**М.В. Локшин, С.А. Рыков**

# ОСНОВЫ КОМПЬЮТЕРНЫХ СЕТЕЙ. ПРОГРАММИРОВАНИЕ И ПРОТОКОЛЫ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

**Учебное пособие**

*Рекомендовано редакционно-издательским советом Воронежского государственного технического университета в качестве учебного пособия для студентов, обучающихся по направлению «Информатика и вычислительная техника»*

**Воронеж**
**Издательство «Научная книга»**
**2014**

**УДК 004.7 (075.8)**
**ББК 32.973.202 я7**
**Л 73**

**Рецензенты:**

**Пасмурнов С.М.,** канд. техн. наук, доц. (Воронежский государственный технический университет);
**Кафедра** вычислительной математики и прикладных информационных технологий Воронежского государственного университета

**Л 73 Локшин, М.В.** Основы компьютерных сетей. Программирование и протоколы. Лабораторный практикум: Учебное пособие/ М.В.Локшин, С.А.Рыков. – Воронеж: Издательство «Научная книга», 2014. - 128 с.

**ISBN 978-5-98222-862-8**

Учебное пособие предназначено для студентов третьего и четвертого курса направления 230100.62 «Вычислительные машины, комплексы, системы и сети», изучающих дисциплину «Сети и телекоммуникации».

**УДК 004.7 (075.8)**
**ББК 32.973.202 я7**
**Л 73**

**ISBN 978-5-98222-862-8** 

# ВВЕДЕНИЕ

В настоящем учебном пособии приводится информация, достаточная для того, чтобы читатель мог начать разрабатывать простейшие сетевые приложения. Прежде всего, для изучения сетевых протоколов нам потребуется вносить ралзичные изменения в сетевые настройки операционной системы. Чтобы это можно было делать, не опасаясь выхода из строя сети вычислительной лаборатории, мы изучим основы работы и конфигурировния виртуальных машин.

Далее изучаются основы работы с сокетами – в настоящее время, это основной способ сетевого взаимодействия между различными сетевыми узлами. На его основе строится множество современных протоколов.

Далее изучаются основы конфигурирования FTP сервера и реализация FTP протокола в среде Windows. Приводится описание API для работы с FTP-серверами.

В следующей части учебного пособия дается теоретическая информация по протоколам канального уровня, и примеры реализаций различных протоколов. Читателю предлагается реализовать эмуляцию протоколов физического уровня, для огранизации передачи данных между двумя программами при помощи изученных протоколов.

В последней части учебного пособия затронуты вопросы сетевой безопасности и изучается работа утилиты Wireshark, предназначенной для перехвата и анализа сетевого траффика.

***Лабораторная работа №1. Создание и настройка виртуальных машин. Конфигурирование сети.***

**Цель работы:** приобрести навыки настройки виртуальных машин и конфигурирования сети из нескольких виртуальных машин. Приобрести навыки конфигурирования сервера контроллера домена, DHCP, DNC и др.

**Установка и настройка виртуальной машины.**

Для работы будем использовать программу VirtualBox 4.2.18. В программе создается виртуальная машина (мы будем использовать операционную системуWindows XP Professional). На рисунках 1.1 – 1.4 показан процесс создания виртуальной машины. На первом рисунке задаем имя конфигурации и выбираем версию операционной системы, далее нажимаем Next. Изображения могут отличаться в зависимости от версии используемого эмулятора (изображения приведены для версии 4.2.18).

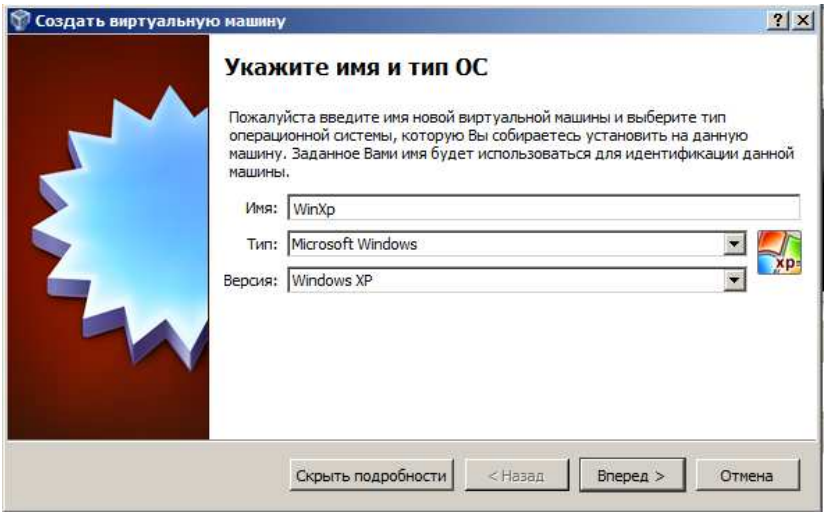


Рисунок 1.1 – Создание виртуальной машины

На рисунке 1.2 указываем объем оперативной памяти, используемый для эмуляции виртуальной машины. В данном случае – оставляем значение, используемое по умолчанию.

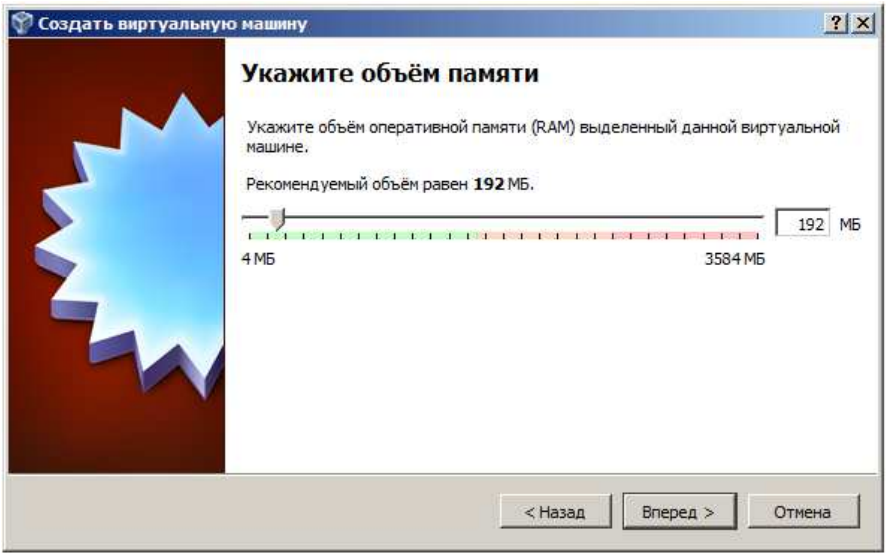


Рисунок 1.2 – Конфигурирование оперативной памяти

На рисунке 1.3 указываем местонахождение жесткого диска виртуальной машины. В данном случае – выбран существующий жесткий диск с уже установленной операционной системой. Если требуется установить операционную систему, то следует создать новый виртуальный жесткий диск и обеспечить доступ к дистрибутиву операционной системы - например, через USB носитель.

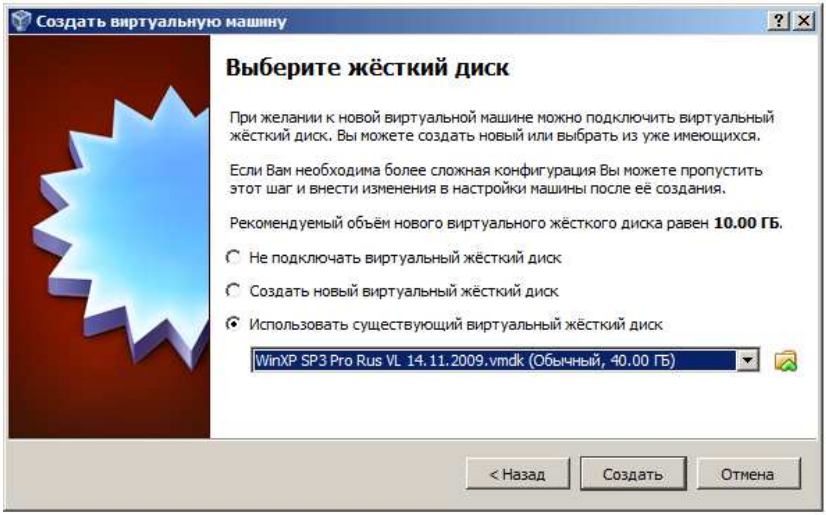


Рисунок 1.3 – Конфигурирование жесткого диска

После создания виртуальной машины на экране должна отображаться информация отраженная на рисунке 1.4.

