат в поле с указателем `addrptr`.

При успешном выполнении ее возвращаемое значение равно `1`. Если входная строка находится в неверном формате представления для заданного семейства (`ai_family`), возвращается нуль. В случае ошибки возвращается `-1`.

Вторая функция:

```
const char *inet_ntop
(int ai_family, const void *addrptr,
char *strptr, size_t len);
```

выполняет обратное преобразование: из сетевого формата (`addrptr`) в формат числового представления (`strptr`).

Аргумент `strptr` функции `inet_ntop` не может быть пустым указателем. Вызывающий процесс должен выделить память для хранения преобразованного значения и задать ее размер. При успешном выполнении функции возвращаемым значением является этот указатель.

Аргумент len – это размер принимающей строки, который передается, чтобы функция не переполнила буфер вызывающего процесса. Если аргумент len слишком мал для хранения результирующего формата представления вместе с символом конца строки (terminating null), возвращается пустой указатель и переменной errno присваивается значение ENOSPC.

Для задания размера можно использовать также и системные константы:

```
/* для точечно-десятичной записи IPv4-адреса */
#define INET_ADDRSTRLEN 16

/* для шестнадцатеричной записи IPv6-адреса */
#define INET6_ADDRSTRLEN 46
```

Функция inet_ntop() возвращает: указатель на результат, если выполнение функции прошло успешно, NULL в случае ошибки.

Значением аргумента ai_family для обеих функций может быть либо AF_INET (для IPv4), либо AF_INET6 (для IPv6). Если ai_family не поддерживается, обе функции возвращают ошибку со значением переменной errno, равным EAFNOSUPPORT.

В многочисленных существующих сетевых программах для преобразования адреса IPv4 из точечно-десятичной записи в 32-разрядное двоичное значение в сетевом порядке байтов используются функции inet_ntoa и inet_addr, входящие в интерфейс Winsock классической редакции. Очевидно, что возможности новых функций для конвертации адресных форматов несколько шире, так как они способны поддерживать несколько типов адресов. Поэтому использование inet_ntoa и inet_addr и для случая IPv4 в некоторых средах разработки считается устаревшим и ненужным, а иногда даже блокируется.

Приведем пример программы, которая на основе информации адресных структур, заполненных для этого хоста функцией

getaddrinfo(), определяет IP-адреса для указанного хоста, используя функцию inet_ntop():

```
    int main() {
    struct addrinfo hints, *res, *p;
    int status;
    char ipstr[16];
    char hostname[]="ipv6.example.com"
//символьное имя хоста
    ZeroMemory(&hints, sizeof(hints);
hints.ai_family=AF_UNSPEC; // AF_INET или AF_INET6
    hints.ai_socktype=SOCK_STREAM;
// получение адресной информации
if((status=getaddrinfo(hostname,NULL,&hints,&res))
        != 0)    {
cout<<"error getaddrinfo:"<<gai_strerror(status));
    return 2;    }
cout<<"IP addresses for"<<hostname << ":"<< endl;
// проход по списку полученных адресных структур
    for(p = res;p != NULL; p = p->ai_next) {
        void *addr;
        char *ipver;
/* получаем указатель на адрес,
    по-разному в разных протоколах */
if (p->ai_family == AF_INET) //адрес в формате IPv4
    {
    struct sockaddr_in *ipv4 =
            (struct sockaddr_in *)p->ai_addr;
        addr = &(ipv4->sin_addr);
        ipver = "IPv4";
    }
    else // адрес в формате IPv6
    {
    struct sockaddr_in6 *ipv6 =
    (struct sockaddr_in6 *)p->ai_addr;
        addr = &(ipv6->sin6_addr);
```

```
        ipver = "IPv6";
    }
// преобразуем адрес IP в строку и выводим его:
inet_ntop(p->ai_family, addr, ipstr, sizeof ipstr);
cout<< ipver<< ":"<< ipstr<<endl; }
freeaddrinfo(res); // освобождаем связанный список
    return 0; }
```

В результате выполнения указанного кода получим:

```
IP addresses for ipv6.example.com:
  IPv4: 192.0.2.101
  IPv6: 2001:db8:8c00:22::171
```

6.3. Новая редакция базовых функций Winsock

Рассмотрим также базовые функции интерфейса сокетов с указанием изменений, произведенных в новейшей редакции с целью обобщить их работу на новые типы адресов и таким образом расширить область применения. Предложенные примеры также продемонстрируют использование данных, полученных функцией `getaddrinfo()` при вызове интерфейсных функций.

Создание дескриптора сокета

Расширены области определения аргументов функции `socket()`:

```
int socket(int domain, int type, int protocol);
```

`domain` – это `PF_INET` или `PF_INET6` (`PF` – protocol family). `PF_INET` по значению равен `AF_INET` (`AF` – address field), который используется при инициализации поля `sin_family` в структуре `sockaddr_in` при использовании протокола `IPv4`.

`type` – это `SOCK_STREAM` или `SOCK_DGRAM`.

protocol может быть установлен в 0 для автоматического выбора подходящего этому семейству протокола. Для получения номера нужного протокола, TCP или UDP, можно также использовать getprotobyname().

Пример использования:

```
    int s;
    struct addrinfo hints, *res;
    …
//  заполняем структуру hints
    …
getaddrinfo("www.example.com","http",&hints,&res);
/* проверить на ошибки результат работы
getaddrinfo() и использовать полученный связанный
список */
    …
    s=socket(res->ai_family,res->ai_socktype,
        res->ai_protocol);
    …
```

Связывание сокета с адресной информацией

Прототип функции *bind:*

```
    int bind(int sockfd, struct sockaddr *my_addr,
int addrlen);
```

sockfd – это файловый дескриптор сокета, возвращаемый вызовом socket(), my_addr – указатель на struct sockaddr, содержащую адресную информацию (IP-адрес и порт), addrlen – это длина адреса в байтах.

Пример использования *bind* совместно с getaddrinfo*:*

```
    struct addrinfo hints, *res;
    int sockfd;
    ZeroMemory(&hints, sizeof(hints);
    hints.ai_family = AF_UNSPEC;  // IPv4 или IPv6
```

```
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
//автоматическое получение IP
    getaddrinfo(NULL, "3490", &hints, &res);
// получение адресной информации
// и использование ее при создании сокета
sockfd = socket(res->ai_family, res->ai_socktype,
res->ai_protocol);

    bind(sockfd, res->ai_addr, res->ai_addrlen);
/* связь сокета с адресными данными, полученными
getaddrinfo() */:
    …
```

Использование флага `AI_PASSIVE` позволяет автоматически поместить в структуру сокета IP-адрес хоста, на котором запущена программа. Если необходимо в сокете указать конкретный адрес, не следует задавать значение флага, нужно просто передать первым аргументом `getaddrinfo()` конкретный IP-адрес. Функция `bind()` возвращает `0` при успешном завершении и `-1` при ошибке .

При вызове `bind()` желательно не выходить за рамки, назначая номер порта. Все порты ниже `1024` – зарезервированы и использовать их желательно только суперпользователю (`root`)! Можно использовать любой порт выше этого значения, вплоть до `65535` (конечно, если порт не используется другой программой). Так, иногда при установлении связи с сервером можно получить ошибку «*адрес уже используется» / «address allready in use*». Это означает, что сокеты, установившие соединение, всё ещё висят в ядре ОС, и оно держит порт занятым.

Напомним, что иногда не нужно вызывать `bind()`. Так, при подсоединении клиента к удалённой машине неважно, каким будет локальный порт клиента. В этом случае на стороне клиента нужно только вызвать функцию `connect()`, которая сама найдёт незанятый порт и его номер укажет в адресной информации клиента, т. е. автоматически выполнит и операции функции `bind()`.

Клиентские функции

Связь с удаленным узлом. Прототип функции connect():

```
int connect(int sockfd, struct sockaddr *serv_addr,
int addrlen);
```

sockfd – дескриптор сокета, полученный функцией socket(), serv_addr – структура sockaddr, содержащая порт назначения и IP-адрес; addrlen – длина этой структуры в байтах. Вводную информацию для этой функции можно получить путем вызова getaddrinfo().

Рассмотрим пример, в котором создаётся подключение клиента к серверу с именем www.example.com через порт 3490:

```
    struct addrinfo hints, *res;
    int sockfd;
/* сначала заполним адресные структуры с помощью
getaddrinfo(): */
    ZeroMemory(&hints, sizeof(hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

// получение адресной информации
getaddrinfo("www.example.com", "3490", &hints,
&res);

/* использование адресной информации при создании
сокета */
    s_sock = socket(res->ai_family, res->ai_sock-
type, res->ai_protocol);

// соединение с сервером
    connect(s_sock, res->ai_addr, res->ai_addrlen);
```

Обратим внимание, что в примере нет вызова `bind()`. В данном случае клиенту нужен только конкретный адрес и порт удалённого узла-сервера, с которым устанавливается соединение, и совершенного все равно, каким будет локальный номер порта клиента. Ядро само автоматически выберет номер локального порта и узел-сервер, с которым мы соединяемся, автоматически получит эту информацию. Получение адресной информации клиента, размещение ее в структуре сокета и передача серверу при установлении связи – все выполняется функцией `connect()`.

Серверные функции

Ожидание входящих сообщений. Прототип функции `listen()`:

```
int listen(int s_sock, int backlog);
```

`s_sock` – это дескриптор сокета, а `backlog` – число возможных соединений во входящей очереди.

Перед вызовом `listen()` необходимо вызвать `bind()`, чтобы сервер запустился на определённом порту (а программа-клиент должна знать этот порт, чтобы к нему обратиться).

Принятие соединения. Входящие соединения должны ожидать в очереди, пока их не примут. При успешной попытке соединения с сервером на порт, который слушает сервер, сведения о соединении попадают в очередь ожидающих принятия. Для принятия ожидающего соединения необходимо вызвать функцию `accept()`, которая возвращает совершенно новый дескриптор сокета для этого конкретного соединения. В результате у нас есть и первый сокет, который по-прежнему всё ещё слушает новые входящие соединения, и только что созданный сокет, готовый уже к обмену данными с помощью `send()` и `recv()`!

Прототип функции выглядит так:

```
int accept(int sockfd, struct sockaddr *addr,
socklen_t *addrlen);
```

sockfd – это слушающий сокет, addr – это обычно указатель на локальную структуру sockaddr_storage. Чтобы не ограничиваться только IPv4 или только IPv6, часто в сетевом программировании используют как тип адресной структуры сокета структуру sockaddr_storage, которой точно хватит для хранения обоих типов протоколов. В этой структуре будет информация о входящем соединении. С её помощью можно узнать, кто именно и с каким исходящим портом соединялся с вами. addrlen – локальная переменная, целое число, которое должно быть установлено в sizeof(sockaddr_storage) перед передачей в accept(). Функция accept() не поместит в структуру объём данных больше, чем указано в этой переменной. Если поместит меньше, то отразит это в значении addrlen.

Пример применения интерфейсных функций на стороне сервера:

```
    int main()
{
    struct sockaddr_storage D_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int s_sock, new_sock;
/* заполнение адресной структуры с помощью
    getaddrinfo() */
    ZeroMemory(&hints, sizeof(hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    getaddrinfo(NULL, myport, &hints, &res);

// создание сокета для прослушивания сети
    s_sock = socket(res->ai_family,
            res->ai_socktype,res->ai_protocol);
// связь сокета с адресом
    bind(s_sock, res->ai_addr, res->ai_addrlen);
// прослушивание
```

```
    listen(s_sock, backlog);

// принятие очередного соединения
    addr_size = sizeof (D_addr);
    new_sock = accept(s_sock,(sockaddr *)&D_addr,
            &addr_size);
// все готово для общения через сокет new_sock!
```

Обмен данными

Передача потоковых данных. Для коммуникации через потоковые сокеты используются функции `send()` и `recv()`.

Прототип функции `send()`:

```
int send(int sockfd, const void *msg, int len,
        int flags);
```

`sockfd` – дескриптор сокета, через который пересылаются данные, `msg` – указатель на данные, которые вы хотите передать; `len` – длина этих данных в байтах. Флаги можно просто установить в `0`.

Функция `send` возвращает число байтов, посланных фактически. Если значение, возвращаемое `send()`, меньше значения `len`, это сигнал, что нужно отправить оставшиеся данные. Небольшой пакет (около `1кb`) с большей вероятностью отправится за один раз. При ошибке возвращается `-1.`

Синтаксис `recv()` во многом схож:

```
int recv(int s_sock, void *buf, int len,
    int flags);
```

`s_sock` – это дескриптор сокета, из которого мы читаем данные, `buf` – буфер, в который мы читаем информацию, а `len` – максимальный размер буфера приёма. Флаги вновь могут быть установлены в `0`. Функция `recv()` возвращает число байтов, прочитанных и записанных в буфер, или `-1` при ошибке. `recv()` может вернуть и `0`. Это означает, что удалённая сторона закрыла канал связи.

Обмен дейтаграммами. При использовании сокетов, не устанавливающих соединения, используются `sendto()` и `recvfrom()`.

Синтаксис функции `sendto()` в основе своей такой же, как и у `send()`:

```
int sendto(int sockfd, char *buf, int len,
unsigned int flags, sockaddr *to,int *tolen)
```

с добавлением двух фрагментов информации: `to` – указатель на структуру `sockaddr` (которая, вероятно, будет другой структурой, `sockaddr_in`, `sockaddr_in6` или `sockaddr_storage`), которая содержит адрес назначения и порт, и `tolen` – указатель на локальное целое `int`, которое должно быть инициализировано как `sizeof(to)` или как `sizeof(struct sockaddr_storage)`. Заполнение адресной структуры пункта назначения можно выполнить вручную или воспользоваться функцией `getaddrinfo()`.

Точно так же, как и `send()`, `sendto` возвращает число фактически отправленных байтов (которое может быть меньше общего числа байт, которые вы пытались передать!), или `-1` при ошибке.

Столь же похожи и `recv()` и `recvfrom()`.

