МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Интеллектуальные информационные технологии»

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ НА БАЗЕ ТРАНСПОРТНЫХ ПРОТОКОЛОВ TCP, UDP С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ WINSOCK

Методические указания

по выполнению лабораторных работ по курсу
«АППАРАТНОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СЕТЕЙ»
для студентов специальностей 1-53 01 02, 1-40 03 01

Брест 2009

УДК 681.3

Методические указания предназначены для выполнения лабораторно-практических работ по дисциплине "Аппаратное и программное обеспечение сетей" и содержат описания цикла лабораторных работ по изучению базовых характеристик транспортных протоколов TCP, UDP и подходов к программированию на их базе сетевых приложений с использованием средств WINSOCK. Задания, требования к выполнению работ и содержанию отчетов разработаны с учетом имеющейся на кафедре «Интеллектуальные информационные технологии» аппаратно-технической базы.

Методические указания предназначены для использования студентами специальностей 1-53 01 02 «Автоматизированные системы обработки информации», 1-40 03 01 «Искусственный интеллект». Рекомендованы к изданию кафедрой «Интеллектуальные информационные технологии» 06.05.2009, протокол №7.

Ил. 6., список лит. – 6 назв.

Составители: Савицкий Ю. В., к.т.н., доцент

Муравьев Г.Л., к.т.н., доцент

Содержание

1 БАЗОВЫЕ ХАРАКТЕРИСТИКИ ТРАНСПОРТНЫХ ПРОТОКОЛОВ TCP, UDP 4
1.1 БАЗОВАЯ ПЕРЕДАЧА ДАННЫХ 4
1.2 ОБЕСПЕЧЕНИЕ ДОСТОВЕРНОСТИ ПЕРЕДАЧИ 4
1.3 АДРЕСАЦИЯ 5
1.4 УПРАВЛЕНИЕ СОЕДИНЕНИЯМИ 5
1.5. УПРАВЛЕНИЕ ПОТОКОМ 6
1.6 ЗАГОЛОВОК TCP-СЕГМЕНТА 8
1.7 ПРОМЕЖУТОЧНЫЕ СОСТОЯНИЯ СОЕДИНЕНИЯ 10
1.8 ПРОТОКОЛ UDP 12
2 ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ НА БАЗЕ WINSOCK 13
2.1 ПОНЯТИЕ СОКЕТОВ 13
2.2 РАБОТА С БИБЛИОТЕКОЙ WINSOCK2.H 14
2.3 ДЕРЕВО ВЫЗОВОВ 19
2.4 АДРЕСА И ПРАВИЛА ИХ ИСПОЛЬЗОВАНИЯ 20
2.5 СЕТЕВОЙ ПОРЯДОК БАЙТ 22
2.6 ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ 23
3 ЛАБОРАТОРНАЯ РАБОТА №1. ОРГАНИЗАЦИЯ TCP-СЕРВЕРА 24
4 ЛАБОРАТОРНАЯ РАБОТА №2. ОРГАНИЗАЦИЯ TCP-КЛИЕНТА 25
5 ЛАБОРАТОРНАЯ РАБОТА №3. ОРГАНИЗАЦИЯ UDP-СЕРВЕРА И UDP-КЛИЕНТА 25
6 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 26

1 Базовые характеристики транспортных протоколов TCP, UDP

Протокол TCP (Transmission Control Protocol) обеспечивает сквозную доставку данных между прикладными процессами, запущенными на узлах, взаимодействующих по сети. Стандартное описание TCP содержится в RFC-793.

TCP – *надежный* байт-ориентированный (byte-stream) протокол *с установлением соединения*. TCP находится на транспортном уровне стека TCP/IP. Протокол IP занимается пересылкой дейтаграмм по сети, никак не гарантируя доставку, целостность, порядок прибытия информации и готовность получателя к приему данных; все эти задачи возложены на протокол TCP.

При получении дейтаграммы, в поле Protocol которой указан код протокола TCP (6), модуль IP передает данные этой дейтаграммы модулю TCP. Эти данные представляют собой TCP-сегмент, содержащий TCP-заголовок и данные пользователя (прикладного процесса). Модуль TCP анализирует служебную информацию заголовка, определяет, какому именно процессу предназначены данные пользователя, проверяет целостность и порядок прихода данных и подтверждает их прием другой стороне. По мере получения правильной последовательности неискаженных данных пользователя они передаются прикладному процессу.

1.1 Базовая передача данных

Модуль TCP выполняет передачу непрерывных потоков данных между своими клиентами в обоих направлениях. Клиентами TCP являются прикладные процессы, вызывающие модуль TCP при необходимости получить или отправить данные процессу-клиенту на другом узле. Протокол TCP рассматривает данные клиента как непрерывный неинтерпретируемый поток октетов. TCP разделяет этот поток на части для пересылки на другой узел в TCP-сегментах некоторого размера. Для отправки или получения сегмента модуль TCP вызывает модуль IP.

Немедленное отправление данных может быть затребовано процессом-клиентом от TCP-модуля с помощью специальной функции PUSH, иначе TCP сам будет решать, как накапливать и когда отправлять данные клиента или когда передавать клиенту полученные данные.

1.2 Обеспечение достоверности передачи

Модуль TCP обеспечивает защиту от повреждения, потери, дублирования и нарушения очередности получения данных. Для выполнения этих задач все октеты в потоке данных сквозным образом пронумерованы в возрастающем порядке. Заголовок каждого сегмента содержит число октетов данных в сегменте и порядковый номер первого октета той части потока данных, которая пересылается в данном сегменте. Например, если в сегменте пересылаются октеты с номерами от 2001 до 3000, то номер первого октета в данном сегменте равен 2001, а число октетов равно 1000. Номер первого байта в потоке определяется на этапе установления соединения и обозначается ISN+. Также для каждого сегмента вычисляется контрольная сумма, позволяющая обнаружить повреждение данных.

При удачном приеме октета данных принимающий модуль посылает отправителю подтверждение о приеме – номер удачно принятого октета. Если в течение некоторого времени отправитель не получит подтверждения, считается, что октет не дошел или был поврежден, и он посылается снова. Этот механизм контроля надежности называется PAR (Positive Acknowledgment with Retransmission). В действительности подтверждение посылается не для одного октета, а для некоторого числа последовательных октетов.

Нумерация октетов используется также для упорядочения данных в порядке очередности и обнаружения дубликатов (которые могут быть посланы из-за большой задержки при передаче подтверждения или потери подтверждения).

1.3 Адресация

Протокол TCP обеспечивает работу одновременно нескольких соединений. Каждый прикладной процесс идентифицируется *номером порта*. Заголовок TCP-сегмента содержит номера портов процесса-отправителя и процесса-получателя. При получении сегмента модуль TCP анализирует номер порта получателя и отправляет данные соответствующему прикладному процессу.

Все распространенные сервисы Интернет имеют стандартизованные номера портов. Например, номер порта сервера электронной почты – 25, сервера FTP – 21. Список стандартных номеров портов можно найти в файле /etc/services (Unix).