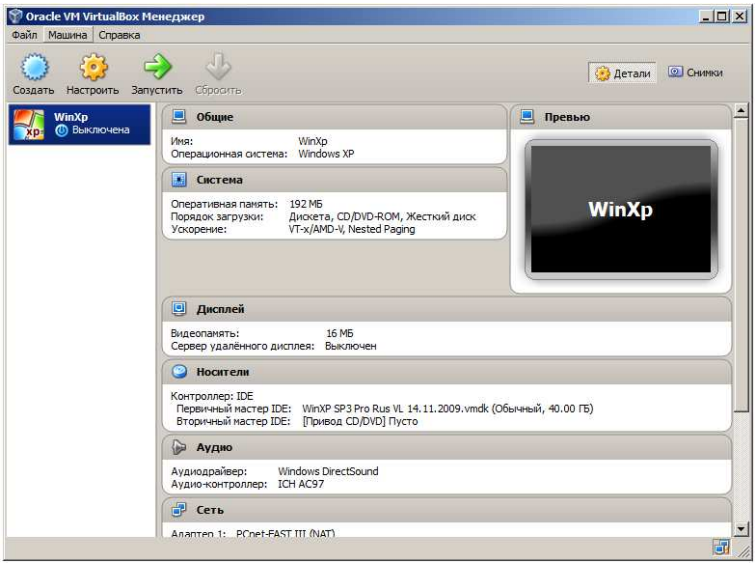


Рисунок 1.4 – Общий вид эмулятора

После установки операционной системы перейдите в меню "Настройки - Сеть". Вид открываемой вкладки соответствует настройкам по умолчанию и представлен на рисунке 1.5. Любая из виртуальных машин может быть настроена на использование четырех сетевых адаптеров - в зависимости от того, какой вам необходим в конкретном случае. Но чаще всего на практике требуется только один из них. Как правило, при установке виртуальной машины, по умолчанию создается простой сетевой адаптер. Этого достаточно для выхода в Интернет. В зависимости от потребностей, может понадобиться создание нескольких сетевых интерфейсов разных типов. Или же нескольких устройств одного типа, но с разными настройками. Это может потребоваться для использования на виртуальной машине как физических, так и виртуальных сетевых адаптеров. Все зависит от того, какие из них подключены.

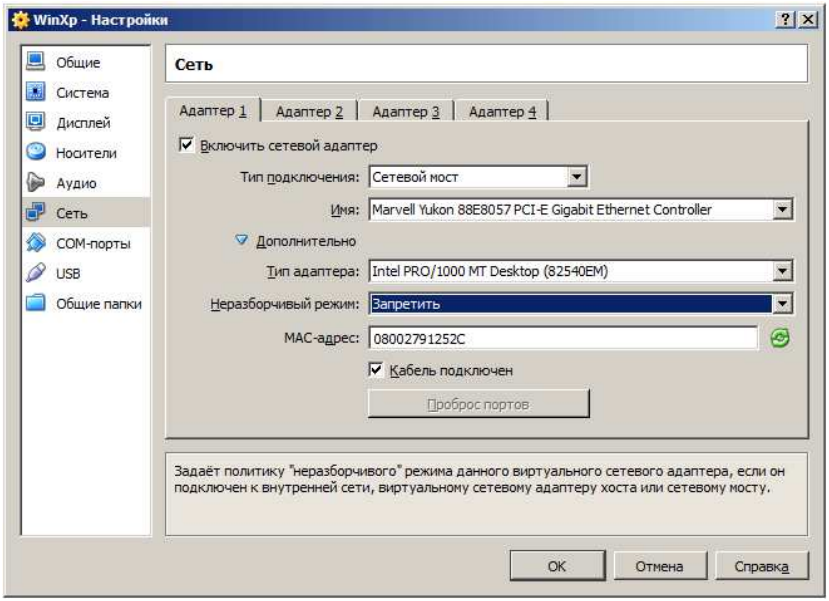


Рисунок 1.5. Управление сетевыми адаптерами виртуальной машины

**Тип адаптера (Adapter Type)**

Виртуальная машина VirtualBox имеет встроенную программную эмуляцию большинства наиболее распространенных типов сетевых карт, под которые созданы драйвера и протоколы. Карта PCnet-FAST III является выбором по умолчанию, однако можно выбрать и любую другую карту. Если возникают проблемы в настройке сетевого соединения, можно попробовать изменить тип адаптера, выбрав другой. Для наиболее старого оборудования подойдет сетевая карта PCnet-FAST II.

**Режим (Mode)**

"Неразборчивый режим" (Promiscuous Mode) обычно применяется для работы виртуальной машины в качестве виртуального маршрутизатора в локальных сетях; как сетевой мост или же хост. В этом режиме порт виртуальной машины способен принимать любые пакеты, отправляемые для других операционных систем и даже для хоста. То есть, принимаются сетевые пакеты, предназначенные не только для этого адаптера, но и для других сетевых устройств. Данный режим используется сетевыми администраторами для диагностики проблем, возникающих в сети.

**MAC адрес (MAC Address)**

MAC-адрес (MAC - аббревиатура от Media Access Control) является уникальным "именем" устройства в сети, однозначно идентифицирующим и отличающим его от остальных адаптеров и узлов. Этот адрес прописывается для каждого сетевого устройства на физическом уровне в памяти самого интерфейса. При создании виртуального сетевого адаптера VirtualBox автоматически генерирует для него MAC-адрес. Если необходимо изменить уже имеющийся MAC-адрес, то для этого служит небольшая кнопка справа, которая генерирует новое значение. В случае клонирования существующей виртуальной машины, для нее необходимо создать свой собственный уникальный MAC-адрес, который будет отличаться от адреса оригинальной машины.

Галочка напротив надписи "Кабель подключен" выполняет ту же роль, что и подключение или отключение физического кабеля в реальности. Эта настройка отвечает за подключение виртуального сетевого адаптера к сети. Не стоит путать ее с другой более важной настройкой "Включить сетевой адаптер",

которая включает или выключает сам адаптер на виртуальной машине.

Кнопка "Проброс портов" открывает диалоговое окно, в котором производится настройка правил поведения трафика на конкретном адаптере; каким образом будет перемещаться трафик определенного типа между хостом и гостевой виртуальной машиной. Эти правила применяются к сетевым моделям, которые будут рассмотрены немного позже. Сами сетевые модели определяются на вкладке "Тип подключения". Эта настройка является наиболее сложным моментом в установке соединений в VirtualBox.

**Типы подключения к сети**

В VirtualBox имеются четыре готовые модели для подключения к сети:

- Трансляция сетевых адресов (NAT), которая является настройкой по умолчанию
- Сетевой мост (Bridged)
- Виртуальный адаптер хоста (Host Only)
- Внутренняя сеть (Internal Network)

Соединение типа "Не подключен" также является настройкой сети, но служит только для одной цели - определения возможных неполадок. В этом режиме VirtualBox сообщает гостевой операционной системе, что сетевая карта присутствует, но соединения с ней нет.

**Трансляция сетевых адресов (NAT)**

Протокол NAT позволяет гостевой операционной системе выходить в Интернет, используя при этом частный IP, который не доступен со стороны внешней сети или же для всех машин локальной физической сети. Такая сетевая настройка позволяет посещать web-страницы, скачивать файлы, просматривать электронную почту. И все это, используя гостевую операционную систему. Однако извне невозможно напрямую соединиться с такой системой, если она использует NAT.

Принцип трансляции сетевых адресов заключается в следующем. Когда гостевая ОС отправляет пакеты на конкретный адрес удаленной машины в сети, сервис NAT, работающий под VirtualBox, перехватывает эти пакеты, извлекает из них сегменты, содержащие в себе адрес пункта отправки (IP-адрес гостевой

операционной системы) и производит их замену на IP-адрес машины-хоста. Затем заново упаковывает их и отправляет по указанному адресу.

Например, в вашей домашней локальной сети хост и другие физические сетевые устройства имеют адреса в диапазоне, начинающемся с 192.168.x.x. В VirtualBox адаптеры, работающие по протоколу NAT, имеют IP-адреса в диапазоне, начинающемся с 10.0.2.1 и заканчивающемся 10.0.2.24. Такой диапазон называется подсетью. Как правило, этот диапазон не используется для присвоения адресов устройствам в основной сети, поэтому такая система недоступна извне, со стороны хоста. Гостевая операционная система может выполнять обновление программного обеспечения и web-серфинг, но остается невидимой для остальных "участников".

Протокол NAT полезен в том случае, когда нет разницы в том, какие IP-адреса будут использовать гостевые операционные системы на виртуальной машине, поскольку все они будут уникальными. Однако, если потребуется настроить перенаправление сетевого трафика, или же расширить функциональность гостевой операционной системы, развернув на ней web-сервер (к примеру), то необходимы дополнительные настройки. В режиме NAT также недоступны такие возможности, как предоставление общего доступа к папкам и файлам.