Прототип функции `recvfrom()`:

```
int recvfrom(int sockfd, void *buf, int len,
unsigned int flags, sockaddr *from,int *fromlen);
```

Так же, как и у `sendto`, добавляется пара полей: `from` – указатель на локальную структуру `sockaddr_storage`, которая будет после вызова заполнена `IP`-адресом и портом удалённой машины, `fromlen` – указатель на размер этой адресной структуры.

Функция `recvfrom()` возвращает число полученных байтов или `-1` при ошибке.

Напомним, если использовать `connect()` с дейтаграммным сокетом, то для коммуникации также можно использовать `send()` и `recv()`. Сокет сам по себе останется дейтаграммным и для передачи пакетов будет по-прежнему использоваться протокол `UDP`,

но интерфейс сокета будет автоматически добавлять информацию об адресах.

Закрытие соединения

Для закрытия соединения на сокете `sockfd` можно использовать функцию `closesocket()`. Это предотвратит дальнейшее чтение и передачу в сокет. При попытке на другом конце прочитать или передать данные через данный сокет, получим ошибку.

Для получения большего контроля над закрытием сокета можно использовать функцию `shutdown()`. Она позволяет прервать связь в выбранном направлении или в обоих направлениях (так же, как это делает `closesocket()`). Отметим, что `shutdown` на самом деле не закрывает дескриптор – она только отключает возможность его использования. Чтобы освободить дескриптор, нужно вызывать `closesocket()`.

Дополнительные полезные функции

Функция `getpeername()` позволяет получить адресную информацию об удаленном узле, соединение с которым производится через данный сокет.

Синтаксис:

```
int getpeername(int s_sock, struct sockaddr *addr,
int *addrlen);
```

`s_sock` – это дескриптор потокового сокета (после установления соединения функцией `connect()`); `addr` – указатель на адресную структуру `struct sockaddr` (или на `struct sockaddr_in`), которая и будет содержать информацию о другом конце соединения, а `addrlen` – указатель на целую переменную `int`, что содержит размер адресной структуры (`sizeof(struct sockaddr)`).

Функция возвращает `-1` при ошибке.

После получения адреса можно использовать `inet_ntop()`, `getnameinfo()` или `gethostbyaddr()`, чтобы вывести информацию в читаемом виде.

Функция `gethostname()` возвращает символьное имя компьютера, на котором запущена программа. Полученное имя затем может быть использовано, например, в `gethostbyname()` при определении адреса компьютера.

Синтаксис:

```
int gethostname(char *hostname, size_t size);
```

Аргументы: `hostname` – указатель на массив `char`, который после выполнения функции будет содержать имя хоста, и `size` – размер длины этого массива в байтах.

Функция возвращает `0` при успехе и `-1` в случае ошибки.

7. HTTP-ВЗАИМОДЕЙСТВИЯ

Hyppertext Transfer Protocol (HTTP), протокол пересылки гипертекста – это язык, которым клиенты и серверы World Wide Web (WWW) пользуются для общения между собой.

Протокол HTTP работает поверх TCP (рис.22), следовательно интерфейс Winsock можно применить для написания программ, взаимодействующих по этому протоколу.

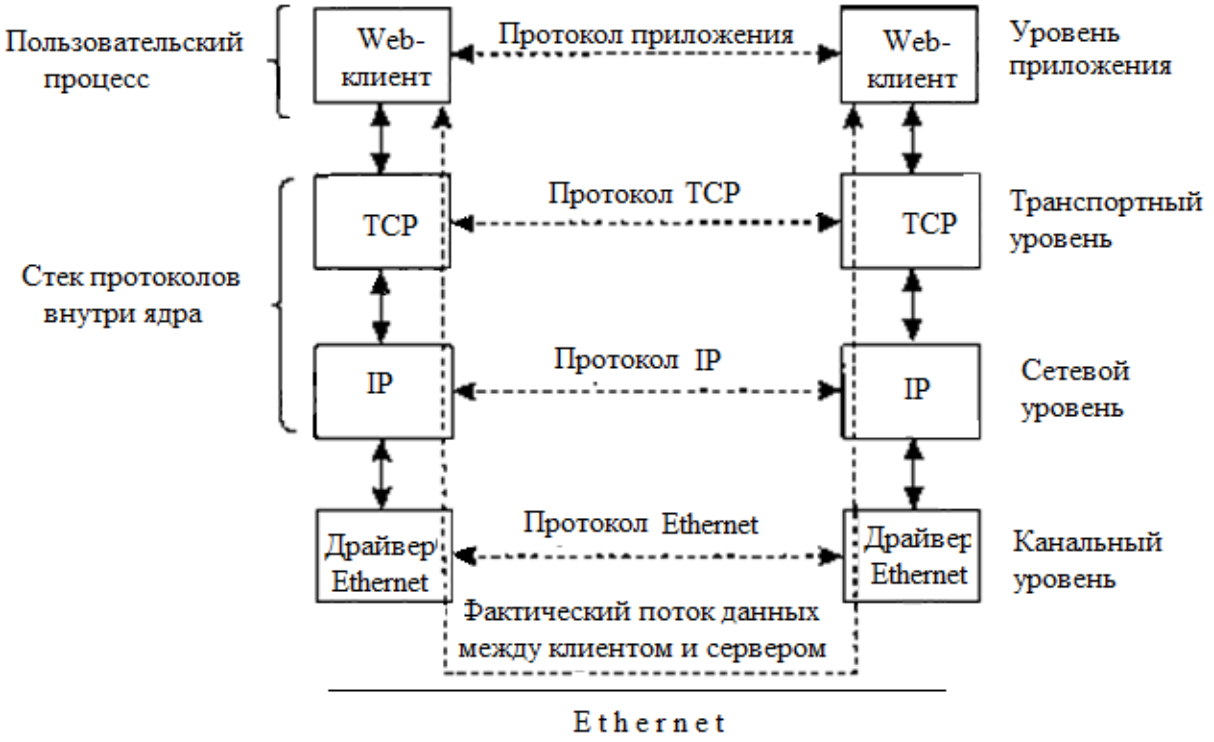


Рис. 22. Организация Web-взаимодействий

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
https://www.kubsu.ru/user – URI, позволяющий вызвать программу входа в личный кабинет, тогда первая часть
https://www.kubsu.ru представляет собой URL (указывает имя хоста), а
user есть URN, идентифицирующее имя ресурса.

В рамках данной главы излагаются основные сведения и демострируются практические навыки в написании программ, работающих по HTTP протоколу с использованием системного вызова socket на языке программирования C++, имеющем средства работы с сокетами.

7.1. Принципы работы

Все HTTP-транзакции имеют один общий формат. Каждый запрос клиента и ответ сервера состоят из трех частей: строки запроса (ответа), раздела заголовка и тела. Клиент инициирует транзакцию следующим образом:

1. Клиент устанавливает связь с сервером по назначенному номеру порта (по умолчанию 80). Затем клиент посылает запрос документа, указав HTTP-команду, называемую методом, адрес документа и номер версии HTTP. Например, в запросе GET /index.html HTTP/1.0 используется метод GET, которым с помощью версии 1.0 HTTP запрашивает документ index.html.

2. Клиент посылает информацию заголовка (необязательную), чтобы сообщить серверу информацию о своей конфигурации и данные о форматах документов, которые он может принимать. Вся информация заголовка указывается построчно, при этом в каждой строке приводится имя и значение. Например, приведенный ниже заголовок, посланный клиентом, содержит его имя и номер версии, а также информацию о некоторых предпочтительных для клиента типах документов:

```
USER-AGENT: Mozilla/2.02Gold (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

Завершается пустой строкой.

3. Послав запрос и заголовки, клиент может отправить и дополнительные данные. Эти данные используются главным образом теми CGI-программами, которые применяют метод POST. Клиенты могут их использовать для помещения отредактированной страницы обратно на Web-сервер.

Ответы сервера на запросы клиента:

1. Первая часть ответа сервера – строка состояния, содержащая три поля: версию HTTP, код состояния и описание. Поле версии содержит номер версии HTTP, которой данный сервер пользуется для передачи ответа. Код состояния – это трехразрядное число, обозначающее результат обработки сервером запроса клиента. Описание, следующее за кодом состояния, представляет собой текст, поясняющий код состояния. Например, строка состояния ответа HTTP/1.0. 200 OK говорит о том, что сервер для ответа использует версию HTTP 1.0. Код состояния 200 означает, что запрос клиента был успешным и затребованные данные будут переданы после заголовков.

2. После строки состояния сервер передает клиенту информацию заголовка, содержащую данные о самом сервере и затребованном документе.

Пример HTTP-заголовка:

```
Date: Fri, 20 Mar 1999 08:17:58 GMT
Server: NCSA/1.5.2
Last-modified: Mon, 17 Jun 1996 21:53:08 GMT
Content-type: text/html
Content-lenght: 2482
```

Завершает заголовок пустая строка.

3. Если запрос клиента успешен, то посылаются затребованные данные. Это может быть копия файла или результат выполнения CGI-программы. Если запрос клиента удовлетворить нельзя, передается разьяснение причин, по которым сервер не смог выполнить данный запрос. В HTTP 1.0 за передачей сервером

затребованных данных следует разъединение с клиентом и транзакция считается завершенной, если не передан заголовок

```
Conection: Keep Alive.
```

В HTTP 1.1 сервер по умолчанию не разрывает соединения и клиент может посылать другие запросы. Поскольку во многие документы встроены другие документы – изображения, кадры, аплеты и т. д., это позволяет сэкономить время и затраты клиента, которому в противном случае пришлось бы для получения всего одной страницы многократно соединяться с одним и тем же сервером. Таким образом, в HTTP 1.1 транзакция может циклически повторяться, пока клиент или сервер не закроет соединение явно.

HTTP **не сохраняет** информацию по транзакциям, поэтому в следующей транзакции приходится начинать все заново. Преимущество состоит в том, что HTTP-сервер может обслужить в заданный промежуток времени гораздо больше клиентов, ибо устраняются дополнительные расходы на отслеживание сеансов от одного соединения к другому. Есть и недостаток: для сохранения информации по транзакциям более сложные CGI-программы должны пользоваться скрытыми полями ввода или внешними средствами.

Запросы клиента разбиваются на три раздела. Первая строка сообщения всегда содержит: HTTP-команду, называемую методом; URI, который обозначает запрашиваемый клиентом файл или ресурс; номер версии HTTP. Следующие строки запроса клиента содержат информацию заголовка. Информация заголовка содержит сведения о клиенте и информационном объекте, который он посылает серверу. Третья часть клиентского запроса представляет собой тело содержимого – собственно данные, посылаемые серверу.

Метод – это HTTP-команда, с которой начинается первая строка запроса клиента. Метод сообщает серверу о цели запроса. Наиболее важными являются три основных метода: GET, HEAD и POST. При задании имен методов учитывается регистр, поэтому GET и get различаются.

Метод GET – это запрос информации, расположенной на сервере по указанному URL. Результат запроса GET может

представлять собой, например, файл, доступный для сервера, результат выполнения программы или CGI-сценария, выходную информацию аппаратного устройства и т. д. Если клиент пользуется в своем запросе методом GET, сервер отвечает строкой состояния, заголовками и затребованными данными. Если сервер не может обработать запрос вследствие ошибки или отсутствия полномочий, он, как правило, посылает в информационном разделе ответа текстовое пояснение. Тело информационного содержимого запроса GET всегда пустое. Для идентификации указанных в запросе клиента файла или программы обычно используется полное имя обьекта на сервере. Ниже приведен пример корректного запроса GET на получение файла.

Клиент посылает запрос:

```
GET /index.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/2.02Gold (WinNT; I)
Host: www.ora.com
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, */*
```

Сервер отвечает:

```
HTTP/1.0 200
Document follows Date: Fri, 20 Sep 1996 08:17:58
GMT
Server: NCSA/1.5.2
Content-type: text/html
Content-lenght: 2482
```

(*далее следует тело документа*)

Метод GET используется также для передачи выходной информации в CGI-программы посредством тегов форм. Поскольку тело запроса GET пусто, входные данные присоединяются к URL в строке GET запроса. Если в теге задано значение атрибута method="GET", то пары ключ–значение, представляющие собой введенные данные из формы, присоединяются к URL после

вопросительного знака. Пары отделяются друг от друга амперсандом (&). Например, по запросу

```
GET /cgi-bin/birthday.pl ? month = august & date=24 HTTP/1.0
```

сервер передаст в CGI-программу birthday.pl значения month и date, указанные в созданной на клиенте форме. Входные данные в конце URL кодируются в спецификации CGI. Чтобы специальные символы интерпретировались обычным образом, используются шестнадцатеричные коды.

Аналогичным образом в методе GET может передаваться информация о дополнительных путях. При этом дополнительный путь указывается после URL, т. е.