Совокупность IP-адреса и номера порта является *составным адресом*, который уникально идентифицирует прикладной процесс.

1.4 Управление соединениями

Соединение – это совокупность информации о состоянии потока данных, включающая составные адреса, номера посланных, принятых и подтвержденных октетов, размеры окон.

Соединение характеризуется для клиента именем, которое является указателем на структуру TCB (Transmission Control Block), содержащую информацию о соединении.

Открытие соединения клиентом осуществляется вызовом функции OPEN, которой передается составной адрес, с которым требуется установить соединение. Функция возвращает имя соединения. Различают два типа открытия соединения: активное и пассивное. При активном открытии TCP-модуль начинает процедуру установления соединения с указанным адресом, при пассивном – ожидает, что удаленный TCP-модуль начнет процедуру установления соединения с указанного адреса. Указание 0.0.0.0:0 в качестве адреса при пассивном открытии означает, что ожидается соединение с любого адреса. Клиент же применяет процедуру активного открытия; адрес при этом формируется из IP-адреса сервера и стандартного номера порта для данного сервиса. Закрытие соединения клиентом производится с помощью функции CLOSE, которой передается имя соединения.

Процедура установления соединения происходит следующим образом (рис. 1.1). Предположим, узел А желает установить соединение с узлом В. Пер-

вый отправляемый из А в В TCP-сегмент не содержит полезных данных, а служит для установления соединения. В его заголовке (в поле Flags) установлен бит SYN, означающий запрос связи, и содержится ISN (Initial Sequence Number – начальный номер последовательности) – число, начиная с которого узел А будет нумеровать отправляемые октеты (например, 0). В ответ на получение такого сегмента узел В откликается посылкой TCP-сегмента, в заголовке которого установлен бит ACK, подтверждающий установление соединения для получения данных от узла А. Так как протокол TCP обеспечивает полнодуплексную передачу данных, то узел В в этом же сегменте устанавливает бит SYN, означающий запрос связи для передачи данных от В к А, и передает свой ISN (например, 0). Полезных данных этот сегмент также не содержит. Третий TCP-сегмент в сеансе посылается из А в В в ответ на сегмент, полученный из В. Так как соединение А -> В можно считать установленным (получено подтверждение от В), то узел А включает в свой сегмент полезные данные, нумерация которых начинается с номера ISN(A)+1. Данные нумеруются по количеству отправленных октетов. В заголовке этого же сегмента узел А устанавливает бит ACK, подтверждающий установление связи В -> А, что позволяет системе В включить в свой следующий сегмент полезные данные для А.

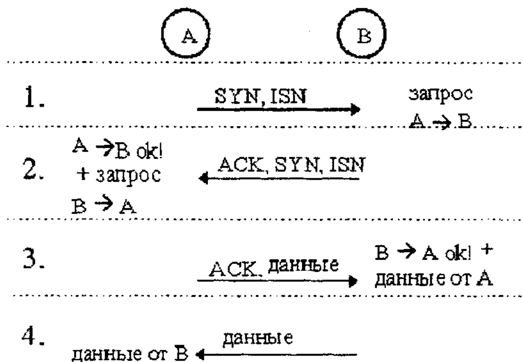

Рис. 1.1 – Установка TCP-соединения

Сеанс обмена данными заканчивается процедурой разрыва соединения, которая аналогична процедуре установки, с той разницей, что вместо SYN для разрыва используется служебный бит FIN ("данных для отправки больше не имею"), который устанавливается в заголовке последнего сегмента с данными, отправляемого узлом.

1.5. Управление потоком

Для ускорения и оптимизации процесса передачи больших объемов данных протокол TCP определяет метод управления потоком, называемый методом скользящего окна, который позволяет отправителю посылать очередной сегмент, не дожидаясь подтверждения о получении в пункте назначения предшествующего сегмента.

Протокол TCP формирует подтверждения не для каждого конкретного успешно полученного пакета, а для всех данных от начала посылки до некоторого порядкового номера ACK SN (Acknowledge Sequence Number) исключительно. В качестве подтверждения успешного приема, например, первых 2000 байт, высылается ACK SN = 2001: это означает, что все данные в байтовом потоке под номерами от ISN+1=1 до данного ACK SN -1 (2000) успешно получены (рис. 1.2).

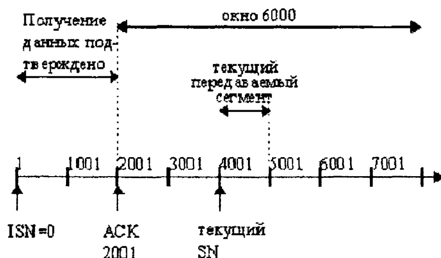

Рис. 1.2 – Метод скользящего окна

Вместе с посылкой отправителю ACK SN получатель объявляет также “размер окна”, например – 6000. Это значит, что отправитель может посылать данные с порядковыми номерами от текущего ACK SN = 2001 до (ACK SN + размер окна -1) = 8000, не дожидаясь подтверждения со стороны получателя. Допустим, в данный момент отправитель посылает тысячеоктетный сегмент с порядковым номером данных SN=4001. Если не будет получено новое подтверждение (новый ACK SN), отправитель будет посылать данные, пока он остается в пределах объявленного окна, то есть до номера 8001. После этого посылка данных будет прекращена до получения очередного подтверждения и (возможно) нового размера окна. Однако размер окна выбирается таким образом, чтобы подтверждения успевали приходить вовремя и остановки передачи не происходило – для этого и предназначен метод скользящего окна. Размер окна может динамически изменяться получателем.

Для временной остановки посылки данных достаточно объявить нулевое окно. Но даже и в этом случае через определенные промежутки времени будут отправляться сегменты с одним октетом данных. Это делается для того, чтобы отправитель гарантированно узнал о том, что получатель вновь объявил ненулевое окно, поскольку получатель обязан подтвердить получение “пробных” сегментов, а в этих подтверждениях он укажет также и текущий размер своего окна.

Как уже было сказано выше, протокол TCP позволяет вести полнодуплексную передачу. Один и тот же сегмент, высылаемый, например, из B в A, может содержать в заголовке служебную информацию по подтверждению получения данных от A, а в поле данных – полезные данные для A.

Модуль TCP может оптимизировать максимальный размер сегмента исходя из значений MTU на разных участках маршрута и других характеристик соединения. Модуль TCP может использовать алгоритм "медленного старта", формируя при установлении соединения окно перегрузки, размер которого из-

начально равен размеру одного сегмента. Это окно показывает, сколько сегментов TCP-модуль, с его собственной точки зрения, может отправить без получения подтверждения. Скользящее же окно, рассмотренное выше, показывает, какой объем неподтвержденных данных модулю разрешено отправить с точки зрения удаленного модуля, получателя его данных. После прихода подтверждения от получателя окно перегрузки увеличивается на 1 сегмент, и отправитель может выслать уже два сегмента, не дожидаясь подтверждения. Такой подход позволяет постепенно увеличивать нагрузку на сеть. Если окно перегрузки становится больше скользящего окна, объявляемого получателем, ограничение на передачу неподтвержденных данных устанавливает уже скользящее окно получателя.