**Сетевой мост (Bridged)**

В соединении типа "Сетевой мост" виртуальная машина работает также, как и все остальные компьютеры в сети. В этом случае адаптер выступает в роли моста между виртуальной и физической сетями. Со стороны внешней сети имеется возможность напрямую соединяться с гостевой операционной системой.

Адаптер в режиме "Сетевой мост" подключается, минуя хост, к устройству, которое распределяет IP-адреса внутри локальной сети для всех физических сетевых карт. VirtualBox соединяется с одной из установленных сетевых карт и передает пакеты через нее напрямую; получается работа моста, по которому передаются данные. Как правило, адаптер в модели "Сетевой мост" получает стандартный адрес из диапазона 192.168.x.x от роутера. Поэтому виртуальная машина в сети выглядит так, как

будто это обычное физическое устройство, неотличимое от остальных.

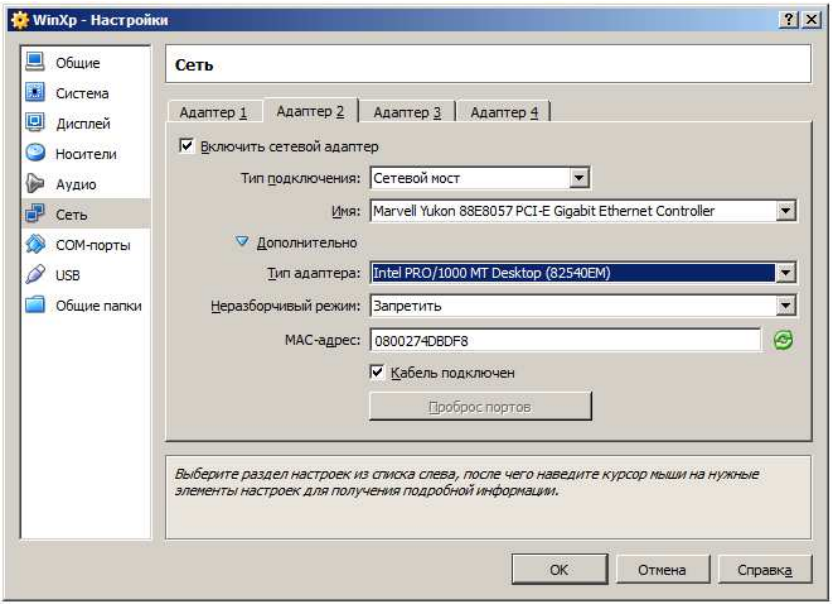


Рисунок 1.6. Настройка сетевого моста.

Поэтому моему хосту назначен роутером IP-адрес 192.168.0.2. Виртуальной машине в режиме "Сетевой мост" присвоен адрес 192.168.2.6. При этом не имеет значения тот факт, что VirtualBox передает и получает трафик как бы "сквозь" хост, минуя его. В результате получается, что виртуальная машина становится просто еще одним компьютером в локальной сети.

**Виртуальный адаптер хоста (Host-only)**

При подключении типа "Виртуальный адаптер хоста" гостевые операционные системы могут взаимодействовать между собой, а также с хостом. Но все это только внутри самой виртуальной машины VirtualBox. В этом режиме адаптер хоста использует свое собственное, специально для этого

предназначенное устройство, которое создается в глобальных настройказ VisrualBox (Файл\Настройки\Сеть) – рисунок 1.7.

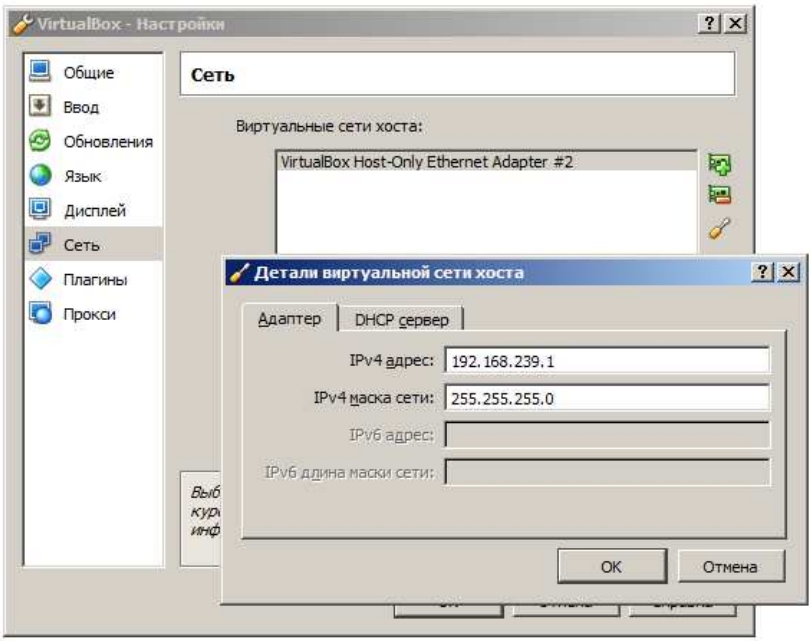


Рисунок 1.7. Настройка виртуальной сети хоста.

Также им создается подсеть и назначаются IP-адреса сетевым картам гостевых операционных систем. Гостевые операционные системы не могут взаимодействовать с устройствами, находящимися во внешней сети, так как они не подключены к ней через физический интерфейс. Режим "Виртуальный адаптер хоста" предоставляет ограниченный набор служб, полезных для создания частных сетей под VirtualBox для ее гостевых операционных систем.

В отличие от других продуктов виртуализации, адаптер, работающий под протоколом NAT в VirtualBox, не может выступать в роли связующего моста между сетевым устройством по умолчанию на хостах. Поэтому невозможен прямой доступ

извне к машинам, "спрятанным" за NAT - ни к программам, работающим на них; ни к данным, находящимся на самих хостах.

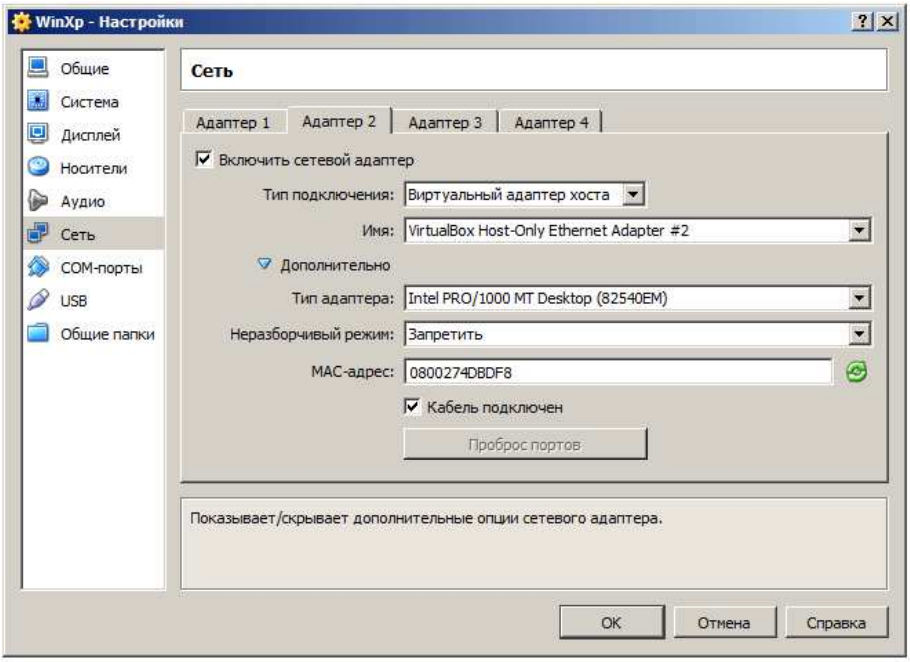


Рисунок 1.8. Настройка виртуального адаптера хоста.

Как правило, хост имеет свой собственный сетевой адрес, который используется для выхода в Интернет. Обычно это 192.168.0.101. В режиме "Виртуальный адаптер хоста" машина-хост также выступает в роли роутера VirtualBox и обладает IP-адресом по умолчанию 192.168.56.1. Создается внутренняя локальная сеть, обслуживающая все гостевые операционные системы, настроенные для режима "Виртуальный адаптер хоста" и видимые для остальной части физической сети. Подобно адаптеру в режиме "Сетевой мост", в режиме "Виртуальный адаптер хоста" используются разные диапазоны адресов. Можно легко настроить гостевые системы для получения IP-адресов, используя для этого встроенный DHCP-сервер виртуальной машины VirtualBox.