```
/cgi-bin/display.pl/cgi/cgi_doc.txt
```

Сервер определяет, где заканчивается имя программы (display.pl). Все данные, которые следуют за именем программы, интерпретируются как дополнительный путь.

Метод HEAD аналогичен методу GET, за исключением того, что сервер ничего не посылает в информационной части ответа. Метод HEAD запрашивает только информацию заголовка о файле и ресурсе. Информация заголовка запроса HEAD должна быть такой же, как в запросе GET. Этот метод используется, когда клиент хочет найти информацию о документе, не получая его. Для метода HEAD существует множество приложений. Например, клиент может затребовать следующую информацию:

– время изменения документа (эти данные полезны для запросов, связанных с кэш-памятью);

– размер документа (необходим для компоновки страницы, оценки времени передачи, определения необходимости запроса более компактной версии документа);

– тип документа (позволяет клиенту изучать документы только определенного типа);

– тип сервера.

Следует отметить, что большая часть информации заголовка, которую посылает сервер, не является обязательной и может предоставляться не всеми серверами.

Ниже приведен пример HTTP-транзакции с использованием запроса HEAD.

Клиент, например браузер, посылает запрос:

```
HEAD /index.html HTTP/1.0
Connection: Kep-Alive
User-Agent: Mozilla/2.02Gold (WinNT; I)
Host: www.ora.com
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, */*
```

Сервер отвечает:

```
HTTP/1.0 200 Document follows
Date: Fri, 20 Sep 1996 08:17:58 GMT
Server: NCSA/1.5.2
Last-modified: Mon, 17 Jun 1996 21:53:08 GMT
Content-type: text/html
Content-lenght: 2482
```

(*Тело содержимого в ответ на запрос* HEAD *не передается*.)

Метод POST позволяет посылать на сервер данные в запросе клиента. Эти данные направляются в программу обработки данных, к которой сервер имеет доступ (например, в CGI-сценарий). Метод POST может использоваться во многих приложениях. Например, его можно применять при передаче входных данных для:

- сетевых служб (таких как телеконференции);
- программ с интерфейсом в виде командной строки;
- аннотирования документов на сервере;
- выполнения операций в базах данных.

Данные, посылаемые на сервер, находятся в теле содержимого запроса клиента. По завершении обработки запроса POST и заголовков сервер передает тело содержимого в программу, заданную URL. В качестве схемы кодирования с методом POST используется URL-кодирование, которое позволяет преобразовывать данные форм в список переменных и значений для CGI-обработки.

Ниже приведен небольшой пример запроса клиента с использованием метода POST. Клиент посылает на сервер данные о дне рождения, введенные в форму:

```
    POST /cgi-bin/birtday.pl HTTP/1.0
    User-Agent: Mozilla/2.02Gold (WinNT; I)
    Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, */*
    Host: www.ora.com Content-type: application/x-
www-form-urlencoded
    Content-lenght: 20

    month=august&date=24
```

Другие методы. Приведенные ниже методы также определены, хотя и используются не столь часто:

LINK – связывает информацию заголовка с документом на сервере;

ULINK – отменяет связь информации заголовка с документом на сервере;

PUT – помещает тело содержимого запроса по указанному URL;

DELETE – удаляет данные, находящиеся на сервере по заданному URL;

OPTIONS – запрашивает информацию о коммутационных параметрах сервера. Чтобы запросить данные обо всем сервере в целом, вместо URL запроса следует использовать символ *;

TRACE – требует, чтобы тело содержимого запроса было возвращено без изменений. Используется для отладки.

Ответ сервера на запрос клиента состоит из трех частей. Первая строка – это строка состояния сервера, которая содержит номер версии HTTP, число, обозначающее состояние запроса, и краткое описание состояния. После строки ответа следует информация заголовка и тело содержимого, если таковое имеется.

Коды ответов сервера:

100–199 – Информационный.

`100 Continue`

Начальная часть запроса принята и клиент может продолжать передачу запроса.

`101 Switching Protocols`

Сервер выполняет требование клиента и переключает протоколы в соответствии с указанием, данным в поле заголовка `Upgrade`.

200–299 – Запрос клиента успешен.

`200 OK`

Запрос клиента обработан успешно, и ответ сервера содержит затребованные данные.

300–399 – Запрос клиента переадресован, необходимы дальнейшие действия.

400–499 – Запрос клиента является неполным.

500–599 – Ошибки сервера.

7.2. Примеры HTTP-взаимодействий

В качестве примера приведем код программы, использующей интерфейс `socket` для получения с удалённого стандартного HTTP сервера нужной HTML странички. Очевидно, чтобы получить с

нужного сайта кусочек HTML кода, необходимо выполнить обращение по известному адресу. Например, чтобы взять HTML страничку ru-ru.html с сайта www.microsoft.com, нужно обратиться по адресу www.microsoft.com/ru-ru.html.

В программе для обмена данными с сервером используется HTTP протокол, хотя можно использовать FTP и другие протоколы. Для интерфейса Winsock, в принципе, все равно, какой протокол будет использоваться для связи с сервером. Его задача – доставить данные серверу и получить ответ.

Программа выполняет функции клиента: используя вызов socket, выполняет HTTP запрос к серверу, получает ответ сервера и выводит его в стандартный поток вывода. Таким образом, по функционалу представленная программа будет отдалённо напоминать Web-браузер.

Структура программы такова:

1. Определение IP-адреса сервера.
2. Устанавление соединения.
3. Передача серверу запроса (в HTTP формате).
4. Получение ответа и вывод его на экран.
5. Закрытие соединения и завершение работы.

Заметим, что данный код будет работать только на машинах, имеющих «прямой» выход в Интернет, т. е. этот код не будет работать, если вы сидите за прокси-сервером и т. п.

Код программы HTTP-клиента:

```
#include "winsock2.h"
#pragma comment (lib, “Ws2_32.lib”)
#define request
"GET /user HTTP/1.0 \r\nHost: www.kubsu.ru
\r\n\r\n“
// HTTP запрос по.адресу :www.kubsu.ru/user
#define MAX_PACKET_SIZE 4096

int main()
```

```
    { WSADATA ws;
    SOCKET s;
    sockaddr_in adr;
    hostent* hn;
    char buff [MAX_PACKET_SIZE];

// Инициализация Winsock версии 1.1
    if (WSAStartup (0x0101, &ws) != 0)
    {return -1;}                // Error

// Создаём сокет
    if(INVALID_SOCKET ==
        (s = socket(AF_INET,SOCK_STREAM,0)))
    { return -1; }              // Error

// Получаем IP-адрес сервера по символьному имени
    if(NULL==( hn = gethostbyname("www.kubsu.ru")))
    {return -1;  }              // Error
// Задаем парамеры сервера в адресной структуре
    adr.sin_family = AF_INET; ..//домен
    ((unsigned long *)&adr.sin_addr)[0] =
    ((unsigned  long  **)  hn->h_addr_list)[0][0];
//адрес
    adr.sin_port = htons (80);      //порт
// Устанавливаем соединение с сервером
    if (SOCKET_ERROR ==
    connect (s, (sockaddr* )&adr, sizeof (adr)))
        {return -1; }               // Error
// Посылаем запрос серверу
    if (SOCKET_ERROR == send (s, &request,
        sizeof (request), 0) )
    {return -1; }               // Error
    int len = 0;                // ждём ответа

do //Получаем данные по частям, пока len не 0
    {
```

```
    if(SOCKET_ERROR ==
        (len = recv (s, (char *)&buff,max_size,0)))
{int res = WSAGetLastError (); return -1;} // Error
    buff[len]=0;
cout << buff;   // вывод полученной порции данных
    } while (len!=0);

// закрываем соединение
    if (SOCKET_ERROR == closesocket (s) )
    {  return -1; }          // error

    return 1;
    }
```

Программа выдает HTML код страницы после системной информации, переданной в ответе сервера. Если открыть ссылку http://www.kubsu.ru/user в окне браузера, то появится страница для входа в личный кабинет пользователя. Именно исходный HTML код этой страницы и будет результатом выполнения представленной в примере программы-клиента.

В качестве второго примера рассмотрим сетевой проект на C++, содержащий клиентскую и серверную компоненты. Для взаимодействия между компонентами используем протокол HTTP. Соединение устанавливается на основе интерфейса сокетов.

Работа сервера заключается в следующем: принятие запроса от HTTP-клиента; формирование сервером ответа согласно протоколу HTTP, частью тела которого является содержание принятого запроса; отправка ответа обратно клиенту.

Код сервера проекта:

```
    #include <iostream>
    #include <sstream>
    #include <string>
    #include <WinSock2.h>
    #include <WS2tcpip.h>
//для использования новых интерфейсных функций
```

```
    #pragma comment(lib, "Ws2_32.lib")
    using std::cerr;

    int main()
    {
    WSADATA wsaData;
int result = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if (result != 0)
{cerr << "WSAStartup failed: "<< result << "\n";
    return result;    }
// Создание сокета
    int listen_socket =
    socket(addr->ai_family, addr->ai_socktype,
    addr->ai_protocol);
    if (listen_socket == INVALID_SOCKET) {
cerr<<"Error at socket: "<<WSAGetLastError()<<"\n";
    freeaddrinfo(addr);
    WSACleanup();
    return 1; }
// Привязываем сокет к IP-адресу
    result = bind (listen_socket,
        addr->ai_addr,(int)addr->ai_addrlen);
    if (result == SOCKET_ERROR)
    {
    cerr << "bind failed with error: "
        << WSAGetLastError() << "\n";
    freeaddrinfo(addr);
    closesocket(listen_socket);
    WSACleanup();
    return -1; }
// Инициализируем слушающий сокет
if(listen(listen_socket,SOMAXCONN) == SOCKET_ERROR)
    {
    cerr << "listen failed with error: "
        << WSAGetLastError()<<"\n";
    closesocket(listen_socket);
```

```
    WSACleanup();
    return -1;
    }
    const int max_client_buffer_size = 1024;
    char buf[max_client_buffer_size];
    int client_socket = INVALID_SOCKET;
// Принимаем входящие соединения
    for (;;)
    {
client_socket = accept(listen_socket, NULL, NULL);
    if (client_socket == INVALID_SOCKET)
    {
cerr <<"accept failed: "<<WSAGetLastError()<< "\n";
    closesocket(listen_socket);
    WSACleanup();
    return 1;
    }
std::stringstream response; // для ответа клиенту
std::stringstream response_body;    // тело ответа

    result =
recv(client_socket,buf, max_client_buffer_size,0);
    if (result == SOCKET_ERROR)
    {
// Ошибка получения данных
    cerr << "recv failed: " << result << "\n";
    closesocket(client_socket);
    }
    else if (result == 0)
    {
// Соединение закрыто клиентом
    cerr << "connection closed...\n";
    }
    else if (result > 0)
    {
// Данные запроса успешно получены
```

```
    buf[result] = '\0';
// Формируем тело ответа (HTML)
        response_body <<
    "<title>Test C++ HTTP Server</title>\n"
    << "<h1>Test page</h1>\n"
    << "<p>This is body of the test page...</p>\n"
    << "<h2>Request headers</h2>\n"
    << "<pre>" << buf << "</pre>\n"
//включаем в ответ полученный запрос
<<"<em><small>Test C++ Http Server</small></em>\n";

// Формируем весь ответ вместе с заголовками
    response <<
    "HTTP/1.1 200 OK\r\n"<< "Version: HTTP/1.1\r\n"
    << "Content-Type: text/html; charset=utf-8\r\n"
<<"Content-Length: "<<response_body.str().length()
    << "\r\n\r\n" << response_body.str();

// Отправляем ответ клиенту
Result = send(client_socket,response.str().c_str(),
        response.str().length(), 0);

    if (result == SOCKET_ERROR)
// ошибка при отправке данных
{cerr<<"send failed:“ << WSAGetLastError() <<"\n";}

// Закрываем соединение с клиентом
    closesocket(client_socket);
    }
    }                        // конец for(;;)

    // финишные операции
    closesocket(listen_socket);
    freeaddrinfo(addr);
    WSACleanup();
    return 0;
```

```
}
```

В программе сервера при программировании сокетов используются новые интерфейсные функции. С этой целью в заголовок добавлена инструкция компилятору:

```
#include <WS2tcpip.h>
```

Сервер обучен общаться с клиентами согласно протоколу HTTP, поэтому он способен принимать запросы не только от клиента данного проекта, но и от других HTTP-клиентов, например, от Web-браузеров или универсальных клиентов как telnet. Ниже приведены примеры взаимодействия этого сервера с различными клиентами.

Код клиента проекта:

```
#include <string>
#define _WINSOCK_DEPRECATED_NO_WARNINGS
// подавление предупреждений библиотеки winsock2
#include <winsock2.h>
#include <iostream>
#pragma comment (lib,"Ws2_32.lib")
#pragma warning(disable: 4996)
// подавление предупреждения 4996
using namespace std;
#define request
"GET/ HTTP/1.1\r\n Host: localhost \r\n\r\n"
//html запрос.
#define max_packet_size   65535

int main()
{WSADATA    ws;
    SOCKET      s;
sockaddr_in adr;
    HOSTENT*    hn;
    char        buff [max_packet_size];
// инициализация
```

```
    if (WSAStartup (0x0202, &ws) != 0)
    { return -1; }          // ERROR
// создаём сокет
    if(INVALID_SOCKET==
    (s=socket (AF_INET,SOCK_STREAM,0) ) )
    {return -1; }           // ERROR
// получаем адрес
    if (NULL == (hn =gethostbyname ("localhost")))
    {return -1; }           // ERROR
/* заполняем  поля структуры adr для использования
ее в connect */
    adr.sin_family = AF_INET;
    ((unsigned long *)&adr.sin_addr)[0] =
    ((unsigned long **) hn->h_addr_list)[0][0];
    adr.sin_port = htons (8000);
// устанавливаем соединение с сервером
    if (SOCKET_ERROR ==
    connect(s,(sockaddr*)&adr, sizeof (adr)))
{int res = WSAGetLastError (); return -1; } //ERROR
// посылаем запрос серверу
    if (SOCKET_ERROR ==
    send (s, (char *)&request,sizeof(request),0))
{int res= WSAGetLastError ();return -1;}// ERROR
// ждём ответа
    int len = 0;
do
    {
    if (SOCKET_ERROR ==
(len = recv (s,(char *)&buff,max_packet_size, 0)))
    {int res = WSAGetLastError (); return -1; }
    buff[len]=0;
    cout << buff;
} while (len!=0);
//получаем данные по частям, пока не len != 0.
// закрываем соединение
    if (SOCKET_ERROR == closesocket (s))
```

```
{return -1; }        // ERROR
return 0;
}
```

Заметим, что в заголовочной части программы присутствуют инструкции

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#pragma warning(disable: 4996)
```

Они включены для подавления предупреждений 4996 в средах, требующих использования при программирования новой редакции Winsocket.

Представленные далее скрины иллюстрируют результаты взаимодействия с различными HTTP-клиентами запущенного в среде Visual Studio C++ сервера, слушающего сеть на порту 8000. В первых двух случаях черное окно принадлежит указанному серверу проекта. В первом случае (рис. 23) роль клиента выполняет универсальный клиент-браузер. Во втором (рис. 24) – программа-клиент, код которой на C++ представлен выше. В третьем случае (рис. 25) клиентом является команда telnet.