В случае, если никакие данные приложениями не передаются, а соединение открыто, модуль TCP может периодически посылать сегменты-зонды для выяснения того, не отключилась ли другая сторона без уведомления партнера (например, в результате обрыва линии или другим некорректным образом). Такое зондирование проводится примерно каждые два часа неактивности.

1.6 Заголовок TCP-сегмента

TCP-сегмент состоит из заголовка и данных. Заголовок сегмента состоит из 32-разрядных слов и имеет переменную длину, зависящую от размера поля Options, но всегда кратную 32 битам (рис. 1.3). За заголовком непосредственно следуют данные – часть потока данных пользователя, передаваемая в данном сегменте.

<table>
<tr><td colspan="2">0 7</td><td colspan="6">15</td><td>23 31</td></tr>
<tr><td colspan="8">Source Port</td><td>Destination Port</td></tr>
<tr><td colspan="9">Sequence Number (SN)</td></tr>
<tr><td colspan="9">Acknowledgment Number (ACK)</td></tr>
<tr><td>Data Offset (0-3)</td><td>reserved (4-9)</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Window</td></tr>
<tr><td colspan="8">Checksum</td><td>Urgent Pointer</td></tr>
<tr><td colspan="8">Options</td><td>Padding</td></tr>
</table>

Рис 1.3 – Формат заголовка сегмента TCP

Source Port (16 бит), **Destination Port** (16 бит) – номера портов процесса-отправителя и процесса-получателя соответственно.

Sequence Number (SN) (32 бита) – порядковый номер первого октета в поле данных сегмента среди всех октетов потока данных для текущего соединения, то есть если в сегменте пересылаются октеты с 2001-го по 3000-й, то SN=2001. Если в заголовке сегмента установлен бит SYN (фаза установления соединения), то в поле SN записывается начальный номер (ISN), например, 0. Номер первого октета данных, посылаемых после завершения фазы установления соединения, равен ISN+1. **Acknowledgment Number (ACK)** (32 бита) – если установлен бит ACK, то это поле содержит порядковый номер октета, кото-

рый отправитель данного сегмента желает получить. Это означает, что все предыдущие октеты (с номерами от ISN+1 до ACK-1 включительно) были успешно получены.

Data Offset (4 бита) – длина TCP-заголовка в 32-битных словах.

Reserved (6 бит) – зарезервировано; заполняется нулями.

Control Bits (6 бит) – управляющие биты; активным является положение "бит установлен":

URG – поле срочного указателя (Urgent Pointer) задействовано;

ACK – поле номера подтверждения (Acknowledgment Number) задействовано;

PSH – осуществить "проталкивание" – если модуль TCP получает сегмент с установленным флагом PSH, то он немедленно передает все данные из буфера приема процессу-получателю для обработки, даже если буфер не был заполнен;

RST – перезагрузка текущего соединения;

SYN – запрос на установление соединения;

FIN – нет больше данных для передачи.

Window (16 бит) – размер окна в октетах.

Checksum (16 бит) – контрольная сумма, представляет собой 16 бит, дополняющие биты в сумме всех 16-битовых слов сегмента (само поле контрольной суммы перед вычислением обнуляется). Контрольная сумма, кроме заголовка сегмента и поля данных, учитывает 96 бит псевдозаголовка, который для внутреннего употребления ставится перед TCP-заголовком. Этот псевдозаголовок содержит IP-адрес отправителя (4 октета), IP-адрес получателя (4 октета), нулевой октет, 8-битное поле "Протокол", аналогичное полю в IP-заголовке, и 16 бит длины TCP сегмента, измеренной в октетах. Такой подход обеспечивает защиту протокола TCP от ошибшихся в маршруте сегментов. Информация для псевдозаголовка передается через интерфейс "Протокол TCP/межсетевой уровень" в качестве аргументов или результатов запросов от протокола TCP к протоколу IP.

Urgent Pointer (16 бит) – используется для указания длины срочных данных, которые размещаются в начале поля данных сегмента. Указывает смещение октета, следующего за срочными данными, относительно первого октета в сегменте. Например, в сегменте передаются октеты с 2001-го по 3000-й, при этом первые 100 октетов являются срочными данными, тогда Urgent Pointer = 100. Протокол TCP не определяет, как именно должны обрабатываться срочные данные, но предполагает, что прикладной процесс будет предпринимать усилия для их быстрой обработки. Поле Urgent Pointer задействовано, если установлен флаг URG.

Options – поле переменной длины; может отсутствовать или содержать одну опцию или список опций, реализующих дополнительные услуги протокола TCP.

Padding – выравнивание заголовка по границе 32-битного слова, если список опций занимает нецелое число 32-битных слов. Поле Padding заполняется нулями.

1.7 Промежуточные состояния соединения

TCP-соединение во время функционирования проходит через ряд промежуточных состояний. Это состояния LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT, а также фиктивное состояние CLOSED. (Состояние CLOSED является фиктивным, поскольку оно представляет отсутствие соединения.) Переход из одного состояния в другое происходит в ответ на события, как то: запросы клиента, приход сегментов, истечение контрольного времени.

Определены следующие запросы процесса-клиента модулю TCP (с каждым запросом, кроме OPEN, передается имя соединения):

ACTIVE-OPEN – активное открытие соединения;

PASSIVE-OPEN – пассивное открытие соединения;

SEND – отправка данных (передается указатель на буфер данных, размер буфера, значения флагов URG и PSH);

RECEIVE – получение данных (передается указатель на буфер данных, размер буфера; возвращается счетчик полученных октетов, значения флагов URG и PSH);

STATUS – запрос состояния соединения;

CLOSE – закрытие соединения (производится досылка всех неотправленных данных и обмен сегментами с битом FIN);

ABORT – ликвидация соединения (уничтожаются блок TCB и все неотправленные данные, посылается сегмент с битом RST).

Деятельность программы протокола TCP можно рассматривать как реагирование на события в зависимости от состояния соединения. Ниже описаны состояния соединения и приведены диаграммы переходов. Под термином "процесс" здесь понимается процесс TCP-модуля, работающий с данным соединением на локальном узле; термин "чужой" относится к процессу, работающему с данным TCP-соединением на удаленном узле.

LISTEN – процесс пассивно ждет запроса со стороны чужих сокетов.

SYN-SENT – процесс отправил свой SYN и ждет чужого SYN.

SYN-RECEIVED – процесс получил чужой SYN, отправил (раньше или только что) свой SYN и ждет ACK на свой SYN.

ESTABLISHED – процесс отправил ACK на чужой SYN, получил ACK на свой SYN; соединение установлено.