В дополнение нужно сказать, что в режиме "Виртуальный адаптер хоста" созданная им сеть не имеет внешнего шлюза для выхода в Интернет, как для хоста, так и для гостевых операционных систем. Он работает только как обычный сетевой коммутатор, соединяя между собой хост и гостевые системы. Поэтому адаптер в режиме "Виртуальный адаптер хоста" не предоставляет гостевым машинам выход в Интернет. Дополнительные возможности для этого адаптера значительно упрощают настройку сети между хостом и гостевыми операционными системами, однако все же отсутствует внешний доступ или перенаправление портов. Поэтому может потребоваться второй адаптер в режиме "Виртуальный адаптер хоста" или "Сетевой мост", который подключается к гостевой операционной системе для получения полного доступа к ней.

**Внутренняя сеть (Internal Network)**

Если на практике вам потребуется настроить взаимосвязь между несколькими гостевыми операционными системами, работающими на одном хосте и могущими сообщаться только между собой, тогда можно воспользоваться режимом "Внутренняя сеть". Конечно, для этой цели можно использовать режим "Сетевой мост", но режим "Внутренняя сеть" обладает большей безопасностью. В режиме "Сетевой мост" все пакеты отправляются и получаются через адаптер физической сети, установленный на машине-хосте. В этом случае весь трафик может быть перехвачен (например, путем установки сниффера пакетов на машине-хосте).

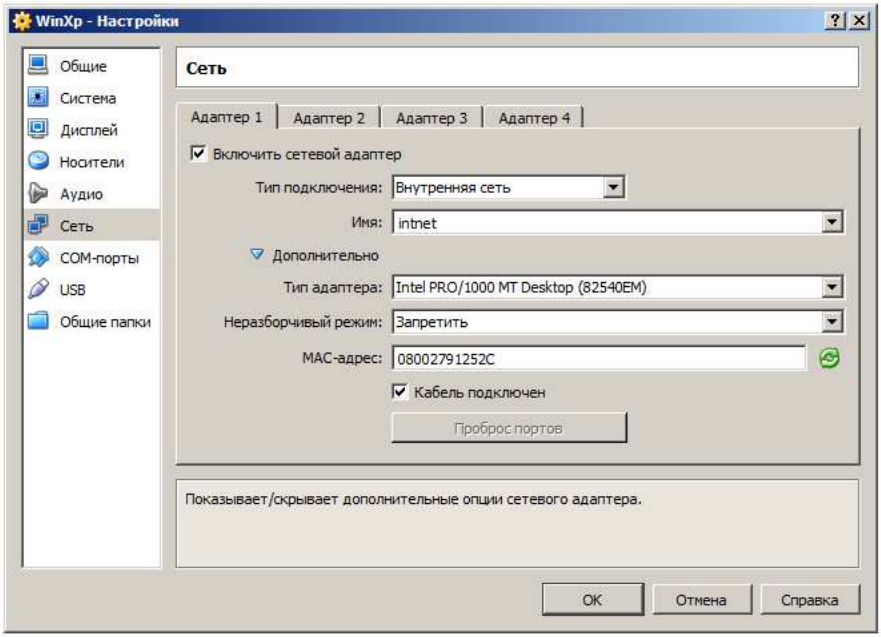


Рисунок 1.9. Настройка внутренней сети.

Внутренняя сеть, согласно руководству VirtualBox, является "программной сетью, которая может быть видима для выборочно установленных виртуальных машин, но не для приложений, работающих на хосте или на удаленных машинах, расположенных извне". Такая сеть представляет из себя набор из хоста и нескольких виртуальных машин. Но ни одно из вышеперечисленных устройств не имеет выхода через физический сетевой адаптер - он полностью программный, используемый VirtualBox в качестве сетевого маршрутизатора. В целом получается частная локальная сеть только для гостевых операционных систем без доступа в Интернет, что делает ее максимально безопасной. Возможное применение такой сети - сверхсекретный сервер с клиентами, предназначенный для разработки; тестирование систем на проникновение или какие-либо другие цели, преследующие создание внутренней сети для команд-разработчиков или организаций.

**Задания:**

1. Создать сеть из 2-х виртуальных машин с клиентскими ОС Windows. Продемонстрировать работоспособность системы.
2. Создать сеть из 3-х виртуальных машин с клиентскими ОС Windows. Продемонстрировать работоспособность системы.
3. Настроить работу в интернет-браузере запущенном на клиентской версии ОС Windows в виртуальной машине.
4. Настроить взаимодействие с сетью лаборатории на клиентской версии ОС Windows в виртуальной машине.
5. Настроить работу службы Avtive Directory на серверной версии ОС Windows в виртуальной машине. Продемонстрировать работу службы Active Directory, используя дополнительную виртуальную машину с клиентской ОС Windows.
6. Настроить работу службы Avtive Directory на серверной версии ОС Windows в виртуальной машине. Продемонстрировать работу службы Active Directory, используя две дополнительные виртуальные машины с клиентской ОС Windows.
7. Настроить работу службы Avtive Directory на серверной версии ОС Windows в виртуальной машине. Продемонстрировать работу службы DHCP, используя дополнительную виртуальную машину с клиентской ОС Windows.
8. Настроить работу службы Avtive Directory на серверной версии ОС Windows в виртуальной машине. Продемонстрировать работу службы DHCP, используя две дополнительные виртуальные машины с клиентской ОС Windows.
9. Настроить работу службы Avtive Directory на серверной версии ОС Windows в виртуальной машине. Продемонстрировать работу службы DNS, используя дополнительную виртуальную машину с клиентской ОС Windows.

10. Настроить работу службы Avtive Directory на серверной версии ОС Windows в виртуальной машине. Продемонстрировать работу службы DNS, используя две дополнительные виртуальные машины с клиентской ОС Windows.
11. * Создать сеть из 2-х виртуальных машин с использованием ОС Wndows и Linux.
12. * Создать сеть из 3-х виртуальных машин с использованием ОС Wndows и Linux.

## *Лабораторная работа №2. Разработка программ, взаимодействующих через Windows Sockets.*

**Цель работы:** научиться разрабатывать программы, использующие сетевой обмен информацией с использованием Windows Sockets.

Сокеты (sockets) представляют собой высокоуровневый унифицированный интерфейс взаимодействия с телекоммуникационными протоколами.

При разработке программ использующих сокеты полезно использовать - *Windows Sockets 2 SDK*. SDK - это документация, набор заголовочных файлов и инструментарий разработчика. Из инструментария, входящего в SDK, в первую очередь необходимо выделить утилиту sockeye.exe. Она позволяет в интерактивном режиме вызывать различные сокет-функции и манипулировать ими по своему усмотрению.

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 [3, 12].

Сокеты позволяют работать со множеством протоколов и являются удобным средством межпроцессорного взаимодействия, однако мы будем работать только с сокетами семейства протоколов TCP/IP, использующихся для обмена данными между узлами сети Интернет

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.

Для работы с библиотекой Winsock 2.x в исходный тест программы необходимо включить директиву "*#include <winsock2.h>*", а в командной строке компоновщика указать "ws2_32.lib". В среде разработки Microsoft Visual Studio для этого достаточно указать дерективу линкеру непосредственно в тексте программы:

#pragma comment(lib," ws2_32").

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. Если инициализация проваливается, функция возвращает ненулевое значение.

Далее необходимо создать объект "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.

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

Дальнейшие шаги при работе с сокетами, зависят от того, являет приложение сервером или клиентом.

Клиентское приложение для установки соединения с удаленным узлом, использующее потоковый сокет, должно 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.

Первый аргумент - дескриптор сокета, возвращенный функцией socket; второй - указатель на структуру "*sockaddr*", содержащую в себе адрес и порт удаленного узла с которым устанавливается соединение. 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.

В серверном приложении прежде, чем использовать сокет, его необходимо связать 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 вмести ними 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 аргументом передается дескриптор сокета, возвращенный функций socket, за ним следуют указатель на структуру sockaddr и ее длина (см. раздел "*Адрес раз, адрес два*").

Клиенти 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 использовать 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. Далее, после выполнения связывания, 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 осторожно. 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.

Датаграммные серверы не вызывают функцию listen, т.к. работают без установки соединения и сразу же после выполнения связывания могут вызывать recvfrom для чтения входящих сообщений, минуя следующие шаги.

Извлечение запросов на соединение из очереди сервера 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.

Далее поведение клиентского и серверного приложения не отличаются – 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 данные или нет. При успешном завершении функция возвращает количество *передаваемых* (*не переданных!*) данных - т. е. успешное завершение еще не свидетельствует от успешной

доставке. Протокол TCP 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. 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. 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 будет браться из локального буфера, а не приниматься из сети, т.к. TCP-провайдер не может получить "кусочек" дейтаграммы, а только целиком.

Работой обоих функций можно управлять с помощью *флагов*, передаваемых в одной переменной типа int третьим 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.

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 так же должен быть установлен.

Дейтаграммный сокет так же может пользоваться функциями send и recv, если предварительно вызовет connect (…), но у него 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 )*".