серверу через Web-

ента проекта, выпол

к серверу клиента `telne`

8. ПРОЦЕССЫ И ПОТОКИ

Распределенное сетевое приложение состоит из одного или нескольких процессов. ***Процесс (process)***, в самых простых терминах, является исполняемой программой. В контексте процесса работает один или несколько потоков. ***Потоки (`thread`)*** – основные модули программы, среди которых операционная система распределяет процессорное время. Поток может выполнять любую часть кода процесса, включая части, в настоящее время выполняемые другим потоком. ***Нить (`fiber`)*** – секция исполняемого кода, для которой должна быть вручную установлена очередность обслуживания прикладной программой. Нити работают в среде потоков, которые устанавливают очередность их обслуживания.

Работа с процессами и потоками имеет много общих моментов, но и различий у них немало. В основном эти различия кроются в самой сути этих понятий. Для начала рассмотрим каждый из них подробней.

8.1. О ПРОЦЕССАХ

Каждому процессу предоставляются ресурсы, необходимые для выполнения программы. Процесс имеет виртуальное адресное пространство, выполняемый код, данные, дескрипторы объектов, системные переменные, основной приоритет и установку минимального и максимального рабочего размера.

Дескрипторы (Handle) и идентификаторы (ID) процесса

Новый процесс создается функцией `CreateProcess`, она возвращает дескрипторы нового процесса и его первичного потока. Эти дескрипторы создаются с полными правами доступа, и – как субъекты, проверяемые системой безопасности доступа – могут быть использованы в любой из функций, которые принимают дескрипторы потока или процесса. Эти дескрипторы могут быть унаследованы дочерними процессами, в зависимости от установки флажка наследования, когда они создаются. Дескрипторы допустимы до тех пор, пока не закроются, даже после того как процесс или поток, который они обозначают, завершил свою работу.

Функция `CreateProcess` также возвращает и идентификатор, который уникально идентифицирует процесс повсюду в системе. Процесс может использовать функцию `GetCurrentProcessId`, чтобы получить собственный идентификатор. Идентификатор допустим с момента, когда процесс создан, и до тех пор, пока процесс не завершит свою работу.

Если у Вас есть идентификатор процесса, Вы можете получить дескриптор процесса путем вызова функции `OpenProcess`. `OpenProcess` разрешает Вам определить права доступа дескриптора и может ли он быть унаследованным.

Между дескриптором и идентификатором процесса (или потока) существуют следующие основные различия:

1) дескриптор действует в пределах своего процесса, в то время как идентификатор работает на системном уровне. Таким образом, дескриптор действует только в пределах того процесса, в котором он был создан;
2) у каждого процесса только один идентификатор, но может быть несколько дескрипторов;
3) некоторым `API`-функциям требуется идентификатор, в то время как другим – дескриптор процесса.

Для получения псевдодескриптора собственного объекта процесса можно использовать функцию `GetCurrentProcess`. Этот псевдодескриптор допустим только для вызывающего процесса; он не может быть унаследован или продублирован для использования другими процессами. Получить действительный дескриптор процесса можно функцией `DuplicateHandle`.

8.2. О ПОТОКАХ

Каждый процесс начинается с единственного потока, часто называемого ***первичным потоком (primary thread)***, но он может создать дополнительные потоки из любого своего потока.

Все потоки процесса ***совместно*** используют свое виртуальное адресное пространство и системные ресурсы. Кроме того, каждый поток поддерживает обработчики особых ситуаций, приоритет диспетчеризации и набор структур, которые система будет

использовать, чтобы сохранять контекст потока до тех пор, пока он не будет обслужен. ***Контекст потока (thread context)*** включает в себя набор машинных регистров потока, стек ядра, блок конфигурации потока и стек пользователя в адресном пространстве потока процесса.

Функция `CreateThread` создает для процесса новый поток. Созданный поток должен определить начальный адрес кода, с которого новый поток должен исполняться. Как правило, начальный адрес – это название функции, определенной в коде программы. Эта функция получает единственный параметр и возвращает значение типа `DWORD.` Процесс может иметь одновременно несколько потоков, выполняющих ту же самую функцию.

Дескрипторы и идентификаторы потоков

Функция `CreateProcess` возвращает идентификатор и дескриптор первого потока, выполняющегося во вновь созданном процессе. Функция `CreateThread` тоже возвращает идентификатор потока, но область действия которого – вся система.

Для получения собственного `Id` потока используется функция `GetCurrentThreadId.` Собственный псевдодескриптора потока можно получить функцией `GetCurrentThread.` Как и в случае псевдодескрипторов процессов, псевдодескриптор потока может использоваться только для вызова процесса и не может наследоваться. Получение настоящего дескриптора потока по его псевдодескриптору реализуется функцией `DuplicateHandle.`

Приоритет потоков

Термин многозадачность, или мультипрограммирование, обозначает возможность управлять несколькими процессами (или несколькими потоками) на базе одного процессора. Многопроцессорной обработкой называется управление некоторым числом процессов или потоков на нескольких процессорах.

Компьютер может одновременно выполнять команды двух разных процессов только в многопроцессорной среде. Однако даже на одном процессоре с помощью переключения задач можно создать впечатление, что одновременно выполняются команды нескольких процессов.

Следующий пример демонстрирует, как создать новый поток, который выполняет локально определяемую функцию THREADFUNC.

```
// вывод значения параметра потока
    #include "stdafx.h"
    #include <windows.h>
    #include <iostream>
    using namespace std;
    DWORD WINAPI ThreadFunc(LPVOID lpparam)
    {
    DWORD szm;
    szm=*(DWORD*)lpparam;
    cout << "thream param="<<szm<< endl;

    return 0;
    }
    int main()
    {
    DWORD thiD, thp = 21;
    HANDLE thh;

    thh = CreateThread(
    NULL,     // атрибуты безопасности по умолчанию
    0,    // размер стека используется по умолчанию
    ThreadFunc,   // функция потока
    &thp, // аргумент функции потока
    0, // флажки создания используются по умолчанию
    &thiD); // возвращает идентификатор потока
/* при успешном завершении проверяет возвращаемое
значение */
    if (thh == NULL)
    {
        cout<< "CreateThread failed." ;
        }
        else
```

```
    {
    cin.get();
    CloseHandle( thh );
    }
}
```

В этом примере параметром функции потока является указатель на значение. Это может *быть* ***указатель на любой тип данных или структуру или это может быть пропущено совсем, при помощи*** передачи указателя `NULL` и удаления ссылок на параметр `в ThreadFunc`.

Опасно передавать адрес локальной переменной*!* Например, если создающий поток заканчивает работу перед созданием нового потока, потому что указатель становится недопустимым. Вместо этого или передайте указатель в динамически распределяемую память, или заставьте создающий поток ждать до тех пор, пока новый поток не завершит свое формирование. Данные могут также быть переданы и из создающего потока в новый поток использованием глобальных переменных. С глобальными переменными обычно необходимо синхронизировать доступ ко многим потокам.

8.3. Многопоточность

В многоядерных процессорах или многопроцессорных системах многопоточность осуществляется тем, что на разных ядрах или процессорах параллельно исполняются несколько процессов. А вот что касается одноядерного компьютера, то многопоточность осуществляется путем разделения рабочего времени процессора между процессами. Операционная система по очереди дает на выполнения процессору некоторое количество инструкций от каждого процесса. Получается, что в реальности процессы не выполняется одновременно, а только имитируется их одновременное выполнение. Это свойство операционной системы и называется многопоточностью. Многопоточность используется в тех случаях, когда параллельное выполнение некоторых задач приводит к более эффективному использованию ресурсов вычислительной системы.

Надо отметить, что в многопоточном приложении потоки выполняются в адресном пространстве приложения.

8.4. Несинхронизированные потоки

Приведем пример, иллюстрирующий работу с несинхронизированными потоками. Основной цикл, который является основным потоком процесса, выводит на экран содержимое глобального массива целых чисел. Поток, названный `Thread`, непрерывно заполняет глобальный массив целых чисел.

```
#include <windows.h>
#include <string>
#include <iostream>
using namespace std;
int num=0;
int a[5];

DWORD WINAPI TR(LPVOID pr)
{int i;

while (1)
{
for (i=0; i<5; i++) a[i]=num;

num++;
}
return 0;
}
int main()
{ DWORD thID;
CreateThread (NULL,NULL,TR,NULL,NULL,&thID);
int k=0;
while (1)
{ k++;
```

```
cout<<a[0]<< " "<<a[1]<< " "<<a[2]<< " "<<a[3]<< "
"<<a[4]<< endl;

        if (k % 10 == 0) cin.get();
    }
        return 0;
    }
```

По результатам работы несинхронизированного многопоточного процесса

…

1118956576 1118956576 1118956576 1118956576 1118956576

1119012135 1119012133 1119012133 1119012132 1119012132

1509763034 1509763034 1509763034 1509763033 1509763033

1509817926 1509817926 1509817926 1509817926 1509817926

1509873689 1509873689 1509873689 1509873689 1509873688

1509932173 1509932172 1509932172 1509932171 1509932170

…

видно, что основной поток (сама программа) и поток `Thread` действительно работают параллельно, т. е. основной поток выводит массив во время его заполнения потоком `Thread`. В силу чего элементы, выведенные в одну строку, могут не совпадать по значению.

8.5. Синхронизация потоков

Выполняющимся потокам часто необходимо каким-то образом взаимодействовать. Например, если несколько потоков пытаются получить доступ к некоторым глобальным данным, то каждому потоку нужно предохранять данные от изменения другим потоком. Иногда одному потоку нужно получить информацию о том, когда другой поток завершит выполнение задачи. Такое

взаимодействие обязательно между потоками как одного, так и разных процессов.

Синхронизация потоков (*thread synchronization*) – это обобщенный термин, относящийся к процессу взаимодействия и взаимосвязи потоков. Синхронизация потоков требует привлечения в качестве посредника самой операционной системы. Потоки не могут взаимодействовать друг с другом без ее участия.

В Win32 существует несколько методов синхронизации потоков. В зависимости от конкретной ситуаций один метод более предпочтителен, чем другой. Методы синхронизации потоков одного или нескольких процессов основаны на использовании объектов синхронизации и функций ожидания. Объекты синхронизации могут находиться в одном из двух состояний – ***signaled*** («свободно») или ***not signaled*** («занято»). В зависимости от состояния объекта синхронизации один поток может узнать об изменении состояния других потоков или общих (разделяемых) ресурсов. Функции ожидания блокируют выполнение потока до тех пор, пока заданный объект находится в состоянии ***not signaled***. Таким образом, поток, которому необходим эксклюзивный доступ к ресурсу, должен выставить какой-либо объект синхронизации в несигнальное состояние, а по окончании – сбросить его в сигнальное. Остальные потоки должны перед доступом к этому ресурсу вызвать функцию ожидания, которая позволит им дождаться освобождения ресурса.

Функции ожидания

Функции синхронизации делятся на две основные категории: функции, ожидающие единственный объект, и функции, ожидающие один из нескольких объектов.

Функции, ожидающие единственный объект. Простейшей функцией ожидания является функция:

```
DWORD WaitForSingleObject(
THANDLE hhandle,    // идентификатор объекта
DWORD dwmilliseconds) // период ожидания
```

Эта функция имеет два аргумента:

(1) дескриптор (handle) какого-либо объекта синхронизации;

(2)·тайм-аут ожидания, т.е. максимальное время ожидания (в миллисекундах).

Эта функция – блокирующая: если в момент ее вызова указанный объект находится в несигнальном состоянии, то вызвавшая нить переводится в состояние ожидания (разумеется, пассивного!). Ожидание, как правило, завершается в одном из двух случаев: либо объект переходит в сигнальное состояние, либо истекает заданный тайм-аут. Значение, возвращаемое функцией, позволяет узнать, какая из причин пробуждения нити имела место.

Список возвращаемых функцией значений:

WAIT_ABANDONED	Поток, владевший объектом, завершился, не переведя объект в сигнальное состояние
WAIT_OBJECT_0	Объект перешел в сигнальное состояние
WAIT_TIMEOUT	Истек срок ожидания. Обычно в этом случае генерируется ошибка либо функция вызывается в цикле до получения другого результата
WAIT_FAILED	Произошла ошибка (например, получено неверное значение hHandle). Более подробную информацию можно получить, вызвав GetLastError

Если в момент вызова функции указанный объект уже находился в сигнальном состоянии, то ожидания не происходит, функция сразу же возвращает результат.

Если задана нулевая величина тайм-аута, то выполнение функции сводится к проверке, находится ли в данный момент объект в сигнальном состоянии.

Если значение тайм-аута равно константе `INFINITE`, то время ожидания не ограничено, т. е. нить может пробудиться только по сигнальному состоянию объекта.

Функции, ожидающие несколько объектов

Иногда требуется задержать выполнение потока до срабатывания одного или сразу всех из группы объектов. Для этого используется функция:

```
    DWORD WaitForMultipleObjects(
    DWORD Count, // Задает количество объектов
lpHandles HandleArray, // Адрес массива объектов
BOOL WaitAll,
/* Задает, требуется ли ожидание всех
объектов или любого */
    DWORD Milliseconds) // Период ожидания
```

Она отличается тем, что вместо единственного дескриптора (handle) принимает в качестве аргументов адрес массива, содержащего несколько дескрипторов, и количество этих дескрипторов (размер массива). Как и в предыдущем случае, задается величина тайм-аута. Дополнительный, четвертый аргумент – булевский флаг режима ожидания. Значение FALSE означает ожидание любого объекта (достаточно, чтобы хоть один из них перешел в сигнальное состояние), TRUE – ожидание всех объектов (ожидание завершится, только когда все объекты окажутся в сигнальном состоянии).