FIN-WAIT-1 – процесс первый отправил свой FIN и ждет реакцию той стороны; при этом он, возможно, продолжает получать данные.

FIN-WAIT-2 – процесс получил ACK на свой ранее отправленный FIN, но не получил чужой FIN; ждет чужой FIN; при этом, возможно, продолжает получать данные.

CLOSE-WAIT – процесс, не отправив свой FIN (возможно, не собираясь прекращать соединение), получает чужой FIN; он отправляет ACK на чужой FIN, но при этом, возможно, продолжает отправлять данные.

LAST-ACK – процесс отправил свой FIN, но ранее он уже получил FIN с той стороны и отправил на него ACK; поэтому процесс ожидает чужой ACK на свой FIN для окончательного закрытия соединения.

CLOSING – процесс ранее отправил свой FIN и еще не получил не него подтверждение, но получил чужой FIN (и отправил на него ACK); ждет ACK на свой FIN.

TIME-WAIT – процесс ранее отправил свой FIN и получил на него подтверждение, получил чужой FIN и только что отправил на него ACK; теперь процесс ждет некоторое время (два времени жизни сегмента, обычно 4 минуты) для гарантии того, что та сторона получит его ACK на свой FIN, после чего соединение будет окончательно закрыто.

CLOSED – соединение отсутствует.

На рис. 1.4 и рис 1.5 приведены фазы установления и закрытия соединения.

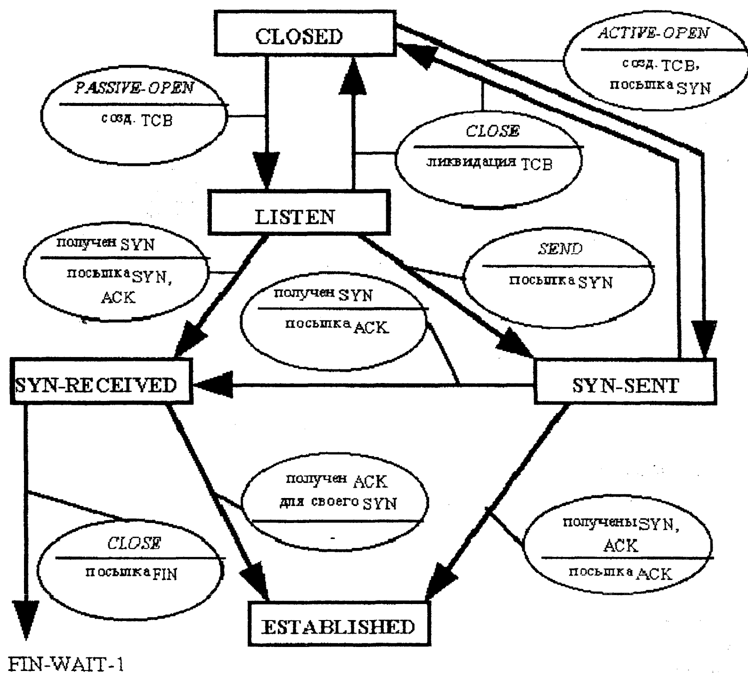

Рис. 1.4 – Фаза установления соединения

На рис. 1.4 и рис 1.5 состояния соединения заключены в прямоугольники, переходы между ними показаны стрелками, с каждой стрелкой соотносится овал, поясняющий причину перехода. В овале над горизонтальной чертой указывается событие, вызвавшее переход (курсивом обозначено поступление запросов от локального процесса-клиента); под горизонтальной чертой – действие, сопутствующее переходу.

Проблемы возникновения некорректных ситуаций, например, наполовину открытое соединение, получение заблудившихся в сети старых SYN-сегментов, неожиданный крах программ и т.п., решаются путем детектирования ошибки (несоответствие или бессмысленные значения ACK или SN), после чего посы-

лается сигнал RST (сегмент с установленным битом RST) и соединение ликвидируется.

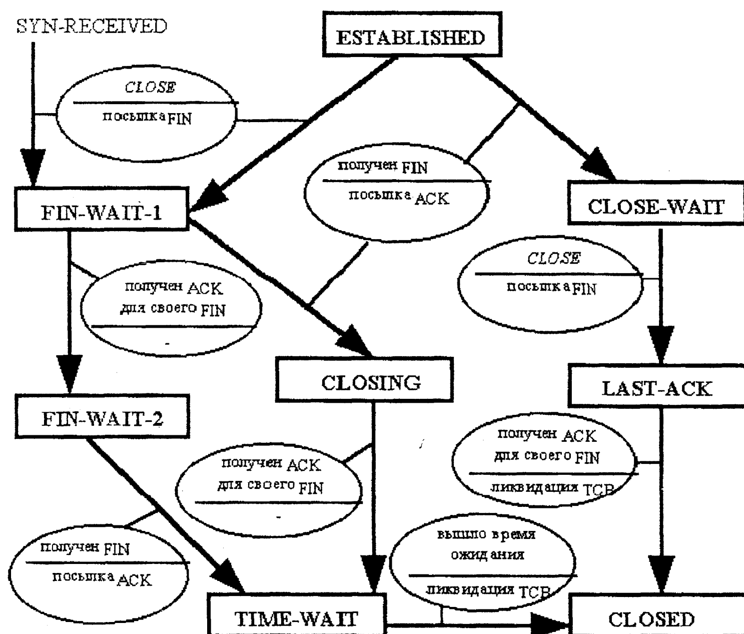

Рис. 1.5 – Фаза закрытия соединения

1.8 Протокол UDP

UDP (User Datagram Protocol, протокол пользовательских дейтаграмм) фактически не выполняет каких-либо особых функций дополнительно к функциям межсетевого уровня (протокола IP). Протокол UDP используется либо при пересылке коротких сообщений, когда накладные расходы на установление сеанса и проверку успешной доставки данных оказываются выше расходов на повторную (в случае неудачи) пересылку сообщения, либо в том случае, когда сама организация процесса-приложения обеспечивает установление соединения и проверку доставки пакетов (например, NFS).

Пользовательские данные, поступившие от прикладного уровня, предваряются UDP-заголовком, и сформированный таким образом UDP-пакет отправляется на межсетевой уровень. UDP-заголовок состоит из двух 32-битных слов (рис 1.6).

0 7	15	23	31
Source Port		Destination Port	
Length		Checksum	

Рис 1.6 – Заголовок UDP

Source Port – номер порта процесса-отправителя.

Destination Port – номер порта процесса-получателя.

Length – длина UDP-пакета вместе с заголовком в октетах.

Checksum – контрольная сумма. Контрольная сумма вычисляется таким же образом, как и в TCP-заголовке; если UDP-пакет имеет нечетную длину, то при вычислении контрольной суммы к нему добавляется нулевой октет.