Данные функции сходны с 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 recvfrom, получившей сообщение от клиента.

Транспортный протокол UDP, на который опираются дейтаграммные сокеты, не гарантирует успешной доставки сообщений и эта задача ложиться на плечи 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, т. к. это рекурсивно неразрешимо. Поэтому - 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).

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.

Завершение процесса функцией ExitProcess автоматически не освобождает ресурсы сокетов, это может привести к утечке ресурсов в операционной системе.

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, наконец, 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.

Вызов shutdown не освобождает от необходимости закрытия сокета функцией closesocket.

Структура sockaddr_in, определяется 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
};
```

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-алрес
    } S_un;
  }
```

Как видно, она состоит из одного IP-адреса, записанного в трех формах - четырехбайтовой последовательности (S_un_b), пары двухбайтовых слов (S_un_W) и одного длинного целого (S_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: "*struc 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 приводит к ощибке. 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
};
```

Функция gethostbyname ожидает на входе только доменные имена, но не цифровые IP-адреса. Обычно же требуется предоставления клиенту возможности как задания доменных имен, так и цифровых IP-адресов [4].

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. Поскольку теоретически могут существовать имена доменов, синтаксически неотличимые от IP-адресов, то лучше всего действовать по следующему алгоритму: передаем введенную пользователем строку функции inet_addr, если она возвращает ошибку, то вызываем gethostbyaddr.

Для решения обратной задачи – определении 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.

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);*

**Внимание**: все значения, возвращенные socket-функциями уже находятся в сетевом формате и "вручную" их преобразовывать нельзя! Т.к. 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).*

**Дополнительные возможности**

Для более гибкой 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**.

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

**Уровень SOL_SOCKET:**

1. SO_RCVBUF (int) - задает размер входного буфера для приема данных.
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**

1. TCP_NODELAY (BOOL) - выключает *Алгоритм Нагла*. Алгоритм Нагла был разработан специально для прозрачного кэширования небольших 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 (достаточно экзотических) 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.

**Примеры программной реализации серверной и клиентской части.**

**Пример реализации TCP-сервера**

```
// Пример простого TCP – эхо сервера

  #include <stdio.h>
  #include <winsock2.h>  // Wincosk2.h должен быть
          // подключен раньше windows.h!
  #include <windows.h>

  #define MY_PORT     666
           // Порт, который слушает сервер

  // макрос для печати количества активных
  // пользователей
  #define PRINTNUSERS if (nclients)\
  printf("%d user on-line\n",nclients);\
  else printf("No User on line\n");

  // прототип функции, обслуживающий
  // подключившихся пользователей
  DWORD WINAPI SexToClient(LPVOID client_socket);

  // глобальная переменная – количество
```

```
// активных пользователей
int nclients = 0;

int main(int argc, char* argv[])
{
  char buff[1024];    // Буфер для различных нужд

  printf("TCP SERVER DEMO\n");

  // Шаг 1 - Инициализация Библиотеки Сокетов
  // Т.к. возвращенная функцией информация
  // не используется ей передается указатель на
  // рабочий буфер, преобразуемый
  // к указателю на структуру WSADATA.
  // Такой прием позволяет сэкономить одну
  // переменную, однако, буфер должен быть не менее
  // полкилобайта размером (структура WSADATA
  // занимает 400 байт)
  if (WSAStartup(0x0202,(WSADATA *) &buff[0]))
  {
    // Ошибка!
          printf("Error WSAStartup %d\n",
          WSAGetLastError());
    return -1;
  }

  // Шаг 2 - создание сокета
  SOCKET mysocket;
  // AF_INET     - сокет Интернета
  // SOCK_STREAM  - потоковый сокет (с
  //       установкой соединения)
  // 0       - по умолчанию выбирается TCP протокол
  if ((mysocket=socket(AF_INET,SOCK_STREAM,0))<0)
  {
    // Ошибка!
    printf("Error socket %d\n",WSAGetLastError());
    WSACleanup();
      // Деинициализация библиотеки Winsock
    return -1;
  }

  // Шаг 3 связывание сокета с локальным адресом
  sockaddr_in local_addr;
```

```
local_addr.sin_family=AF_INET;
local_addr.sin_port=htons(MY_PORT);
         // не забываем о сетевом порядке!!!
local_addr.sin_addr.s_addr=0;
         // сервер принимает подключения
         // на все IP-адреса

// вызываем bind для связывания
if (bind(mysocket,(sockaddr *) &local_addr,
           sizeof(local_addr)))
{
  // Ошибка
  printf("Error bind %d\n",WSAGetLastError());
  closesocket(mysocket;  // закрываем сокет!
  WSACleanup();
  return -1;
}

// Шаг 4 ожидание подключений
// размер очереди – 0x100
if (listen(mysocket, 0x100))
{
  // Ошибка
  printf("Error listen %d\n",WSAGetLastError());
  closesocket(mysocket);
  WSACleanup();
  return -1;
}

printf("Ожидание подключений\n");

// Шаг 5 извлекаем сообщение из очереди
SOCKET client_socket;    // сокет для клиента
sockaddr_in client_addr;    // адрес клиента
          // (заполняется системой)

// функции accept необходимо передать размер
// структуры
int client_addr_size=sizeof(client_addr);

// цикл извлечения запросов на подключение из
// очереди
while((client_socket=accept(mysocket, (sockaddr *)
```

```
            &client_addr, &client_addr_size)))
    {
      nclients++;       // увеличиваем счетчик
              // подключившихся клиентов

      // пытаемся получить имя хоста
      HOSTENT *hst;
      hst=gethostbyaddr((char *)
          &client_addr.sin_addr.s_addr,4, AF_INET);

      // вывод сведений о клиенте
      printf("+%s [%s] new connect!\n",
      (hst)?hst->h_name:"",
      inet_ntoa(client_addr.sin_addr));
      PRINTNUSERS

      // Вызов нового потока для обслуживания клиента
      // Да, для этого рекомендуется использовать
      // _beginthreadex но, поскольку никаких вызов
      // функций стандартной Си библиотеки поток не
      // делает, можно обойтись и CreateThread
      DWORD thID;
      CreateThread(NULL, NULL, SexToClient,
                &client_socket, NULL, &thID);
    }
    return 0;
}

// Эта функция создается в отдельном потоке и
// обсуживает очередного подключившегося клиента
// независимо от остальных
DWORD WINAPI SexToClient(LPVOID client_socket)
{
    SOCKET my_sock;
    my_sock=((SOCKET *) client_socket)[0];
    char buff[20*1024];
    #define sHELLO "Hello, Sailor\r\n"

    // отправляем клиенту приветствие
    send(my_sock, sHELLO, sizeof(sHELLO), 0);

    // цикл эхо-сервера: прием строки от клиента и
    // возвращение ее клиенту
```

```
  while( (int bytes_recv=
              recv(my_sock,&buff[0],sizeof(buff),0))
          && bytes_recv !=SOCKET_ERROR)
    send(my_sock,&buff[0],bytes_recv,0);

  // если мы здесь, то произошел выход из цикла по
  // причине возращения функцией recv ошибки –
  // соединение клиентом разорвано
  nclients--; // уменьшаем счетчик активных клиентов
  printf("-disconnect\n"); PRINTNUSERS

  // закрываем сокет
  closesocket(my_sock);
  return 0;
}
```

**Пример реализации TCP-клиента**

```
  // Пример простого TCP клиента
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>


#define PORT 666
#define SERVERADDR "127.0.0.1"

int main(int argc, char* argv[])
{
  char buff[1024];
  printf("TCP DEMO CLIENT\n");

  // Шаг 1 - инициализация библиотеки Winsock
  if (WSAStartup(0x202,(WSADATA *)&buff[0]))
  {
    printf("WSAStart error %d\n",WSAGetLastError());
    return -1;
  }

  // Шаг 2 - создание сокета
```

```
SOCKET my_sock;
my_sock=socket(AF_INET,SOCK_STREAM,0);
if (my_sock < 0)
{
  printf("Socket() error %d\n",WSAGetLastError());
  return -1;
}

// Шаг 3 - установка соединения

// заполнение структуры sockaddr_in
// указание адреса и порта сервера
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 сервера
  if (hst=gethostbyname(SERVERADDR))
  // hst->h_addr_list содержит не массив адресов,
  // а массив указателей на адреса
  ((unsigned long *)&dest_addr.sin_addr)[0]=
    ((unsigned long **)hst->h_addr_list)[0][0];
  else
  {
    printf("Invalid address %s\n",SERVERADDR);
    closesocket(my_sock);
    WSACleanup();
    return -1;
  }

// адрес сервера получен – пытаемся установить
// соединение
if (connect(my_sock,(sockaddr *)&dest_addr,
            sizeof(dest_addr)))
{
  printf("Connect error %d\n",WSAGetLastError());
```