Возвращаемое значение позволяет определить причину пробуждения: (1) переход одного из заданных объектов в сигнальное состояние, и какого именно объекта; (2) истечение тайм-аута ожидания. Функция возвращает одно из следующих значений:

Число в диапазоне от **WAIT_OBJECT_0** *до* **WAIT_OBJECT_0 + Count – 1**	Если `WaitAll` равно `TRUE`, то это число означает, что все объекты перешли в сигнальное состояние. Если `FALSE` — то, вычтя из возвращенного значения `WAIT_OBJECT_0`, мы получим индекс объекта в массиве `lpHandles`
Число в диапазоне от **WAIT_ABANDONED_0** до **WAIT_ABANDONED_0 + nCount – 1**	Если `WaitAll` равно `TRUE`, это означает, что все объекты перешли в сигнальное состояние, но хотя бы один из владевших ими потоков завершился, не сделав объект сигнальным Если `FALSE` — то, вычтя из возвращенного значения `WAIT_ABANDONED_0`, мы получим в массиве `lpHandles` индекс объекта, при этом поток, владевший этим объектом, завершился, не сделав его сигнальным

Кроме двух названных, имеется еще ряд функций ожидания. `WaitForSingleObjectEx` и `WaitForMultipleObjectsEx` ожидают завершения асинхронных операций ввода/вывода для указанного файла. Функция `MsgWaitForMultipleObjects` позволяет, наряду с объектами синхронизации, использовать для пробуждения появление сообщений, ожидающих обработки. Функция `SignalObjectAndWait` переводит один объект в сигнальное состояние и тут же начинает дожидаться другого объекта.

8.5.1. Критические секции

Один из методов синхронизации потоков состоит в использовании критических секций (***Critical Sections***). Этот метод может использоваться только для синхронизации потоков только одного процесса (рис.26).

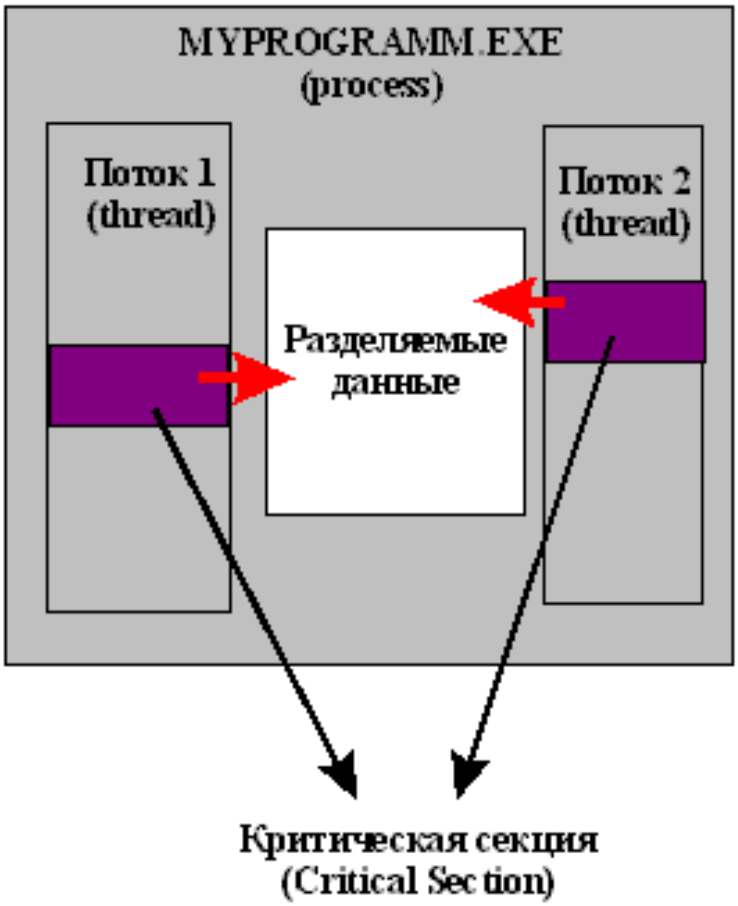


Рис. 26. Критическая секция для синхронизации потоков

Библиотека WIN32 API для работы с критическими секциями предлагает ряд функций API и тип данных Critical_Section. Для использования критической секции нужно создать переменную данного типа. Затем проинициализировать ее перед использованием с помощью функции InitializeCriticalSection(). Для того чтобы войти в секцию, нужно вызвать функцию EnterCriticalSection(), а после завершения работы – LeaveCriticalSection(). Тот поток, который обратится к секции, в которой сейчас другой поток, будет блокирован до тех пор, пока критическая секция не будет освобождена. Саму критическую секцию можно удалить функцией DeleteCriticalSection(). Для того чтобы обойти блокировку потока при обращении к занятой секции, есть функция TryEnterCriticalSection(), которая позволяет проверить критическую секцию на занятость.

Возьмем за основу предыдущий пример и предположим, что основной поток должен читать данные из массива только после обработки массива в параллельном процессе. Если код, используемый для инициализации массива, поместить в критическую секцию, то другие потоки не смогут войти в этот участок кода до тех пор, пока первый поток не завершит его выполнение. Результатом подобной модификации явится пример использования критических секций для синхронизации разделяемых данных:

```
#include <windows.h>
#include <iostream>
using namespace std;

Critical_Section cs;
int a[5];

DWORD WINAPI tr(LPVOID pr)
{int i, num=0;

while (1)
```

```
    {
        EnterCriticalSection(&cs);
        for (i=0; i<5; i++) a[i]=num;
        LeaveCriticalSection (&cs);
        num++;
    }
    return 0;
    }

    int main()
    { DWORD thID;
    InitializeCriticalSection( &cs );
    CreateThread (NULL,NULL,tr,NULL,NULL,&thID);
    while (1)
    {
        EnterCriticalSection(&cs);
        for ( int i=0; i<5; i++) cout<<" "<<a[i];
        LeaveCriticalSection (&cs);
        cin.get();
    }
        return 0;
    }
```

Результат выполнения приложения с критической секцией:

```
0 0 0 0
23322543 23322543 23322543 23322543 23322543
45742213 45742213 45742213 45742213 45742213
54925473 54925473 54925473 54925473 54925473
66352753 66352753 66352753 66352753 66352753
71632843 71632843 71632843 71632843 71632843
86922973 86922973 86922973 86922973 86922973
99299400 99299400 99299400 99299400 99299400
112976136 112976136 112976136 112976136 112976136
124906920 124906920 124906920 124906920 124906920
…
```

Захват критической секции последовательно каждым из потоков приводит к появлению одинаковых значений для элементов каждой строки. Заметно, что вывод элементов строки в главном потоке происходит значительно медленнее, чем изменение элементов во вспомогательном потоке tr, этим объясняется разница между значениями соседних строк.

8.5.2. Мьютексы

Мьютекс ***(mutex, взаимоисключение)*** – это объект синхронизации, который можно использовать для синхронизации потоков из разных процессов. Мьютекс устанавливается в особое сигнальное состояние, когда не занят каким-либо потоком. Только один поток владеет этим объектом в любой момент времени, отсюда и название таких объектов – одновременный доступ к общему ресурсу исключается. Например, чтобы исключить запись двух потоков в общий участок памяти в одно и то же время, каждый поток ожидает, когда освободится мьютекс, становится его владельцем и только потом пишет что-либо в этот участок памяти. После всех необходимых действий мьютекс освобождается, предоставляя другим потокам доступ к общему ресурсу.

Мьютекс создается функцией `CreateMutex:`

```
    HANDLE CreateMutex
    (
LPSECURITY_ATTRIBUTES lpMutexAttributes,
// атрибут безопастности
BOOL bInitialOwner, // флаг начального владельца
LPCTSTR lpName              // имя объекта
    );
```

Результатом будет дескриптор объекта `mutex`, если такое имя уже есть, то дескриптор существующего. Функция `GetLastError()` при вызове будет выдавать `ERROR_ALREADY_EXISTS`.

Для освобождения объекта нужно вызвать функцию:

```
BOOL ReleaseMutex
(
    HANDLE hMutex   // дескриптор mutex
);
```

При успешном выполнении функция вернет ненулевое значение.

По окончании работы вызвав `CloseHandle()`закрываем дескриптор.

Используя метод `CreateMutex,` два (или более) процесса могут создать *мьютекс* с одним и тем же именем. Первый процесс действительно создает мьютекс, а следующие процессы получают `handle` существующего объекта. Это дает возможность нескольким процессам получить `handle` одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает *мьютекс*. Если используется такой подход, желательно установить флаг `bInitialOwner` в `FALSE,` иначе возникнут определенные трудности при определении действительного создателя *мьютекса*.

Для синхронизации потоков одного процесса более эффективным подходом является использование критических секций.

Модифицируем предыдущий пример, используя для синхронизации объект `Mutex`.

```
#include <windows.h>
#include <iostream>

using namespace std;

HANDLE hMutex;
int a[5];

DWORD WINAPI TR(LPVOID pr)
{int i, num=0;
```

```
while (1)
{
    WaitForSingleObject( hMutex, INFINITE );
        for ( i = 0; i < 5; i++ ) a[ i ] = num;
        ReleaseMutex( hMutex );
num++;

}
return 0;
}
int main()
{ DWORD thID;
hMutex = CreateMutex( NULL, FALSE, NULL );
CreateThread (NULL,NULL,TR,NULL,NULL,&thID);
while (1)
{
    WaitForSingleObject( hMutex, INFINITE );
    for ( int i=0; i<5; i++) cout<<" "<<a[i];
    cout<<endl;
    ReleaseMutex( hMutex );
    cin.get();
}
    return 0;
}
```

Результат выполнения приложения с объектом Mutex для синхронизации потоков:

```
0 0 0 0 0
488559 488559 488559 488559 488559
850420 850420 850420 850420 850420
1169406 1169406 1169406 1169406 1169406
1419191 1419191 1419191 1419191 1419191
1804696 1804696 1804696 1804696 1804696
…
```

8.5.3. События

События (`Event`) используются в качестве сигналов о завершении какой-либо операции. Однако в отличие от мьютексов они не принадлежат ни одному потоку. Например, поток `A` создает событие с помощью функции `CreateEvent` и устанавливает объект в состояние «занято». Поток `B` получает дескриптор этого объекта, вызвав функцию `OpenEvent`, затем вызывает функцию `WaitForSingleObject`, чтобы приостановить работу до того момента, когда поток `A` завершит конкретную задачу и освободит указанный объект. Когда это произойдет, система выведет из состояния ожидания поток B, который теперь владеет информацией, что поток `A` завершил выполнение своей задачи.

Например, при асинхронных операциях ввода и вывода из одного устройства система устанавливает событие в сигнальное состояние, когда заканчивается какая-либо из этих операций. Один поток может использовать несколько различных событий в нескольких перекрывающихся операциях, а затем ожидать прихода сигнала от любого из них.

Существует два типа событий:

Тип объекта	Описание
Событие с ручным сбросом	Это объект, сигнальное состояние которого сохраняется до ручного сброса функцией ***ResetEvent***. Как только состояние объекта установлено в сигнальное, все находящиеся в цикле ожидания этого объекта потоки продолжают свое выполнение (освобождаются).
Событие с автоматическим сбросом	Объект, сигнальное состояние которого сохраняется до тех пор, пока не будет освобожден единственный поток, после чего система автоматически устанавливает несигнальное состояние события. Если нет потоков, ожидающих этого события, объект остается в сигнальном состоянии.

Для создания объекта события поток может использовать функцию `CreateEvent`:

```
    HANDLE CreateEvent
    (
    LPSECURITY_ATTRIBUTES lpEventAttributes,
// атрибут доступа адрес структуры
    BOOL ManualReset,
/* задает, будет Event переключаемым вручную (TRUE)
или автоматически (FALSE) */
    BOOL InitState;
/* задает начальное состояние. если TRUE - объект в
сигнальном состоянии */
    PCHAR lpName    // имя объекта (или NULL)
    );
```

Создающий событие поток устанавливает начальное состояние объекта синхронизации. В создающем потоке можно указать имя события. Потоки других процессов могут получить доступ к этому событию по имени, указав его в функции OpenEvent.

Состояние объекта синхронизации Event может быть установлено в ***сигнальное*** путем вызова функций `SetEvent` или `PulseEvent` .

Функция `PulseEvent` используется потоком для установки состояния события в ***сигнальное*** и затем сбрасывания состояния в ***несигнальное*** после освобождения соответствующего количества ожидающих потоков. В случае объектов с ручным сбросом освобождаются все ожидающие потоки. В случае объектов с автоматическим сбросом освобождается только единственный поток, даже если этого события ожидают несколько потоков. Если ожидающих потоков нет, `PulseEvent` просто устанавливает состояние события в несигнальное.

Модифицируем предыдущий пример таким образом, чтобы второй поток запускался каждый раз после того, как основной поток закончит печать содержимого массива, т. е. значения двух последующих строк будут отличаться строго на 1.

```
    #include <windows.h>
    #include <iostream>
```

```
    using namespace std;
    HANDLE hEvent1, hEvent2;
    int a[5];
    DWORD WINAPI TR(LPVOID pr)
    {int i, num=0;
    while (1)
    {
    WaitForSingleObject( hEvent2, INFINITE );
            for ( i = 0; i < 5; i++ ) a[ i ] = num;
            SetEvent( hEvent1 );
    num++;
    }
    return 0;
    }
    int main()
    { DWORD thID;
hEvent1 = CreateEvent( NULL, FALSE, TRUE,NULL );
// начальное значение 1-го объекта signaled
hEvent2 = CreateEvent( NULL, FALSE, FALSE,NULL );
// начальное значение 2-го объекта not signaled
    CreateThread (NULL,NULL,TR,NULL,NULL,&thID);
    while (1)
    {
        WaitForSingleObject( hEvent1, INFINITE );
        for ( int i=0; i<5; i++) cout<< " "<<a[i];
        cout<<endl;
    SetEvent( hEvent2 );
        cin.get();
    }
        return 0;
    }
```

Результат выполнения приложения с объектом Event:

```
10 10 10 10 10
11 11 11 11 11
```

```
12 12 12 12 12
13 13 13 13 13
14 14 14 14 14
15 15 15 15 15
…
```

показывает, что вывод строки происходит строго после каждого изменения значений элементов этой строки.