После заголовка непосредственно следуют пользовательские данные, переданные модулю UDP прикладным уровнем за один вызов. Протокол UDP рассматривает эти данные как целостное сообщение; он никогда не разбивает сообщение для передачи в нескольких пакетах и не объединяет несколько сообщений для пересылки в одном пакете. Если прикладной процесс N раз вызвал модуль UDP для отправки данных (т.е. запросил отправку N сообщений), то модулем UDP будет сформировано и отправлено N пакетов, и процесс-получатель будет должен N раз вызвать свой модуль UDP для получения всех сообщений.

При получении пакета от межсетевого уровня модуль UDP проверяет контрольную сумму и передает содержимое сообщения прикладному процессу, чей номер порта указан в поле "Destination Port".

Если проверка контрольной суммы выявила ошибку или если процесса, подключенного к требуемому порту, не существует, пакет игнорируется. Если пакеты поступают быстрее, чем модуль UDP успевает их обрабатывать, то поступающие пакеты также игнорируются. Протокол UDP не имеет никаких средств подтверждения безошибочного приема данных или сообщения об ошибке, не обеспечивает приход сообщений в порядке отправки, не производит предварительного установления сеанса связи между прикладными процессами, поэтому он является *ненадежным* протоколом *без установления соединения*. Если приложение нуждается в подобного рода услугах, оно должно использовать на транспортном уровне протокол TCP.

Максимальная длина UDP-сообщения равна максимальной длине IP-дейтаграммы (65535 октетов) за вычетом минимального IP-заголовка (20) и UDP-заголовка (8), т.е. 65507 октетов. На практике обычно используются сообщения длиной 8192 октета.

Примеры прикладных процессов, использующих протокол UDP: NFS (Network File System – сетевая файловая система), TFTP (Trivial File Transfer Protocol – простой протокол передачи файлов), SNMP (Simple Network Management Protocol – простой протокол управления сетью), DNS (Domain Name Service – доменная служба имен).

2 Программирование сетевых приложений на базе WINSOCK

2.1 Понятие сокетов

Сокеты (sockets) представляют собой высокоуровневый унифицированный интерфейс взаимодействия с телекоммуникационными протоколами. В технической литературе встречаются различные переводы этого слова – их называют и гнездами, и соединителями, и патронами, и патрубками, и т .д. По

причине отсутствия устоявшегося русскоязычного термина, в настоящей работе сокеты будет именоваться сокетами и никак иначе.

В работе рассмотрена одна реализации сокетов – библиотека Winsock 2, на языке программирования – **Си/Си ++** и один вид сокетов – **блокируемые синхронные сокеты.**

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.

ОС Windows 3.x поддерживает только асинхронные сокеты, поскольку, в среде с корпоративной многозадачностью захват управления одной задачей "подвешивает" все остальные, включая и саму систему. ОС Windows 9x\NT поддерживают оба вида сокетов, однако, в силу того, что синхронные сокеты программируются более просто, чем асинхронные, последние не получили большого распространения. Эта статья посвящена исключительно синхронным сокетам (асинхронные – тема отдельного разговора).

Сокеты позволяют работать со множеством протоколов и являются удобным средством межпроцессорного взаимодействия, но в данной работе речь будет идти только о сокетах семейства протоколов TCP/IP, использующихся для обмена данными между узлами сети Интернет. Все остальные протоколы, такие как IPX/SPX, NetBIOS в рамках данной учебно-методической разработки рассматриваться не будут.

Независимо от вида, сокеты делятся на два типа – ***потоковые*** и ***дейтаграммные***. Потоковые сокеты работают с установкой соединения, обеспечивая надежную идентификацию обоих сторон и гарантируют целостность и успешность доставки данных. Дейтаграмные сокеты работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных, зато они заметно быстрее потоковых.

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.

***Замечание**: дейтаграммные сокеты опираются на протокол UDP, а потоковые – на TCP.*

2.2 Работа с библиотекой Winsock2.h

Первый этап. Для работы с библиотекой Winsock 2.x в исходный тест программы необходимо включить директиву

"*#include <winsock2.h>*",

а также убедиться, что библиотека "ws2_32.lib" находится в каталоге /lib.

Перед началом использования функций библиотеки Winsock ее необходимо подготовить к работе вызовом функции

"*int **WSAStartup** (WORD wVersionRequested, LPWSADATA lpWSAData)*",

передав в старшем байта слова *wVersionRequested* номер требуемой версии, а в младшем – номер подверсии.

Аргумент *lpWSAData* должен указывать на структуру *WSADATA*, в которую при успешной инициализации будет занесена информация о производителе библиотеки. Никакого особенного интереса она не представляет, и прикладное приложение может ее игнорировать. В случае неудачной инициализации функция возвращает ненулевое значение.

Второй этап. Создание объекта "сокет". Это осуществляется функцией "*SOCKET **socket** (int af, int type, int protocol)*".

Первый слева аргумент указывает на семейство используемых протоколов. Для Интернет – приложений он должен иметь значение AF_INET.

Следующий аргумент задает тип создаваемого сокета – *потоковый* (*SOCK_STREAM*) или *дейтаграммный* (*SOCK_DGRAM*).

Последний аргумент уточняет, какой транспортный протокол следует использовать. 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.

Дальнейшие шаги зависит от того, является приложение сервером или клиентом. Ниже эти два случая будут описаны раздельно.

Третий этап (клиент). Для установки соединения с удаленным узлом потоковый сокет должен вызвать функцию

"*int **connect** (SOCKET s, const struct sockaddr FAR* name, int namelen)*".

Датаграмные сокеты работают без установки соединения, поэтому обычно не обращаются к функции connect.

***Примечание**: вызов connect позволяет дейтаграммному сокету обмениваться данными с узлом не только функциями sendto, recvfrom, но и более удобными и компактными send и recv. Этот нюанс описан в Winsocket SDK и широко используется как самой Microsoft, так и сторонними разработчиками. Поэтому ее использование вполне безопасно.*

Первый слева аргумент – дескриптор сокета, возращенный функцией socket; второй – указатель на структуру "*sockaddr*", содержащую в себе адрес и порт удаленного узла, с которым устанавливается соединение. Структура sockaddr используется множеством функций, поэтому ее описание вынесено в отдельный подраздел 2.4. Последний аргумент сообщает функции размер структуры sockaddr.

После вызова connect система предпринимает попытку установить соединение с указанным узлом. Если по каким-то причинам это сделать не удастся (адрес задан неправильно, узел не существует или "висит", компьютер находится не в сети), функция возвратит ненулевое значение.

Третий этап (сервер) – прежде, чем сервер сможет использовать сокет, он должен связать его с локальным адресом. Локальный адрес состоит из IP-адреса узла и номера порта. Если сервер имеет несколько IP адресов, то сокет может быть связан как со всеми сразу (для этого вместо IP-адреса следует указать константу INADDR_ANY равную нулю), так и с каким-то конкретным одним.

Связывание осуществляется вызовом функции

"*int **bind** (SOCKET s, const struct sockaddr FAR* name, int namelen)*".

Первым слева аргументом передается дескриптор сокета, возращенный функций socket, за ним следуют указатель на структуру sockaddr и ее длина.