```
    return -1;
  }

  printf("Соединение с %s успешно установлено\n\
  Type quit for quit\n\n", SERVERADDR);

  // Шаг 4 - чтение и передача сообщений
  int nsize;
  while((nsize=recv(my_sock, &buff[0],
                    sizeof(buff)-1, 0))
                !=SOCKET_ERROR)
  {
    // ставим завершающий ноль в конце строки
    buff[nsize]=0;

    // выводим на экран
    printf("S=>C:%s", buff);

    // читаем пользовательский ввод с клавиатуры
    printf("S<=C:"); fgets(&buff[0],sizeof(buff)-1,
           stdin);

    // проверка на "quit"
    if (!strcmp(&buff[0],"quit\n"))
    {
      // Корректный выход
      printf("Exit...");
      closesocket(my_sock);
      WSACleanup();
      return 0;
    }

    // передаем строку клиента серверу
    send(my_sock, &buff[0], nsize, 0);
  }

  printf("Recv error %d\n", WSAGetLastError());
  closesocket(my_sock);
  WSACleanup();
  return -1;
}
```

**Пример реализации UDP-сервера**

```
// Пример простого UDP-эхо сервера
  #include <stdio.h>
  #include <winsock2.h>

  #define PORT 666    // порт сервера
  #define sHELLO "Hello, %s [%s] Sailor\n"

  int main(int argc, char* argv[])
  {
    char buff[1024];

    printf("UDP DEMO echo-Server\n");

    // шаг 1 - подключение библиотеки
    if (WSAStartup(0x202,(WSADATA *) &buff[0]))
    {
      printf("WSAStartup error: %d\n",
              WSAGetLastError());
      return -1;
    }

    // шаг 2 - создание сокета
    SOCKET my_sock;
    my_sock=socket(AF_INET,SOCK_DGRAM,0);
    if (my_sock==INVALID_SOCKET)
    {
      printf("Socket() error: %d\n", WSAGetLastError());
      WSACleanup();
      return -1;
    }

    // шаг 3 - связывание сокета с локальным адресом
    sockaddr_in local_addr;
    local_addr.sin_family=AF_INET;
    local_addr.sin_addr.s_addr=INADDR_ANY;
    local_addr.sin_port=htons(PORT);

    if (bind(my_sock,(sockaddr *) &local_addr,
        sizeof(local_addr)))
    {
```

```
    printf("bind error: %d\n", WSAGetLastError());
    closesocket(my_sock);
    WSACleanup();
    return -1;
  }

  // шаг 4 обработка пакетов, присланных клиентами
  while(1)
  {
    sockaddr_in client_addr;
    int client_addr_size = sizeof(client_addr);
    int bsize=recvfrom(my_sock, &buff[0],
      sizeof(buff)-1, 0,
      (sockaddr *) &client_addr, &client_addr_size);
    if (bsize==SOCKET_ERROR)
    printf("recvfrom() error: %d\n",
           WSAGetLastError());

    // Определяем IP-адрес клиента и прочие атрибуты
    HOSTENT *hst;
    hst=gethostbyaddr((char *)
          &client_addr.sin_addr, 4, AF_INET);
    printf("+%s [%s:%d] new DATAGRAM\n",
    (hst)?hst->h_name:"Unknown host",
    inet_ntoa(client_addr.sin_addr),
    ntohs(client_addr.sin_port));

    // добавление завершающего нуля
    buff[bsize]=0;

    // Вывод на экран
    printf("C=>S:%s\n", &buff[0]);

    // посылка датаграммы клиенту
    sendto(my_sock, &buff[0], bsize, 0,
      (sockaddr *)&client_addr, sizeof(client_addr));
  }
  return 0;
}
```

**Пример реализации UDP-клиента**

```
// пример простого UDP-клиента
  #include <stdio.h>
  #include <string.h>
  #include <winsock2.h>
  #include <windows.h>

  #define PORT 666
  #define SERVERADDR "127.0.0.1"

  int main(int argc, char* argv[])
  {
    char buff[10*1014];
    printf("UDP DEMO Client\nType quit to quit\n");

    // Шаг 1 - инициализация библиотеки Winsocks
    if (WSAStartup(0x202,(WSADATA *)&buff[0]))
    {
      printf("WSAStartup error: %d\n",
             WSAGetLastError());
      return -1;
    }

    // Шаг 2 - открытие сокета
    SOCKET my_sock=socket(AF_INET, SOCK_DGRAM, 0);
    if (my_sock==INVALID_SOCKET)
    {
      printf("socket() error: %d\n", WSAGetLastError());
      WSACleanup();
      return -1;
    }

    // Шаг 3 - обмен сообщений с сервером
    HOSTENT *hst;
    sockaddr_in dest_addr;

    dest_addr.sin_family=AF_INET;
    dest_addr.sin_port=htons(PORT);

    // определение IP-адреса узла
    if (inet_addr(SERVERADDR))
```

```
  dest_addr.sin_addr.s_addr=inet_addr(SERVERADDR);
else
  if (hst=gethostbyname(SERVERADDR))
    dest_addr.sin_addr.s_addr=((unsigned long **)
          hst->h_addr_list)[0][0];
else
  {
    printf("Unknown host: %d\n",WSAGetLastError());
    closesocket(my_sock);
    WSACleanup();
    return -1;
  }

while(1)
{
  // чтение сообщения с клавиатуры
  printf("S<=C:");fgets(&buff[0],sizeof(buff)-1,
        stdin);
  if (!strcmp(&buff[0],"quit\n")) break;

  // Передача сообщений на сервер
  sendto(my_sock,&buff[0],strlen(&buff[0]),0,
    (sockaddr *) &dest_addr,sizeof(dest_addr));

  // Прием сообщения с сервера
  sockaddr_in server_addr;
  int server_addr_size=sizeof(server_addr);

  int n=recvfrom(my_sock,&buff[0],sizeof(buff)-1,0,
    (sockaddr *) &server_addr, &server_addr_size);

  if (n==SOCKET_ERROR)
  {
    printf("recvfrom() error:"\
      "%d\n",WSAGetLastError());
    closesocket(my_sock);
    WSACleanup();
    return -1;
  }

  buff[n]=0;

  // Вывод принятого с сервера сообщения на экран
```

```
        printf("S=>C: %s", &buff[0]);
    }

    // шаг последний - выход
    closesocket(my_sock);
    WSACleanup();