8.5.4. Семафоры (semaphore)

Существует еще один метод синхронизации потоков, в котором используются семафорные объекты API. В семафорах применен принцип действия мьютексов, но с добавлением одной существенной детали. В них заложена возможность подсчета ресурсов, что позволяет заранее определенному числу потоков одновременно войти в синхронизуемый участок кода. Данный объект синхронизации позволяет ограничить доступ потоков к объекту синхронизации на основании их количества.

При инициализации семафора ему передается количество потоков, которые к нему могут обратиться. Дальше ***при каждом обращении к ресурсу его счетчик уменьшается. Когда счетчик уменьшится до 0, к ресурсу обратиться больше нельзя***. При отсоединении потока от семафора его счетчик увеличивается, что позволяет другим потокам обратиться к нему. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается неустановленным (сброшенным).

Для создания семафора используется функция `CreateSemaphore()`. При создании семафора необходимо указать начальное и максимальное значения счетчика, используемые в дальнейшем для ограничения доступа к разделяемому ресурсу.

```
    HANDLE CreateSemaphore (
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
// атрибут доступа
    LONG lInitialCount,
// инициализированное начальное состояние счетчика
```

```
    LONG lMaximumCount,
// максимальное количество обращений
    LPCTSTR lpName  );      // имя объекта
```

При успешном выполнении функция вернет дискриптор семафора, в противном случае NULL.

Функция OpenSemaphore() осуществляет доступ к семафору.

После того как необходимость в работе с объектом отпала, нужно вызвать функцию ReleaseSemaphore(), чтобы увеличить значение счетчика. Счетчик может меняться от 0 до максимального значения.

```
    BOOL ReleaseSemaphore (
HANDLE hSemaphore,  // handle семафора
LONG lReleaseCount, // на сколько изменять счетчик
LPLONG lpPreviousCount  // предыдущее значение
                            );
```

При успешном выполнении возвращаемое значение ненулевое. После завершения работы для уничтожения семафора нужно вызвать CloseHandle().

Семафоры могут быть полезны при совместном использовании ограниченных ресурсов. Предположим, имеется три приложения, каждое из которых должно выполнить вывод на печать, а у компьютера только два параллельных порта. Установив семафор с начальным значением счетчика ресурсов, равным двум, можно заставить приложения запрашивать сервис печати только тогда, когда есть свободный параллельный порт.

Рассмотрим пример синхронизации потоков при помощи семафоров. В данном примере запускаются три потока, каждый из которых обращается к семафору. Для имитации работы потока с разделяемым ресурсом используем функцию задержки Sleep().

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
```

```
    using namespace std;

    HANDLE hSemaphore;
    LONG cMax = 2;

    DWORD WINAPI Tr1(LPVOID pr);
    DWORD WINAPI Tr2(LPVOID pr);
    DWORD WINAPI Tr3(LPVOID pr);

    DWORD tID1, tID2,tID3;
    HANDLE tD1,tD2,tD3;
    int main()
    {
        hSemaphore = CreateSemaphore(
            NULL,   // нет атрибута
            cMax,   // начальное состояние
            cMax,   // максимальное состояние
            NULL// без имени
                                    );

        if (!hSemaphore == NULL)
{ tD1=CreateThread(NULL, NULL,Tr1,NULL,NULL,&tID1);
        if (tD1==NULL)
    cout << "Error begin thread1 " << endl;

tD2=CreateThread(NULL, NULL,Tr2,NULL,NULL,&tID2);
        if (tD2==NULL)
        cout << "Error begin thread2 " << endl;
tD3=CreateThread(NULL, NULL,Tr3,NULL,NULL,&tID3);
        if (tD3==NULL)
        cout << "Error begin thread3 " << endl;
            Sleep(10000);
            cin.get();
            CloseHandle(hSemaphore);
        }
        else
```

```
    cout << "error create semaphore" << endl;
    return 0;
}

DWORD WINAPI Tr1(LPVOID pr)
{
    cout << "Tr1 Running \n";
    while(WaitForSingleObject
(       hSemaphore,
        1       // интервал ожидания милисек.
)!=WAIT_OBJECT_0)
    {
    cout << "Tr1 TIMEOUT \n";
    }
    cout << " Start Tr1 \n";
    Sleep(10000);
    if (ReleaseSemaphore
    (   hSemaphore, // указатель на светофор
            1,  // изменяет счетчик на 1
            NULL)
        )
        cout << " ReleaseSemaphore Ok Tr1 \n";
    CloseHandle(tD1);
    return 0;
}
DWORD WINAPI Tr2(LPVOID pr)
{
    cout << "Tr2 Running \n" ;
while(WaitForSingleObject(hSemaphore,1)!=
            WAIT_OBJECT_0)
    {
        cout << "Tr2 TIMEOUT \n";
    }
    cout << " Start Tr2 \n";
    Sleep(1000);
if (ReleaseSemaphore(hSemaphore,1,NULL))
```

```
        cout << " ReleaseSemaphore Ok Tr2 \n";
    CloseHandle(tD2);
        return 0;
    }
    DWORD WINAPI Tr3(LPVOID pr)
    {
        cout << "Tr3 Running \n";
    while(WaitForSingleObject(hSemaphore,1)!=
            WAIT_OBJECT_0)
        {
        cout << "Tr3 TIMEOUT \n";
        }
    cout << " Start Tr3" << endl;
        Sleep(1000);
        if (ReleaseSemaphore(hSemaphore,1,NULL))
            cout << " ReleaseSemaphore Ok Tr3 \n";
            CloseHandle(tD3);
        return 0;
    }
```

Результат работы приложения с объектом Semaphore :

```
Tr1 Running
Tr2 Running
Tr3 Running
 Start Tr1
 Start Tr2
Tr3 TIMEOUT
Tr3 TIMEOUT
Tr3 TIMEOUT
Tr3 TIMEOUT
…
Tr3 TIMEOUT
Tr3 TIMEOUT
Tr3 TIMEOUT
 ReleaseSemaphore Ok Tr2
```

```
Start Tr3
ReleaseSemaphore Ok Tr3
ReleaseSemaphore Ok Tr1
```

Семафор разрешает обращаться к ресурсу всего двум потокам. Поэтому поток `Tr3` будет ждать, пока кто-то освободит семафор. Первым завершил работу второй поток (`Sleep(1000)`). Тогда семафор позволил начать работу третьему потоку. Он тоже успел ее завершить (`Sleep(1000)`), а период работы первого потока в десять раз превышает период второго и третьего (`Sleep(10000)`). Поэтому первый завершает работу уже после третьего.

Сравнение объектов синхронизации

MSDN* News представляет сравнительные характеристики рассмотренных ранее механизмов синхронизации [23]:

Объект	Относительная скорость	Доступ нескольких процессов	Подсчет числа обращений к ресурсу
Критическая секция	*Быстро*	*Нет*	*Нет (эксклюзивный доступ)*
Мьютекс	*Медленно*	*Да*	*Нет (эксклюзивный доступ)*
Семафор	*Медленно*	*Да*	*Автоматически*
Событие	*Медленно*	*Да*	*Да*

Методы синхронизации (семафоры, критические секции и исключения) впервые были предложены голландским профессором математиком Эдсжер Дейкстра (**Edsger Wybe Dijkstra**) в начале 1970-х гг. (1965–1973).

* **Microsoft Developer Network** — подразделение компании Майкрософт, ответственное за взаимодействие фирмы с разработчиками

Рассматривая ситуации с многопользовательским доступом, он ввел понятие критический интервал (**critical section**). Под критическим интервалом подразумевалась всякая последовательность шагов обработки, последовательный процесс которой не должен прерываться никаким другим процессом. Он сопоставил каждому общему для некоторых процессов набору данных некоторую запирающую переменную (**mutex**), начальное значение которой равно 1. Если к началу критического интервала значение этой переменной больше или равно 1, то интервал должен выполняться, иначе ждать.

Пример использования средств синхронизации для решения проблемы блокировки сокета

При пошаговом выполнении практически любой программы, использующей интерфейс сокетов, можно заметить, что некоторые `Winsock`-функции ожидают завершения выполняемых ими операций. Особенно надолго «подвисает» функция **`recv`**. Другими словами, выход из функции не происходит до момента завершения текущей операции. Чтобы не дать приложению полностью застыть из-за отсутствия ожидаемых данных, можно предложить такое

решение данной проблемы: разделить приложение на два потока: читающий и обрабатывающий данные, разделяющие буфер данных. Доступ к общему буферу осуществляется объектом синхронизации как событие (event) или мьютекс (mutex). Задача читающего потока состоит в считывании поступающих данных из сети на сокет в общий буфер. Когда читающий поток считал минимально необходимое количество данных, предназначенных для обрабатывающего потока, он переключает событие в несигнальное состояние, таким образом сообщая обрабатывающему потоку о наличии в общем буфере данных для обработки. Обрабатывающий поток в свою очередь забирает из буфера данные и обрабатывает их.

Для иллюстрации данного метода во фрагменте кода представлены две функции: ReadingThread обеспечивает чтение данных из сети, ProcessingThread – обработку данных.

Код:

```
    #define MAX_BUFFER_SIZE 4096
// Инициализация critical section
    CRITICAL_SECTION data;
/* и события с автосбросом перед инициализацией потоков */
    HANDLE  hEvent;
    SOCKET  sock;
    CHARbuffer[MAX_BUFFER_SIZE]; // общий буфер
// создание читающего сокета
    DWORD WINAPI ReadingThread(void)
    {   int nTotal = 0,
        nRead = 0,
        nLeft = 0,
        nBytes = 0;
        while (true)
        {   nTotal = 0;
            nLeft = NUM_BYTES_REQUIRED;
/* получение минимального количества байтов для обработки */
```

```
            while (nTotal < NUM_BYTES_REQUIRED)
            { EnterCriticalSection(&data);
            nRead = recv(sock,
    &(buffer[MAX_BUFFER_SIZE-nBytes]), nLeft, 0);
                if (nRead == -1)
            { cout<<"errorn"<<endl;
            ExitThread();}
                nTotal += nRead;
                nLeft -= nRead;
                nBytes += nRead;
                LeaveCriticalSection(&data);
            }
            SetEvent(hEvent);
        } }
// обрабатывающий поток
    DWORD WINAPI ProcessingThread(void)
    { while (true)
    { // ждем данных
        WaitForSingleObject(hEvent);
        EnterCriticalSection(&data);
// обработка части поступивших данных
        DoSomeComputationOnData(buffer);
// удаление из буфера обработанных данных
            nBytes -= NUM_BYTES_REQUIRED;
        LeaveCriticalSection(&data);
        }  }
```

Данное приложение составлено таким образом, чтобы иметь на каждый сокет читающий и обрабатывающий потоки. Очевидно, что такого вида приложение плохо масштабируется при большом количестве сокетов.

9. ПРОЦЕСС-СЕРВЕР

Серверы бывают последовательной и параллельной обработки. Процесс-сервер, обрабатывающий каждый запрос клиента индивидуально, является последовательным. Последовательный сервер обрабатывает запросы поочередно, выстраивая их в очередь, если необходимо. Чтобы быть эффективным, такой сервер должен затрачивать ограниченное и заранее предсказуемое время на обработку поступающих запросов. Если сервер должен одновременно обработать несколько поступивших запросов, когда заранее неизвестно, сколько времени будет затрачено на обработку каждого, он конструируется параллельным. Параллельный сервер создает отдельный процесс (или поток, если это позволяет операционная система) для обработки каждого запроса.

Более подробно рассмотрим ситуацию, когда появляется очередной запрос, а сервер еще не закончил обработку предыдущего. То есть, когда программа-клиент пытается послать еще один запрос, в то время как последовательный сервер еще не закончил обработку предыдущего, или когда клиент посылает запрос до того момента, как параллельный сервер закончил создание нового процесса – обработчика предыдущего запроса.

Как только на сокете, обслуживаемом функцией *accept*, появляется запрос, функция возвращает серверу дескриптор только что созданного нового сокета. Что сервер будет делать с этим дескриптором, зависит от реализации сервера. Сервер может обрабатывать запросы параллельно или последовательно. Предположим, что вы создали последовательный сервер. Следовательно, он будет последовательно обрабатывать, а затем закрывать все переданные ему функцией *accept* дескрипторы сокетов. По окончании обработки очередного запроса последовательный сервер вновь вызывает *accept*, а она возвращает ему дескриптор сокета для следующего запроса, если он имеется, и т. д. Если до этого вызывалась функция *listen*, запрос может быть выбран из входной очереди, если нет – сервер прослушивает сетевые запросы напрямую через сокет.

Теперь предположим, что вы написали сервер параллельной обработки. После того как выполнится функция *accept*,

параллельный сервер создаст новый (дочерний) процесс и передаст ему задачу по обслуживанию нового запроса. Каким образом создается дочерний процесс, зависит от операционной системы. Независимо от того, как создается дочерний процесс, процесс-родитель передает ему копию нового сокета. Далее родительский процесс закрывает собственную копию сокета и вновь вызывает функцию *accept*.

Можно видеть, что и последовательный и параллельный серверы действуют практически одинаково, обслуживая сетевые запросы. Между ними существует лишь одно различие: параллельный сервер не ждет, пока закончится обработка запроса, а вызывает функцию *accept* сразу же. Последовательный сервер не может вызвать *accept* до тех пор, пока не обработает запрос. Программа-сервер с параллельной обработкой запросов создает дочерний процесс для обработки поступающих запросов и сразу после этого вызывает функцию *accept*. За исключением случаев, когда обработка запроса почти не занимает времени, сервер успевает вызвать *accept* еще до того, как дочерний процесс ее закончит. Другими словами, сервер параллельной обработки запросов способен отвечать на множество запросов одновременно.

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

На рис. 27 изображена ситуация, когда два клиента обращаются к серверу параллельной обработки запросов. Конструкция сервера параллельной обработки предполагает, что для каждого нового запроса от клиента создается новая копия процесса-сервера. Если сервер параллельной обработки реализовать как многопоточный, его производительность значительно увеличивается. Вместо создания нового процесса или их переключения, многопоточный сервер параллельной обработки просто образует новый поток, выполняющийся гораздо быстрее.

в клиентов паралле

Если сетевой компьютер оборудован несколькими процессорами, сервер будет работать еще быстрее, поскольку каждый поток сможет выполняться на отдельном процессоре.

9.1. Принципы проектирования параллельного сервера

Поясним, каким образом параллельный сервер одновременно выполняет запросы, поступившие на один и тот же порт протокола.

Ранее указывалось, что для сетевого соединения сокету необходимо иметь две конечные точки. Конечные точки сетевого соединения определяют адреса взаимодействующих процессов. До тех пор, пока сетевые процессы соединены с различными точками сетевого соединения, они могут работать на одном и том же порту протокола, не мешая друг другу. Сервер параллельной обработки создает новый процесс для каждого соединения. Другими словами, на сетевом компьютере постоянно работает одиночный главный сервер, ожидая запроса от любого процесса в сети. В другой момент времени на том же компьютере по-прежнему работает главный сервер, а вместе с ним — множество подчиненных. Каждый подчиненный сервер работает с уникальным адресом конкретного, соединенного с ним процесса.

Например, сегмент TCP идентифицируется своим уникальным адресом. Когда сервер принимает сегмент TCP, он отправляет его на сокет, связанный именно с адресом сегмента. Если такого сокета не существует (что означает поступление запроса на новую сетевую службу), сервер передаст сегмент сокету, соединенному с любым адресом (`ANY_BODY address`). Мы помним, что сокет, соединенный с любым адресом, принимает запросы на соединения, поступающие с любого сетевого адреса. Начиная с этого момента весь процесс повторяется. Другими словами, сокет главного процесса-сервера (прослушивающий запросы с любого сетевого адреса) порождает дочерний процесс-сервер, который в свою очередь и обрабатывает запрос.

Необходимо понимать, что сокет, прослушивающий любой сетевой адрес, не может иметь открытого соединения. Мы помним, что для установления сетевого соединения необходимо иметь

пару конечных точек, каждая из которых должна обладать определенным адресом. Поскольку сокет главного процесса-сервера всегда прослушивает запросы с любого адреса, он в состоянии обработать только вновь поступивший запрос.

9.2. Пример параллельного многопоточного TCP-сервера

Сервер повторяет переданную клиентом строку. Клиент разрывает соединение с сервером после передачи строки определенного содержания, например строки `"quit".` Для работы с очередным клиентом сервер создает отдельный поток, который функционирует независимо от других потоков. Таким образом осуществляется параллельное обслуживание сервером нескольких клиентов.

Код сервера:

```
    #include <iostream>
    #include <winsock2.h>
    #include <string>
    #pragma comment (lib,"Ws2_32.lib")
    using namespace std;
    u_short MY_PORT =666;
// порт, который слушает сервер
/* макрос для печати количества активных пользова-
телей */
    #define PRINTNUSERS if (nclients)\
    cout << " user on-line " << nclients <<endl;\
    else cout << "No User on line\n";
/* прототип функции, обслуживающий  подключившихся
пользователей */
    DWORD WINAPI ConToClient(LPVOID client_socket);
/* глобальная переменная – количество активных
пользователей */
    int nclients = 0;
    int main( )     {
    char buff[1024]; // Буфер для различных нужд
```

```
    cout << "TCP ECHO-SERVER \n";
// Инициализация библиотеки сокетов
    if (WSAStartup(0x0202,(WSADATA *) &buff[0]))
        {               // Ошибка!
cout << "Error WSAStartup \n" <<WSAGetLastError();
        return -1;
        }
// создание сокета
        SOCKET mysocket;
if ((mysocket=socket(AF_INET,SOCK_STREAM,0))<0)
        {               // Ошибка!
    cout << "Error socket \n" << WSAGetLastError();
WSACleanup();// Деинициализация библиотеки Winsock
        return -1;
        }
// связывание сокета с локальным адресом
        sockaddr_in local_addr;
        local_addr.sin_family=AF_INET;
        local_addr.sin_port=htons(MY_PORT);
        local_addr.sin_addr.s_addr=0;
if (bind(mysocket,(sockaddr *) &local_addr,
sizeof(local_addr)))
    {               // Ошибка
    cout <<"Error bind \n" << WSAGetLastError();
    closesocket(mysocket);  // закрываем сокет!
    WSACleanup();
    return -1;
    }
// ожидание подключений
// размер очереди – 0x100
    if (listen(mysocket, 0x100))
    {               // Ошибка
    cout<<"Error listen \n"<<WSAGetLastError();
    closesocket(mysocket);
    WSACleanup();
    return -1;
```

```
    }
    cout << "Waiting connections\n";
    SOCKET client_socket;   // сокет для клиента
    sockaddr_in client_addr;    // адрес клиента
    int client_addr_size=sizeof(client_addr);

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

    HOSTENT *hst;  // пытаемся получить имя хоста
    hst=gethostbyaddr((char *) &cli-
ent_addr.sin_addr.s_addr,4, AF_INET);
    cout << "+new connect!\n" ;
// вывод сведений о клиенте
    if(hst) cout<< hst->h_name ;
// символьное имя клиента
    else cout<< "";
cout <<inet_ntoa(client_addr.sin_addr); // IP-адрес
    PRINTNUSERS
    cout<<endl;
// создание нового потока для обслуживания клиента
    DWORD thID;
    CreateThread(NULL,NULL,ConToClient, &cli-
ent_socket,NULL,&thID);
    }
    return 0;
    }

/* Эта функция создается для отдельного потока и
обсуживает очередного подключившегося клиента неза-
висимо от остальных */
```

```
    DWORD WINAPI ConToClient(LPVOID client_socket)
    {
    SOCKET my_sock;
    int len;
    my_sock=((SOCKET *)client_socket)[0];char
buff[1024];
    char sHELLO[] ="Hello, Student \r\n";
// отправляем клиенту приветствие
    send(my_sock,sHELLO,sizeof(sHELLO),0);
// цикл эхо-сервера
    while (SOCKET_ERROR !=
(len = recv (my_sock, (char *) & buff[0], 1024, 0))
    {
// получение сообщение от клиента
    buff[len]='\0';
    cout<<“received:"<< buff << endl;
// отправка его обратно клиенту
    send(my_sock,buff,len,0);
    }
/* произошел выход из цикла, соединение с клиентом
разорвано  */
nclients--; // уменьшаем счетчик активных клиентов
    cout << "-disconnect\n";
    PRINTNUSERS
    closesocket(my_sock);       // закрываем сокет
    return 0;
    }
```

10. ПРОЕКТИРОВАНИЕ СЕТЕВЫХ СЛУЖБ

По своей архитектуре сетевые службы являются клиент-серверными системами и могут быть представлены клиентской и серверной частями либо только одной из них. Сетевая служба предоставляет пользователям сети некоторый набор услуг, который называют также *сетевым сервисом* (service). На практике обычно под *службой* понимают сетевой компонент, который реализует предоставляемый службой набор услуг, а под *сервисом* – описание этого набора услуг, т. е. *сервис* – это интерфейс между потребителем услуг и поставщиком услуг (*службой*).

В современных сетях набор сервисов очень разнообразный и широкий. В гл. 2 были рассмотрены стандартные системные службы, как правило, представляющие собой программную реализацию протоколов прикладного уровня TCP/IP. Помимо служб, решающих задачи сетевого администрирования, существуют службы, ориентированные на конечных пользователей. В поставку современных операционных систем, как правило, входит текстовый файл services (/etc/services), содержащий информацию обо всех зарегистрированных в IANA (Internet Assigned Numbers Authority) службах Internet, назначенных им номерах портов и типах сетевых протоколов.

Именно многообразие предлагаемых сервисов позволяет практически каждому человеку, независимо от возраста, вкусовых предпочтений и других критериев, найти в сети для себя площадки, которые помогут ему учиться, общаться с товарищами по интересам и работать. Однако, как утверждал Кузьма Прутков, «нельзя объять необъятное». И современный пользователь вновь ищет в сети средства для удовлетворения своих новых потребностей. Необходимость в разработке продуктов указанного назначения будет еще долгое время мотивировать сетевое программирование.

10.1. ЧАТ-СЕРВИС

Раньше чатами называли сервис для групповых общений в Интернете. Характерной особенностью его является

коммуникация именно в реальном времени или близкая к этому, что отличает чат от форумов и других «медленных» средств. После отправки сообщения собеседник сразу его получает… Сообщения видны не только ему, но и другим собеседникам. До появления социальных сетей, Skype, Viber и других программ для быстрой связи, они были очень востребованы. Потому что только здесь можно было переписываться в реальном времени. Люди договаривались встретиться в каком-то чате, чтобы просто пообщаться.

В настоящее время самостоятельный программный продукт, реализующий чат-сервис, практически не найдешь. Однако его можно встретить в любом мессенджере (Viber, Telegram, WhatsApp) как составную часть функционала, позволяющего переписываться в реальном времени. Но само слово осталось и периодически встречается в Интернете, СМИ и личных беседах.

Подводя итог сказанному, можно дать следующее определение для указанного вида сетевой коммуникации:

Чат (chat) – средство обмена сообщениями по компьютерной сети в режиме реального времени (online) с другими пользователями, а также программное обеспечение, позволяющее организовывать такое общение.

В настоящей главе на основе рассмотренных ранее в работе сетевых технологий представляется вариант распределенной программы на C++, реализующей чат-сервис. Данный вариант демонстрирует «***публичный***», или «широковещательный», чат, т. е. сообщения, передаваемые пользователем в чат, сразу отправляются всем участникам чата.

Как и большинство сетевых проектов, программа имеет «клиент-серверную» архитектуру. Клиентский модуль реализует интерфейсные функции, а именно: передает сообщение в чат или принимает новые сообщения, поступившие в чат от других пользователей. Серверный модуль имеет следующий функционал: принимает от очередного пользователя сообщение и рассылает его обратно всем участникам чата. Исходный код проекта снабжен для наглядности многочисленными комментариями.

Для удобства тестирования в предлагаемом варианте использован интерфейс «обратной петли». Для применения в реальной сети необходимо указать конкретный адрес сервера.

Код сервера:

```
    // server_chat
    #define _WINSOCK_DEPRECATED_NO_WARNINGS
    #include <winsock2.h>
    #include <windows.h>
    #include <string>
    #include <iostream>
    #pragma comment(lib, "ws2_32.lib")
    #pragma warning(disable: 4996)
    using namespace std;

    SOCKET Connections[100];
//массив сокетов активных клиентов
int Counter = 0;        // число активных клиентов
enum Packet {Pack,Test};// допустимые типы пакетов

// функционал потока отдельного клиента
    DWORD WINAPI ServerThread( LPVOID number)
     {
    Packet packettype;
// определение номера сокета клиента
    int index = ((int *)number)[0];
    cout << "socket number="<< index<< endl;
    SOCKET Con=Connections[index];
// общение с клиентом:
    while (true)
    {
//получение информации от клиента
recv(Con,(char*) &packettype, sizeof(Packet),NULL);
// определение типа полученного пакета
    if (packettype == Pack)
```

```
        {
        cout << "received packet PACK"<< endl;
        int msg_size;
// определение объема пакета
    recv(Con,(char*) &msg_size, sizeof(int),NULL);
/* резервирование буфера нужного размера для принятия
пакета */
        char* msg = new char[msg_size + 1];
        msg[msg_size] = 0;
// получение пакета
        recv(Con, msg, msg_size, NULL);
        cout << msg<<endl;
/* передача полученного сообщения другим участникам
чата */
            for (int i = 0; i < Counter; i++)
            {
            if (i == index) continue;
            Packet msgtype = Pack;
/* передача типа, объема и содержания информационного
пакета */
send(Connections[i],(char*)&msgtype,sizeof(Packet),
NULL);
send(Connections[i],(char*)&msg_size,sizeof(int),
NULL);
send(Connections[i], msg, msg_size, NULL);
            }
            delete[] msg;
        }
        else
        {
// получен неопознанный пакет!
cout<< "Unrecognized packet: "<<packettype << endl;
break;
        }
    }
    closesocket(Con);
```

```
    return 0;
    }

    int main() {
// Инициализация Winsock
    WSAData wsaData;
    WORD DLLVersion = MAKEWORD(2, 1);
    if (WSAStartup(DLLVersion, &wsaData) != 0)
    {
    cout << "Error" << endl;
    exit(1);
    }
/* сохранение в слушающем сокете информации о сервере
*/
    sockaddr_in addr;
    int sizeofaddr = sizeof(addr);
    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    addr.sin_port = htons(123);
    addr.sin_family = AF_INET;
SOCKET sListen = socket(AF_INET,SOCK_STREAM,NULL);
    bind(sListen, (SOCKADDR*)& addr, sizeof(addr));
// режим прослушивания, организация очереди
    listen(sListen, 10);
    SOCKET newConnection;
    for (int i = 0; i < 10; i++)
    {
// извлечение запросов из очереди
    newConnection=accept(sListen,(SOCKADDR*)&addr,
    &sizeofaddr);
    if (newConnection == 0)
    {cout << "Error #2\n";} //ошибка подключения
    else
    {
//приветствие нового подключившегося клиента
    cout << "Client Connected!\n";
    string msg = “Welcome to chat!";
```

```
    int msg_size = msg.size();
//передача клиенту пакета типа Pack
    Packet msgtype = Pack;
send(newConnection,(char*)&msgtype, sizeof(Packet),
NULL);
send(newConnection, (char*)&msg_size, sizeof(int),
NULL);
send(newConnection, (char*)&msg[0], msg_size, NULL);
/* сохранение сокета клиента в массиве участников
чата */
    Connections[i] = newConnection;
    Counter++;
//количество участников увеличивается
    cout << "count="<< Counter<< endl;