Строго говоря, клиент также должен связывать сокет с локальным адресом перед его использованием, однако за него это делает функция connect, ассоциируя сокет с одним из портов, наугад выбранных из диапазона 1024-5000. Сервер же должен "садиться" на заранее определенный порт, например, 21 – для FTP, 23 – для telnet, 25 – для SMTP, 80 – для WEB, 110 – для POP3 и т.д. Поэтому ему приходится осуществлять связывание "вручную".

При успешном выполнении функция возвращает нулевое значение и ненулевое в противном случае.

<u>Четвертый этап (сервер).</u> Выполнив связывание, потоковый сервер переходит в режим ожидания подключений, вызывая функцию

"*int **listen** (SOCKET s, int backlog)*",

где *s* – дескриптор сокета, а *backlog* – максимально допустимый размер очереди сообщений.

Размер очереди ограничивает количество одновременно обрабатываемых соединений, поэтому к его выбору следует подходить внимательно. Если очередь полностью заполнена, очередной клиент при попытке установить соединение получит отказ (TCP пакет с установленным флагом RST). В то же время максимально разумное количество подключений определяются 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, минуя четвертый и пятый шаги.

<u>Пятый этап (сервер).</u> Извлечение запросов на соединение из очереди осуществляется функцией

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

которая автоматически создает новый сокет, выполняет связывание и возвращает его дескриптор, а в структуру *sockaddr* заносит сведения о подключившемся клиенте (IP-адрес и порт). 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.

Для параллельной работы с несколькими клиентами 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.

После того как соединение установлено, потоковые сокеты могут обмениваться с удаленным узлом данными, вызывая функции

"*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.

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 не переданными, но вызывающий код не получит об этом никакого уведомления. А ошибка возвращается лишь в том случае, если соединение разорвано до вызова функции send.

Функция *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 часть дейтаграммы, а только ею всю целиком. Работой обеих функций можно управлять с помощью *флагов*, передаваемых в одной переменной типа int третьим слева аргументом. Эта переменная может принимать одно из двух значений: *MSG_PEEK* и *MSG_OOB*.

Флаг MSG_PEEK заставляет функцию recv просматривать данные вместо их чтения. Просмотр, в отличие от чтения, не уничтожает просматриваемые данные. Некоторые источники утверждают, что при взведенном флаге MSG_PEEK функция recv не задерживает управления, если в локальном буфере нет данных, доступных для немедленного получения. Однако это неверно. Аналогично, иногда приходится встречать откровенно ложное утверждение, якобы функция send со взведенным флагом MSG_PEEK возвращает количество уже переданных байт (вызов send не блокирует управления). На самом деле функция send игнорирует этот флаг.

Флаг MSG_OOB предназначен для передачи и приема *срочных* (*Out Of Band*) данных. Срочные данные не имеют преимущества перед другими при пересылке по сети, а всего лишь позволяют оторвать клиента от нормальной обработки потока обычных данных и сообщить ему "срочную" информацию. Если данные передавались функцией send с установленным флагом MSG_OOB, для их чтения флаг MSG_OOB функции recv также должен быть установлен.

***Замечание**: рекомендуется воздержаться от использования срочных данных в своих приложениях. Во-первых, они совершенно необязательны – гораздо проще и надежнее вместо них создать отдельное TCP-соединение. Во-вторых, по поводу их реализации нет единого мнения, и интерпретации различных производителей очень сильно отличаются друг от друга. Так, разработчики до сих пор не пришли к окончательному соглашению по поводу того, куда должен указывать указатель срочности: либо на последний байт срочных данных, либо на байт, следующий за последним байтом срочных данных. В ре-*

БИБЛИОТЕКА
Брестского государственного

зультате, отправитель никогда не имеет уверенности, что получатель сможет правильно интерпретировать его запрос.

Дейтаграммный сокет так же может пользоваться функциями send и recv, если предварительно вызовет connect, но у него есть и свои, "персональные", 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 принимаемого или передаваемого данные. Вызов recvfrom не требует 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 recvfrem, получившей сообщение от клиента.

Замечание: *транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной доставки сообщений, и эта задача возлагается на самого 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).

Шестой этап (заключительный). Для закрытия соединения и уничтожения сокета предназначена функция

"*int **closesocket** (SOCKET s)*",

которая в случае удачного завершения операции возвращает нулевое значение.

Перед выходом из программы необходимо вызвать функцию

"*int **WSACleanup** (void)*"

для деинициализации библиотеки WINSOCK и освобождения используемых этим приложением ресурсов.

Внимание: *завершение процесса функцией ExitProcess автоматически не освобождает ресурсы сокетов.*

Примечание: *более сложные приемы закрытия соединения – 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" и 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.

***Внимание**: вызов shutdown не освобождает от необходимости закрытия сокета функцией closesocket!*

2.3 Дерево вызовов

Для большей наглядности демонстрации взаимосвязи socket-функций друг с другом на рис. 2.1 приведено дерево вызовов, показывающее, в каком порядке должны следовать вызовы функций в зависимости от типа сокетов (потоковый или дейтаграммный) и рода обработки запросов (клиент или сервер).

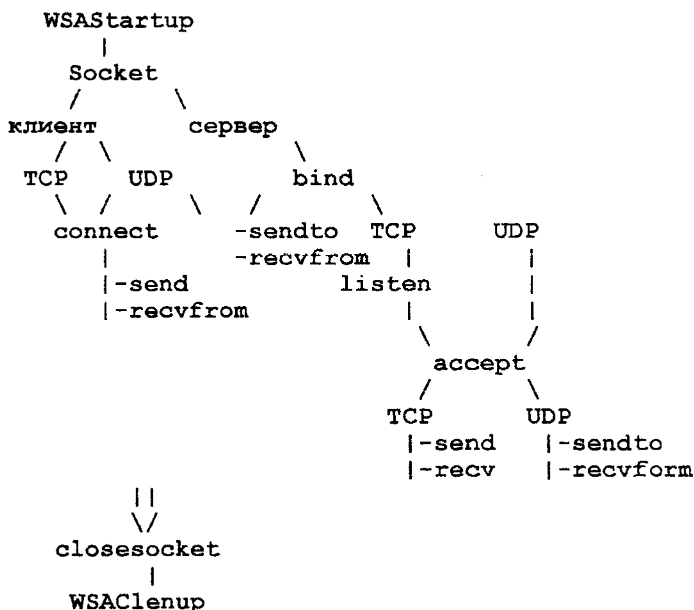

Рис. 2.1 – Последовательность вызова функций сокетов при различных операциях

2.4 Адреса и правила их использования

Рассмотрим более детально структуру адреса. Структура sockaddr, определенная так:

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

теперь уже считается устаревшей, и в Winsock 2.x на смену ей пришла структура sockaddr_in, определенная следующим образом:

```
struct sockaddr_in
{
  short  sin_family;      // семейство протоколов
           // (как правило AF_INET)
  u_short  sin_port;     // порт
  struct  in_addr sin_addr;  // IP - адрес
  char  sin_zero[8];     // хвост
};
```

В принципе ничего не изменилось, замена беззнакового короткого целого на знаковое короткое целое для представления семейства протоколов ничего не дает. Зато теперь адрес узла 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***. Он необходим, поскольку структура sockaddr не привязана именно к Интернету и может работать и с другими сетями. Адреса же некоторых сетей требуют для своего представления гораздо больше четырех байт.