    return 0;
}
```

**Проверка 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 вашего узла (это специально зарезервированный для этой цели адрес и он выглядит одинаково для всех узлов), а 666 - номер порта на котором был установлен сервер. Если все работает успешно, то telnet установит соединение и на экране появится приветствие "Hello, Sailor!". Теперь можно набирать на клавиатуре некоторый текст и получать его назад от сервера.

**Проверка работоспособности TCP-клиента**: запустите TCP-сервер и затем одну или несколько копий клиента. В каждом из них можно набирать некоторый текст на клавиатуре и после нажатия на Enter получать его обратно от сервера.

**Проверка работоспособности UDP сервера и клиента**: запустите UDP-сервер и одну или несколько копий клиента - в каждой из них можно набирать на клавиатуре некоторые текстовые сообщения, и после нажатия клавиши Enter получать их обратно в ответ от сервера.

### Задания:

1. Разработать TCP сервер, который по запросу клиента возвращает содержимое указанной папки в файловой системе.
2. Разработать UDP сервер, который по запросу клиента возвращает содержимое указанной папки в файловой системе.
3. Разработать TPC сервер, который по запросу одного клиента, копирует указанный файл из файловой системы другого клиента.

4. Разработать UDP сервер, который по запросу одного клиента, копирует указанный файл из файловой системы другого клиента.
5. Разработать TPC сервер, который по запросу клиента архивирует файл и возвращает результат клиенту.
6. Разработать UDP сервер, который по запросу клиента архивирует файл и возвращает результат клиенту.
7. Разработать TPC сервер, который по запросу клиента сохраняет на сервере некоторую информацию, а в последствии по запросу возвращает сохраненную информацию (обеспечить случай одновременного сохранения различной информации).
8. Разработать UDP сервер, который по запросу клиента сохраняет на сервере некоторую информацию, а в последствии по запросу возвращает сохраненную информацию (обеспечить случай одновременного сохранения различной информации).
9. Разработать TPC сервер, который по запросу клиента может управлять своими подключениями (вывести список активных подключений, время последней активности, удалить подключение).
10. Разработать UDP сервер, который по запросу клиента может управлять своими подключениями (вывести список активных подключений, время последней активности, удалить подключение).
11. * Разработать TPC сервер реализующий протокол POP3.
12. * Разработать TPC сервер реализующий протокол SMTP.

## *Лабораторная работа №3. Установка и настройка FTP сервера. Написание FTP клиента.*

**Цель работы:** приобрести навыки конфигурирования службы FTP в ОС семейства Microsoft Windows, ознакомиться с подмножеством функций библиотеки WinINet для работы с FTP сервером.

**Установка и настройка сервера FTP (File Transfer Protocol) в ОС Microsoft Windows.**

Установка служб IIS и FTP

Служба FTP зависит от служб IIS (Internet Information Services). Чтобы установить службы IIS и FTP, выполните следующие действия.

Примечание: В Windows служба FTP не устанавливается по умолчанию одновременно со службами IIS. Если службы IIS уже установлены, воспользуйтесь для установки службы FTP компонентом «Установка и удаление программ» панели управления [7].

1. В меню **Пуск** выберите пункт **Панель управления** и запустите компонент **Установка и удаление программ**.
2. Нажмите кнопку **Установка компонентов Windows**.
3. В списке **Компоненты** выберите пункт **Сервер приложений**, затем — **Службы IIS** (но не меняйте состояния флажка) и нажмите кнопку **Состав**.
4. Установите следующие флажки (если они не установлены) :

   **Общие файлы**
   **Служба FTP**
   **Диспетчер служб IIS**

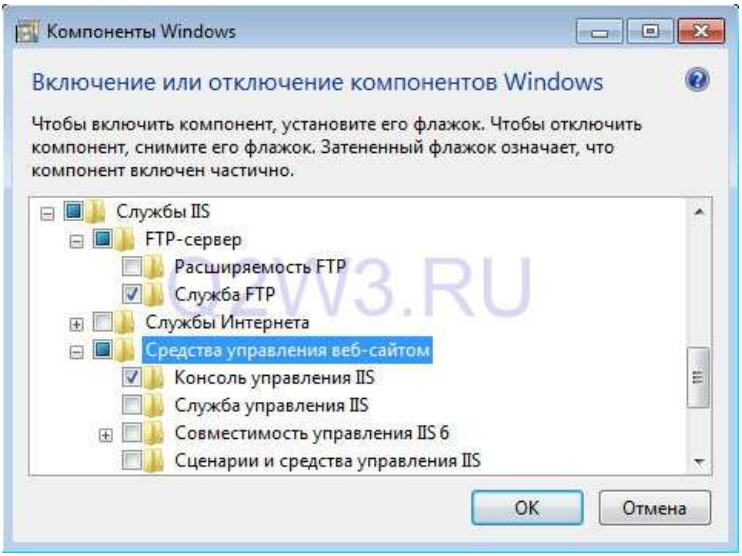


Рисунок 3.1. Установка компонент Windows

5. Установите флажки других необходимых компонентов или служб и нажмите кнопку **ОК**.
6. Нажмите кнопку **Далее**.
7. В ответ на соответствующий запрос вставьте компакт-диск с дистрибутивом Windows или укажите путь к месту расположения файлов и нажмите кнопку **ОК**.
8. Нажмите кнопку **Готово**.

Службы IIS и FTP установлены. Перед началом использования службы FTP ее необходимо настроить.

**Настройка службы FTP**

Чтобы настроить службу FTP на прием анонимных подключений, выполните следующие действия:

1. Запустите «Диспетчер служб IIS» или откройте оснастку IIS.
2. Разверните компонент ***имя_сервера***, где *имя_сервера* — имя сервера.
3. Разверните компонент **Узлы FTP**.

4. Щелкните правой кнопкой мыши элемент **FTP-узел по умолчанию** и выберите пункт **Свойства**.
5. Производим настройку службы.

Сервер FTP готов принимать входящие запросы FTP. Скопируйте или переместите файлы, к которым следует открыть доступ, в папку публикации FTP. По умолчанию используется папка *диск*:\Inetpub\Ftproot, где *диск* — это диск, на котором установлены службы IIS.
Следующий этап — настройка брандмауэра Windows. Откройте Панель управления -> Система и безопасность -> Брандмауэр Windows -> Дополнительные параметры. В разделе «Правила для входящих подключений» находим и активируем «FTP-сервер (входящий трафик)» и «FTP Server Passive (FTP Passive Traffic-In)». Последнее правило позволяет подключаться ftp-клиенту в пассивном режиме.

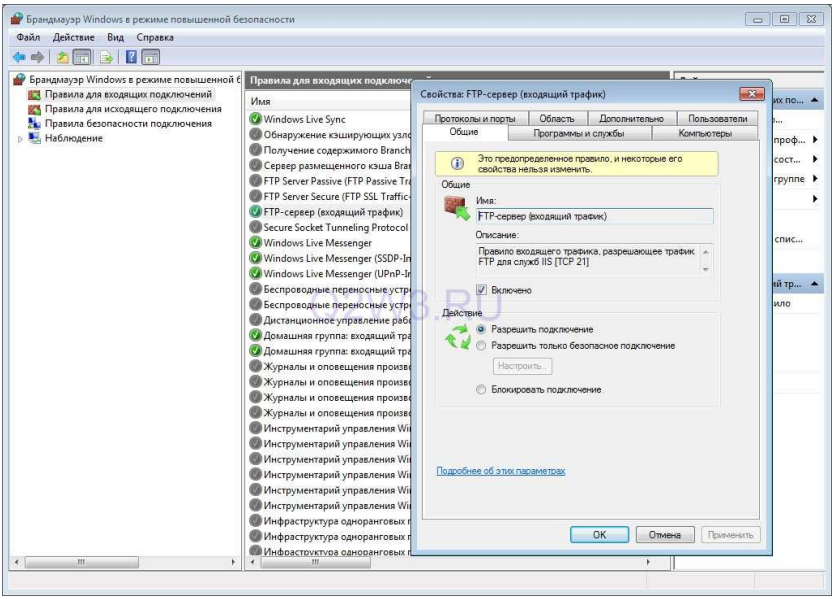


Рисунок 3.2. Настройка входящих подключений

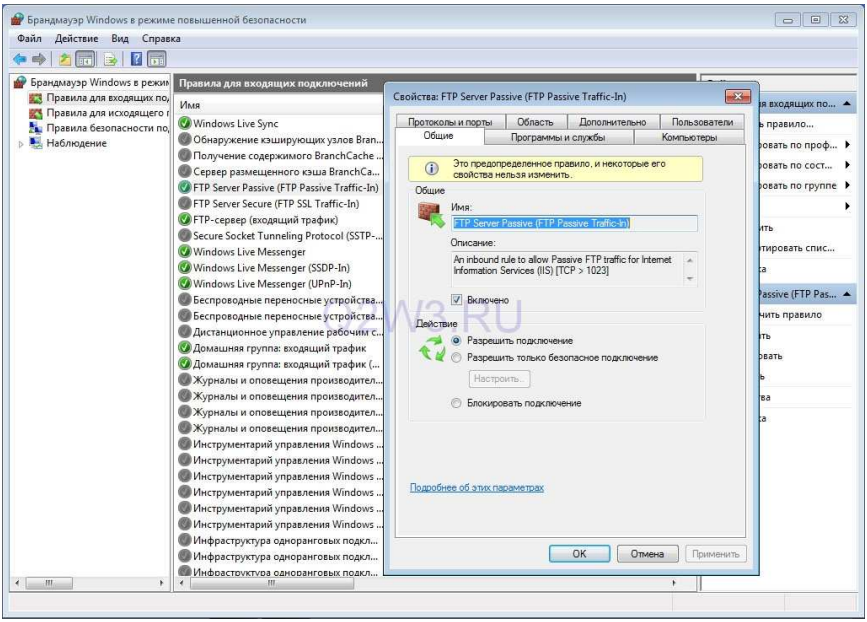


Рисунок 3.3 Настройка пассивных подключений

В разделе «Правила для исходящего подключения» находим и активируем «FTP Server (FTP Traffic-Out)».

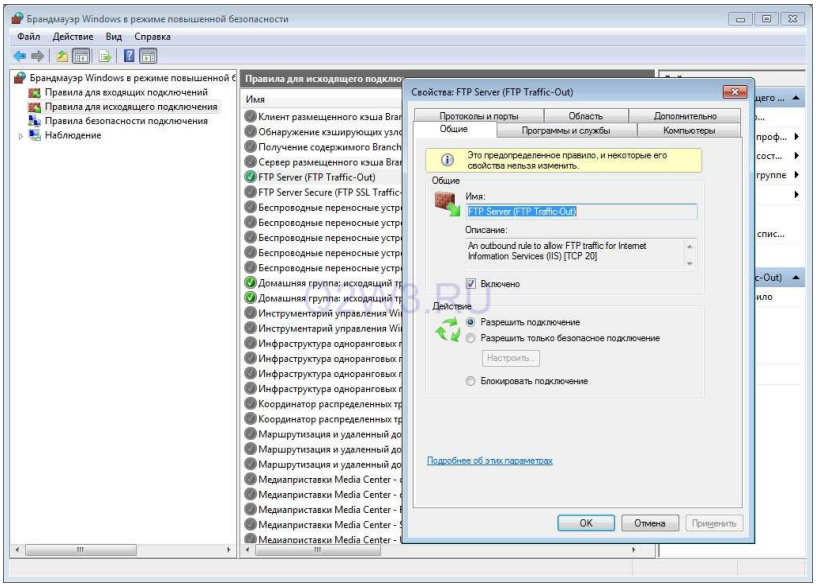


Рисунок 3.4. Настройка исходящих подключений

Если в системе установлен дополнительный брандмауэр, то в нем также необходимо открыть порт 21 (TCP) для входящих подключений и порт 20 (TCP) для исходящих.

Работа с FTP с использованием библиотеки WinINet.

Для работы с функциями, реализующими доступ к FTP прежде всего следует установить интернет-соединение (на самом деле оно нужно для использования большинства функций из библиотеки WinINet) [5,6]. Это достигается посредством последовательного вызова функций InternetOpen(…) и InternetConnect(…). Описание функций приводится ниже.