// создание нового потока для обслуживания клиента
CreateThread(NULL,NULL,ServerThread,&i,NULL,NULL);
    Sleep (1000);
// передача клиенту сигнального сообщения
    Packet testpacket = Test;
    cout << "type packet=TEST"<<endl;
send(newConnection,(char*)&testpacket,
sizeof(Packet), NULL);
    }
    }
    system("pause");
    return 0;
    }
```

Сервер демонстрирует параллельную работу с клиентами: для обслуживания каждого клиента создается отдельный поток. Во внутреннем потоке сервер приветствует подключившегося к чату клиента и в бесконечном цикле общается с ним: получает от клиента сообщение и пересылает его другим клиентам.

Для обмена информацией участниками чата в перечислимом типе `Packet` определены допустимые типы передаваемых

пакетов: сигнальный пакет (Test) используется сервером для контроля связи с клиентом; информационный пакет (Pack) – для передачи сообщения. Для информационного пакета необходимо указать тип, объем, а затем содержимое пакета. При получении пакета неопределенного типа связь разрывается.

Код клиента:

```
    // CLIENT_CHAT
    //#define _WINSOCK_DEPRECATED_NO_WARNINGS
    #include <winsock2.h>
    #include <windows.h>
    #include <string>
    #include <iostream>
    #pragma comment(lib, "ws2_32.lib")
    // #pragma warning(disable: 4996)
    using namespace std;

    SOCKET Connection;
    enum Packet {Pack, Test }; // типы пакетов
/* дополнительный поток клиента для отправки сообще-
ний */
    DWORD WINAPI ClientThread(LPVOID sock)
    {
    SOCKET Con=((SOCKET*)sock)[0];
    Packet packettype;
// получение пакетов от сервера
    while (1)
    {
recv(Con,(char*)&packettype, sizeof(Packet),NULL);
// прием типа пакета
    if (packettype == Pack)
    {
    cout << "type packet=PACK"<<endl;
// определение объема информационного пакета
    int msg_size;
```

```
    recv(Con,(char*)& msg_size, sizeof(int), NULL);
    char* msg = new char[msg_size + 1];
    msg[msg_size] = '\0';
// прием данных пакета
    recv(Con, msg, msg_size, NULL);
    cout << msg << endl;
    delete[] msg;
    }
    else
    {if (packettype == Test)
// прием сигнального пакета от сервера
    cout << "Test packet.\n";
        else
        {
// тип пакета не определен
cout<<"Unrecognized packet: "<<packettype<< endl;
    break;
    }
    }
    }
    closesocket(Connection);
        return 0;
    }
    int main()      // основной поток
    {
    WSAData wsaData;
    WORD Ver = MAKEWORD(2, 1);
    if (WSAStartup(Ver, &wsaData) != 0)
    {
    cout << "Error" << endl;
    exit(1);
    }
// определение параметров сервера
    sockaddr_in addr;
    int sizeofaddr = sizeof(addr);
    addr.sin_addr.s_addr=inet_addr("127.0.0.1");
```

```
// loopback!
    addr.sin_port = htons(123);
    addr.sin_family = AF_INET;
Connection = socket(AF_INET, SOCK_STREAM, NULL);
// соединение с сервером
if(connect(Connection,(SOCKADDR*)&addr,
sizeof(addr))!= 0)
    {
    cout << "Error: failed connect to server.\n";
    return 1;
    }
    cout << "Connected!\n";
/* создание дочернего потока для приема сообщений от
сервера */
CreateThread(NULL, NULL, ClientThread, &Connection,
NULL, NULL);

// отправка данных серверу в основном потоке
    string msg1;
    while (true) {
    getline(cin, msg1);
    int msg_size = msg1.size();
// передача информационного пакета
    Packet packettype = Pack;

send(Connection,(char*)&packettype, sizeof(Packet),
NULL);
send(Connection,   (char*)&msg_size,   sizeof(int),
NULL);
send(Connection, (char*)&msg1[0], msg_size, NULL);

    Sleep(1000);
    }
    system("pause");
    return 0;
    }
```

Процессы отправки и приема сообщений клиентом в общем случае являются асинхронными. Клиент может отправить `n` сообщений, а получить от сервера `m` сообщений. Причем заранее количества `n` и `m` неизвестны, они могут быть и нулевыми. Их значения определяются в ходе функционирования чата. Именно поэтому в клиентском модуле также использован механизм многопоточности. Отправка сообщений серверу выполняется в основном потоке (`main`), а прием сообщений происходит в дочернем потоке. Эти потоки работают параллельно, каждый в своем ритме.

Чат-сервис можно реализовать и в «***приватном***» режиме. Этот вариант более сложный, но и более интересный. В этом случае сообщение, передаваемое в чат, не обязательно должно быть адресовано всему сообществу, но возможно и только конкретному пользователю. При таком виде коммуникации, как правило, каждый участник беседы должен быть наделен уникальным именем-ником. Обеспечение уникальности имен – обычно задача сервера. При передаче серверу приватного сообщения указывается и ник адресата. Ники также можно использовать для указания факта присоединения пользователя к беседе или ухода из нее. Как правило, предложенная разработчиками ненавязчивая комфортная обстановка повышает привлекательность сервиса и как результат увеличивает число его почитателей.

11.СЕТЕВОЙ ПРАКТИКУМ

Для выполнения практических заданий необходимо иметь несколько объединенных TCP/IP-сетью компьютеров с операционной системой Windows x.

Кроме того, для разработки приложений на языке C++ требуется среда разработки Microsoft Visual Studio.

Приложение в проекте разрабатывается как распределенное клиент-серверное.

Каждая практическая работа состоит из нескольких заданий. Задания, как правило, связаны между собой, требуют последовательного выполнения и указаны в порядке повышения сложности.

Практическая работа считается выполненной, если успешно выполнены все ее задания.

Проект №1. *Клиент-серверное приложение. Общая среда – файл*

Сценарий 1 (один клиент, один сервер)

Приложение составить из двух программ: клиент и сервер. Для взаимодействия этих программ использовать 2 файла. Файл **f1**заполняется клиентом (клиент пишет в него запросы серверу), сервер считывает из него очередной запрос клиента. Файл **f2** заполняется сервером (сервер пишет в него свой ответ на запрос клиента), клиент считывает из него ответ на свой запрос.

Взаимодействие клиента и сервера происходит по следующему алгоритму:

Клиент дописывает свой запрос в конец файла **f1**. Сервер в бесконечном цикле проверяет, появились ли в файле **f1** новые запросы. Для этого он сравнивает предыдущий размер файла с текущим (в начале работы предыдущий и текущий размеры файлов равны 0!).

Если эти размеры совпали – нет новых запросов. Иначе – появился новый запрос, в этом случае сервер считывает его из файла **f1**, обрабатывает и результат записывает в конец файла **f2**.

Клиент аналогично постоянно проверяет файл **f2** на наличие новых ответов от сервера.

Реализовать данный сценарий для случаев:

– файлы текстовые;
– файлы бинарные (передача структур);

Сценарий 2 (для большого количества клиентов)

Приложение состоит из двух отдельных модулей (программ на C++): клиент и сервер.

Для общения используются файл ***con*** (создается сервером, используется сервером и всеми клиентами) и для каждого клиента файл, имя которого совпадает с именем клиента (каждый такой файл создается клиентом, используется сервером и клиентом, имя которого совпадает с именем файла). Итого: количество файлов равно 1 + m, где m – количество активных клиентов. Получив от клиента его имя, сервер определяет файл для общения с этим клиентом.

Клиент и сервер заранее согласовывают тип файлов и формат сообщений.

Возможные варианты предметных областей для разрабатываемого проекта:

1. Сервер – медицинский центр. Клиент передает серверу фамилию студента, его **рост** и **вес**. Сервер на основании этих данных выдает результат о нормальности развития студента (**нормально, превышение веса, нехватка веса**).
2. Сервер – деканат. Клиент передает серверу **фамилию** студента и **четыре оценки** по экзаменам сессии. Сервер на основании этих данных выдает результат, есть ли у студента задолженности; если нет, то возможно ли получение стипендии и ее размер.

Проект №2. *Интерфейс сокетов. TCP и UDP сокеты*

Целью данного проекта является выявление особенностей и реализация сетевых взаимодействий двух видов: потоковое соединение и дейтаграммное. При разработке конструкций приложений

использование параллельного программирования не предусматривается. Особое внимание уделить рассмотрению общения сервера с клиентами в каждом случае взаимодействия.

1. Реализовать модельные сетевые приложения:

а) диалоговое общение на основе TCP-сокетов;

б) эхо-сервер на основе UDP-сокетов.

Объяснить использование необходимых функций интерфейса WinSocket.

2. Предусмотреть:

а) использование в приложениях произвольного количества клиентов (1 или более);

б) распределенный характер приложения, т. е. выполнение клиентов и сервера на различных компьютерах сети, указать необходимые изменения в кодах приложений.

3. Изменить код приложений таким образом, чтобы возможно было передавать от клиента к серверу и обратно данные произвольных типов (структуру). Можно реализовать сценарий, используемый в проекте 1 при создании клиент-серверных приложений на основе бинарных файлов.

Проект №3. ***HTTP-взаимодействия***

Для выполнения проекта необходимо изучить основные положения протокола HTTP (форматы запросов клиента и ответа сервера) и выполнить следующие действия:

1. Написать на C++ клиент серверное приложение, осуществляющее взаимодействие по протоколу HTTP на основе TCP-сокетов;
2. Используя HTTP-клиент, написанный на C++, обратиться методом GET к одному из стандартных Web-серверов (например, `json.org` или `library.ru`).
3. Используя стандартный браузер, обратиться к HTTP-серверу, написанному на C++.
4. С помощью программы telnet отправить HTTP-запрос к веб-серверу. Запрос к HTTP-серверу вначале записать в текстовый файл. Для соединения с веб-сервером с использованием telnet нужно выполнить команду:

```
telnet <ip-адрес-веб-сервера> 80
```

и далее через буфер обмена мышью скопировать запрос из текстового файла, при необходимости нажать Enter два раза. В новых версиях Windows по умолчанию telnet не устанавливается и его нужно установить отдельно через установщик компонентов Windows.

Результаты работы представить в виде скринов.

Проект №4. *Многопоточность. Средства синхронизации*

Организовать параллельную работу сервера с клиентами на основе TCP-соединения.

1. Создание многопоточного эхо-сервера.

2. Передача структур данных между клиентом и сервером.

По возможности реализовать распределенное клиент-серверное приложение (на разных компьютерах).

Коды клиента и сервера представить в виде файлов *.cpp.

3. Изучить механизмы синхронизации данных, разделяемых потоками (критические секции, мьютексы, события, семафоры).

Для каждого механизма написать программу, иллюстрирующую синхронизацию данных.

Работоспособность программ подтвердить при помощи снимков экрана (скриншоты).

Проект №5. *Создание многопоточного клиент-серверного сетевого приложения чат(chat) с использованием технологии WinSocket*

Разработать клиент-серверное приложение для реализации сетевого сервиса. Для реализации многопользовательского режима использовать элементы параллельного программирования. В качестве модельной задачи можно рассмотреть варианты чат-сервиса:

1. ***Создание публичного чата*** (переданная в чат информация мгновенно передается всем входящим в чат участникам).

2. Разработать ***возможность указания ника*** (прозвища) для участников чата. При подключении участника сообщать всем остальным членам чата, что подключился участник с определенным ником. При уходе участника также сообщать о его уходе с указанием его ника. При отправке сервером сообщения клиентам отправлять и ник участника, который это сообщение выложил в чат. Для этого, возможно, придется расширить количество значений перечислимого типа PACKET, задающего тип пакета, и разработать формат пакета для каждого типа.

3. Включить в приложение чат ***возможность приватного общения***, когда при передаче сообщения участник указывает серверу ник участника, которому это сообщение нужно передать. Остальные участники, естественно, данное сообщение не получают.

Проверить работу приложения, используя интерфейс обратной петли, а также в сетевом варианте, запустив процессы сервера и клиентов на различных компьютерах сети.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Букатов А.А., Гуда С.А. Компьютерные сети. Расширенный начальный курс. СПб., 2019.
2. Васильев А. Программирование на C++ в примерах и задачах. М., 2017.
3. Власов Ю. В., Рицкова Т.И. Администрирование сетей на платформе MS Windows Server: учеб. пособие. М., 2010.
4. Гельбух С. Сети ЭВМ и телекоммуникации. Архитектура и организация: учеб. пособие. СПб., 2019.
5. Дубовик С. C++. Сетевое программирование. URL: https:// www.vr-online.ru/content/s-setevoe-programmirovanie-chast-1-3999
6. Каев А. Программирование сети в VC++. URL: https:// http://www.firststeps.ru/mfc/net/socket/
7. Касперски К. Секреты поваров компьютерной кухни или ПК: решение проблем. СПб., 2003.
8. Кенин А., Колисниченко Д. Самоучитель системного администратора. 5-е изд. СПб., 2019.
9. Кумар В., Кровчик Э., Лагари Н. Net. Сетевое программирование. М., 2014.
10. Мэрфи Н., Мэлоун Д. IPv6. Администрирование сетей. СПб., 2007.
11. Неволин А.О. Базовые принципы сетевого взаимодействия: учеб. пособие для вузов. М., 2020.
12. Ногл М. Иллюстрированный учебник. М., 2001.
13. Оланд Д., Джонс Э. Программирование в сетях Windows. СПб., 2002.
14. Олифер В.Г., Олифер Н.А. Компьютерные сети. Принципы, технологии, протоколы. СПб., 2020.
15. Побегайло А. П. Системное программирование в Windows. СПб., 2005.
16. Снейдер Й. Эффективное программирование TCP/IP. Создание сетевых приложений. М., 2019.
17. Стивен П. Язык программирования C++. Лекции и упражнения. М., 2018.

18. Стивенс У. Р. Протоколы TCP/IP: практическое руководство. СПб., 2003.

19. Таненбаум Э., Уэзеролл Д. Компьютерные сети. СПб., 2019.

20. Шмидт Д., Хьюстон С. Программирование сетевых приложений на C++. М., 2010.

21. Winsock для всех. Клуб программистов «Весельчак.У». URL: https://club.shelek.ru/

22. Russell J. Winsock (на англ. яз.). М., 2012.

23. William T. Block Introduction to Multi-threaded Code, 26 Mar 2000. URL: https://www.codeproject.com/Articles/438/

ОГЛАВЛЕНИЕ