Структура in_addr определяется следующим в 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-алрес
  } S_un;
}
```

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

На практике можно с одинаковым успехом пользоваться как sockaddr, так и sockaddr_in. Однако поскольку прототипы остальных функций не измени-

лись, при использовании sockaddr_in придется постоянно выполнять явные преобразования, например, так:

*"**sockaddr_in** dest_addr; connect (mysocket, (**struct sockaddr***) &dest_addr, sizeof(dest_addr)".*

Для преобразования IP-адреса, записанного в виде символьной последовательности наподобие "127.0.0.1" в четырехбайтовую числовую последовательность, предназначена функция

*"unsigned long **inet_addr** (const char FAR * cp)".*

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

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

Попытка передать inet_addr доменное имя узла будет неудачной.

Узнать IP-адрес такого-то домена можно с помощью функции

*"struct hostent FAR * **gethostbyname** (const char FAR * name);".*

Функция обращается к DNS и возвращает свой ответ в структуре hostent или нуль, если DNS сервер не смог определить IP-адрес данного домена.
Структура hostent выглядит следующим образом:

```
struct hostent
{
  char FAR *  h_name;     // официальное имя узла
  char FAR * FAR* h_aliases;  // альтернативные имена
           // узла (массив строк)

  short    h_addrtype;    // тип адреса
  short    h_length;    // длина адреса
           // (как правило, AF_INET)

  char FAR * FAR * h_addr_list;  // список указателей
           //на IP-адреса
           // ноль - конец списка
};
```

Как и в случае с in_addr, во множестве программ и прилагаемых к Winsock SDK примерах активно используется недокументированное поле структуры h_addr. Например, вот строка из файла *"simplec.c"*

*"memcpy(&(server.sin_addr),hp->**h_addr**,hp->h_length);"*

Обратившись в "winsock2.h", можно найти, что оно обозначает:

"#define h_addr h_addr_list[0]".

Следует отметить, что с некоторыми доменными именами связано сразу несколько IP-адресов. В случае неработоспособности одного узла клиент может

попробовать подключится к другому или просто выбрать узел с наибольшей скоростью обмена. Но в приведенном примере клиент использует только первый IP-адрес в списке и игнорирует все остальные. В связи с этим все же будет лучше, если в своих программах будет учтена возможность подключения к остальным IP-адресам при невозможности установить соединение с первым.

Функция gethostbyname ожидает на входе только доменные имена, но не цифровые IP-адреса. Между тем, правила "хорошего тона" требуют предоставления клиенту возможности как задания доменных имен, так и цифровых IP-адресов.

Решение заключается в следующем: необходимо проанализировать переданную клиентом строку. Если это IP адрес, то передать его функции inet_addr в противном случае – gethostbyaddr, полагая, что это доменное имя. Для отличия IP-адресов от доменных имен многие программисты используют следующий прием: если первый символ строки – цифра, это IP-адрес, иначе – имя домена. Однако такой прием не всегда приемлем – доменные имена могут начинаться с цифры, например, "666.ru", могут они и заканчиваться цифрой, например, к узлу "666.ru" члены субдомена "666" могут так и обращаться – "666". Поэтому лучше всего действовать так: передаем введенную пользователем строку функции inet_addr, если она возвращает ошибку, то вызываем gethostbyaddr.

Для решения 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 тем исключением, что ее аргументом является не указатель на строку, содержащую имя, а указатель на четырехбайтовый IP-адрес. Еще два аргумента задают его длину и тип (соответственно, 4 и AF_INET).

Определение имени узла по его адресу бывает полезным для серверов, желающих "в лицо" знать своих клиентов.

Для преобразования IP-адреса, записанного в сетевом формате в символьную строку, предусмотрена функция

"*char FAR * **inet_ntoa** (struct in_addr)*",
которая принимает на вход структуру in_addr, а возвращает указатель на строку, если преобразование выполнено успешно, и ноль – в противном случае.

2.5 Сетевой порядок байт

Для устранения возможных проблем при межсетевом взаимодействии был введен специальный *сетевой порядок байт*, предписывающий старший байт передавать первым.

Для преобразований чисел из сетевого формата в формат локального хоста и наоборот предусмотрено 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 (short) и длинное (long) 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.

***Внимание**: все значения, возвращенные socket-функциями уже находятся в сетевом формате и "вручную" их преобразовывать нельзя. Так как это преобразование исказит результат и приведен к неработоспособности. Чаще всего к вызовам этих функций прибегают для преобразования номера порта согласно сетевому порядку. Например: dest_addr.sin_port = htons(110).*

2.6 Дополнительные возможности

Для более "тонкой" настойки сокетов предусмотрена функция

"*int **setsockopt** (SOCKET s, int level, int optname, const char FAR * optval, int optlen)*".

Первый слева аргумент – дескриптор сокета, который собираются настраивать, *level – уровень* настойки. С каждым уровнем связан свой набор опций. Всего определено два уровня – **SOL_SOCKET** и **IPPROTO_TCP**. Ниже будет рассказано только о самых интересных опциях, а сведения обо всех остальных можно почерпнуть из Winsock SDK.

Третий слева аргумент представляет собой указатель на переменную, содержащую значение опции. Ее размер варьируется в зависимости от рода опции и передается через четвертый слева аргумент.

Уровень SOL_SOCKET

1. SO_RCVBUF (int) – задает размер входного буфера для приема данных. К TCP-окну никакого отношения не имеет, поэтому, может безболезненно варьироваться в широких пределах.
2. SO_SNDBUF (int) – задает размер входного буфера для передачи данных. Увеличение размера буферов на медленных каналах приводит к задержкам и снижает производительность.

Уровень IPPROTO_TCP

TCP_NODELAY (BOOL) – выключает Алгоритм Нагла. Алгоритм Нагла был разработан специально для прозрачного кэширования крохотных пакетов (тиниграмм). Когда один узел посылает другому несколько байт, к ним дописываются заголовки TCP и IP, которые в совокупности обычно занимают более 50 байт. Таким образом, при побайтовом обмене между узлами свыше 98% передаваемой по сети информации будет приходиться на служебные данные! Алгоритм Нагла состоит в следующем: отправляем первый пакет и, до тех пор, пока получатель не возвратит TCP-уведомление успешности доставки, не передаем в сеть никаких пакетов, а накапливаем их на локальном узле, собирая в один большой пакет. Такая техника совершенно прозрачна для прикладных приложений и в то же время позволяет значительно оптимизировать трафик, но в некоторых случаях, когда требуется действительно побайтовый обмен, Алгоритм Нагла приходится отключать (по умолчанию он включен).

Для получения текущих значений опций сокета предусмотрена функция

"int getsockopt (SOCKET s, int level, int optname, char FAR optval, int FAR* optlen)"*, которая полностью аналогична предыдущей, за исключением того, что не устанавливает опции, а возвращает их значения.

3 Лабораторная работа №1. Организация TCP – сервера

Задание на выполнение

3.1 Изучить теоретический материал по организации транспортной службы TCP, функции WINSOCK. Получить индивидуальное задание у преподавателя.

3.2 Используя одну из реализаций системы программирования C++ и выбрав некоторый свободный номер порта (например, 666), разработать программу работы TCP-эхо-сервера, выполняющего вывод приветствия "Hello, Student!" и функции согласно выданному варианту задания. В качестве клиента использовать программу telnet.

Для проверки работоспособности TCP-сервера запустите TCP-сервер и наберите в командной строке Windows "telnet.exe 127.0.0.1 666", где 127.0.0.1 обозначает локальный адрес узла (это специально зарезервированный для этой цели адрес, и он выглядит одинаково для всех узлов), а 666 – номер порта, на котором активизирован сервер. Если все работает успешно, то telnet установит соединение, и на экране появится приветствие "Hello, Student!". После этого возможно проверять функциональность сервера. Продемонстрировать работу системы преподавателю.

3.3 Выполнить п.3.2, рассредоточив клиента и сервера на разных ЭВМ. Продемонстрировать работу системы преподавателю.

3.4 Представить отчет, содержащий титульный лист, листинг программы с подробными комментариями основных фрагментов программы.

3.5 Подготовиться к защите лабораторной работы по теоретическому материалу, функциям WINSOCK, собственным результатам, полученным в ходе выполнения лабораторной работы.

Контрольные вопросы

1) Что является блоком данных протокола TCP?
2) Как обеспечивается передача данных в TCP?
3) Средства обеспечения достоверности передачи в TCP.
4) Структура адреса TCP.
5) Каков порядок создания логического канала в TCP? Охарактеризуйте фазу создания соединения по диаграмме, приведенной на рис 1.4.
6) Каков порядок разрыва логического канала в TCP? Охарактеризуйте фазу закрытия соединения по диаграмме, приведенной на рис 1.5.
7) Понятие и виды сокетов WINSOCK.
8) Как выполнить инициализацию окружения WINSOCK?
9) Как создать объект «сокет» WINSOCK?
10) Как установить соединение в WINSOCK?
11) Как осуществляется связывание сокета WINSOCK с локальным адресом?
12) Как инициализировать в сервере режим ожидания соединения в WINSOCK?

4 Лабораторная работа №2. Организация TCP- клиента

Задание на выполнение

4.1 Изучить теоретический материал, функции WINSOCK и процесс организации на их базе TCP-клиента. Получить индивидуальное задание у преподавателя.

4.2 Используя одну из реализаций системы программирования C++ и выбрав некоторый свободный номер порта (например, 666), разработать программу работы TCP-клиента. Выполнить проверку программы с использованием разработанного в предыдущей лабораторной работе TCP-сервера. Продемонстрировать работу системы преподавателю.

4.3 Представить отчет, содержащий титульный лист, листинг программы с подробными комментариями основных фрагментов программы.

4.4 Подготовиться к защите лабораторной работы по теоретическому материалу, функциям WINSOCK, собственным результатам, полученным в ходе выполнения лабораторной работы.

Контрольные вопросы

1) Порядок работы протокола TCP при управлении потоком.

2) Охарактеризуйте поля заголовка TCP-сегмента.

3) Каковы промежуточные состояния соединения TCP?

4) Как осуществляется на сервере извлечение запросов на соединение из очереди в WINSOCK?

5) Как осуществляется посылка и прием данных в WINSOCK?

5 Лабораторная работа №3. Организация UDP-сервера и UDP-клиента

Задание на выполнение

5.1 Изучить теоретический материал, функции WINSOCK и листинг программы реализации UDP-сервера и UDP-клиента. Получить индивидуальное задание у преподавателя.

5.2 Используя одну из реализаций системы программирования C++ и выбрав некоторый свободный номер порта (например, 666), разработать программу работы UDP-сервера, выполняющую функции согласно варианту задания.

5.3 Разработать программу работы UDP-клиента.

5.4 Выполнить проверку программ UDP – сервера и UDP – клиента. Продемонстрировать работу системы преподавателю.

5.5 Представить отчет, содержащий титульный лист, листинг программы с подробными комментариями основных фрагментов программ.

5.6 Подготовиться к защите лабораторной работы по теоретическому материалу, функциям WINSOCK, собственным результатам, полученным в ходе выполнения лабораторной работы.

Контрольные вопросы

1) Общая характеристика протокола UDP.

2) Структура заголовка UDP.

3) Охарактеризуйте процесс передачи данных по протоколу UDP.

4) Каковы особенности осуществления посылки и приема данных для дейтаграммных сокетов WINSOCK?

5) Как осуществляется закрытие соединения и уничтожение сокета WINSOCK?

6 Список использованных источников

1 М.А.Мамаев. Телекоммуникационные технологии: Сети TCP/IP. Учебное пособие – Владивосток: Изд-во ВГУЭиС, 1999.

2 А.Леинванд, Б.Пински. Конфигурирование маршрутизаторов Cisco. 2-е издание – М.: "Вильямс", 2001.

3 В.Олифер, Н.Олифер. Компьютерные сети. Принципы, технологии, протоколы – СПб: "Питер", 2001.

4 Э. Немет, Г. Снайдер, С. Сибасс, Т. Хейн. UNIX: руководство системного администратора. Для профессионалов – СПб: "Питер", 2002.

5 Й.Снейдер. Эффективное программирование TCP/IP. Библиотека программиста – СПб: "Питер", 2002.

УЧЕБНОЕ ИЗДАНИЕ

Составители:
Савицкий Юрий Викторович
Муравьев Геннадий Леонидович

ПРОГРАММИРОВАНИЕ СЕТЕВЫХ ПРИЛОЖЕНИЙ НА БАЗЕ ТРАНСПОРТНЫХ ПРОТОКОЛОВ TCP, UDP С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕКИ WINSOCK

Методические указания

по выполнению лабораторных работ по курсу
«АППАРАТНОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ СЕТЕЙ»
для студентов специальности 1-53 01 02, 1-40 03 01

Ответственный за выпуск: Савицкий Ю.В.

Редактор: Строкач Т. В.

Компьютерная вёрстка: Кармаш Е.Л.

Корректор: Никитчик Е.В.

Подписано к печати 28.09.2009 г. Формат 60x84/16
Усл. п. л. 1,63. Уч. изд. л. 1,75. Тираж 40 экз. Заказ № 895
Отпечатано на ризографе учреждение образования «Брестский государственный технический университет»
224017, Брест, ул. Московская, 267.