```
HINTERNET InternetOpen(
  _In_  LPCTSTR lpszAgent,
  _In_  DWORD dwAccessType,
  _In_  LPCTSTR lpszProxyName,
  _In_  LPCTSTR lpszProxyBypass,
  _In_  DWORD dwFlags
);
```

*Параметры:*

***lpszAgent***

Указатель на строку, которая указывает на имя приложения или сущности, вызывающего функции WinINet. Это имя используется как пользовательский агент в протоколе HTTP.

***dwAccessType*** Тип требующегося доступа. Этот параметр может принимать одно из перечисленных в таблице значений:

| **Символьная константа** | **Dec-значение** | **Описание** |
|---|---|---|
| INTERNET_OPEN_TYPE_DIRECT | 1 | Разрешает имена всех хостов локально. |
| INTERNET_OPEN_TYPE_PRECONFIG | 0 | Запрашивает прокси или прямую конфигурацию из регистра. |
| INTERNET_OPEN_TYPE_PRECONFIG_WITH_NO_AUTOPROXY | 4 | Запрашивает проки или прямую конфигурацию из регистра и предотвращает использование стартовых Microsoft JScript или Internet Setup (INS) файлов. |
| INTERNET_OPEN_TYPE_PROXY | 3 | Направляет запрос прокси-серверу, в случае если не предоставлен лист обходов прокси-серверов и имени прокси-сервера нет списке прокси, которые можно обойти. В этом случае функция использует INTERNET_OPEN_TYPE_DIRECT. |

Таблица 3.1. Значения параметра dwAccessType.

***lpszProxyName***

Указатель на строку с терминатором CHR(0), указывает имя прокси-сервера (серверов) для использования с соединением, когда доступ через прокси-сервер определен установкой параметра dwAccessType в значение INTERNET_OPEN_TYPE_PROXY. Не используйте пустую строку, так как функция InternetOpen в таком случае будет использовать ее как имя прокси-сервера. Фукнции WinINet распознают только зафиксированные CERN типы прокси (только HTTP) и TIS FTP шлюзы (только FTP). Если в системе инсталлирован Microsoft Internet Explorer, эти функции, кроме того, поддерживают SOCKS - прокси. Запросы к FTP и Gopher могут быть выполнены через определенные CERN типы либо путем изменения их в запрос HTTP или путем использования функции InternetOpenUrl. Если параметр dwAccessType не установлен в значение INTERNET_OPEN_TYPE_PROXY, то этот параметр игнорируется и должен быть установлен в NULL.

***lpszProxyBypass***

Указатель на строку с терминатором CHR(0), который определяет необязательный списко имен хостов или IP адресов, или тех и других, запросы к которым не должны отправляться через прокси для случая, когда значение параметра dwAccessType установлено в INTERNET_OPEN_TYPE_PROXY. Список может содержать модификаторы в виде звездочки. Не используйте пустую строку в качестве параметра, так как InternetOpen будет использовать ее как список обхода прокси-серверов. Если этот параметр указан как макро "<local>" в виде единственного члена списка, функция будет обходить любые имена хостов, которые не имеют в своем имени точки. Если же значение параметра dwAccessType не установлено в INTERNET_OPEN_TYPE_PROXY, то этот параметр будет игнорирован и и его значение должно быть установлено в NULL.

***dwFlags***

Опции. Этот параметр может представлять собой комбинацию следующих значений

| Символьная константа | Dec-значение | Описание |
|---|---|---|
| INTERNET_FLAG_ASYNC | 1 | Выполняет только асинхронные запросы на дескрипторах по убывающей от дескриптора возвращенного этой функцией. |
| INTERNET_FLAG_FROM_CACHE | 2 | Не выполняет сетевых запросов. Все сущности возвращаются из кэша. Если запрашиваемый элемент не находится в кэше, то возвращается соответствующая ошибка - ERROR_FILE_NOT_FOUND. |
| INTERNET_FLAG_OFFLINE | 3 | Идентично INTERNET_FLAG_FROM_CACHE. Не выполняет сетевых запросов. Все сущности возвращаются из кэша. Если запрашиваемый элемент не находится в кэше, то возвращается соответствующая ошибка - ERROR_FILE_NOT_FOUND. |

Таблица 3.2. Значения параметра *dwFlags.*

**Возвращаемое значение**

Возвращает действительный дексриптор, который направляется приложениям ко всем последующим функциям WinINet. Если вызов функции не получился InternetOpen, то

возвращается NULL. Для получения описания соответствующего сообщения об ошибке используйте функцию WinAPI GetLastError.

InternetOpen является первой WinINet, которая должна быть вызвана приложением. Она сообщает Internet DLL о необходимости инициализировать внутренние структуры данных и позволяет подготовиться для последующих вызовов из приложения. Когда приложение завершает использование функций Internet, оно должно вызвать функцию WinAPI InternetCloseHandle для освобождения дескриптора и любых связанных ресурсов.

Приложение может выполнять любое количество функции InternetOpen, хотя, обычно, достаточно единичного вызова. Приложению может потребоваться определить раздельное поведение каждого вызова InternetOpen, такое, как, например, конфигурирование прокси-серверов для каждой из них. После того, как приложение завершит использование вызовов функций, использующих дескриптор HINTERNET, возвращенный вызовом InternetOpen, он должен быть закрыт с помощью функции InternetCloseHandle.

```
HINTERNET InternetConnect(
  _In_  HINTERNET hInternet,
  _In_  LPCTSTR lpszServerName,
  _In_  INTERNET_PORT nServerPort,
  _In_  LPCTSTR lpszUsername,
  _In_  LPCTSTR lpszPassword,
  _In_  DWORD dwService,
  _In_  DWORD dwFlags,
  _In_  DWORD_PTR dwContext
);
```

Открывает сессию File Transfer Protocol (FTP), Gopher или HTTP для данного сайта.

*Параметры*

***nInet_Handle***

Дексриптор текущей Internet сессии. Дескриптор должен быть получен в результате вызова функции InternetOpen.

***lpcServer***

Указатель на строку с нуль-терминатором, которая определяет имя

хоста Интернет сервера. Альтернативно строка может содержать IP адрес сайта, в ASCII формате (например, 11.0.1.45).

***nPort***

Порт Transmission Control Protocol/Internet Protocol (TCP/IP) на сервере. Эти флаги устанавливаются только на используемый порт. Сервис устанавливается с помощью значения nService. Этот параметр может принимать одно из приведенных ниже значений.

| **Символьная константа** | **Dec-значение** | **Описание** |
|---|---|---|
| INTERNET_DEFAULT_FTP_PORT | 21 | Использовать порт по-умолчанию для FTP серверов (port 21). |
| INTERNET_DEFAULT_GOPHER_PORT | 70 | Использовать порт по-умолчанию для Gopher серверов (port 70). |
| INTERNET_DEFAULT_HTTP_PORT | 80 | Использовать порт по-умолчанию для HTTP серверов (port 80). |
| INTERNET_DEFAULT_HTTPS_PORT | 443 | Использовать порт по-умолчанию для Secure Hypertext Transfer Protocol (HTTPS) серверов (port 443). |
| INTERNET_DEFAULT_SOCKS_PORT | 1080 | Использовать порт по-умолчанию для SOCKS firewall серверов (port 1080). |
| INTERNET_INVALID_PORT_NUMBER | 0 | Использовать порт по-умолчанию для сервиса, определеямого с помощью значения nService. |

Таблица 3.3. Значения параметра nPort

***lpcUserName***

Указатель на строку с нуль-терминатором, которая определяет имя пользователя для регистрации. Если этот параметр установлен в NULL, то функция использует подходящее по умолчанию значение, за исключением случая с HTTP; NULL параметр для HTTP приведет к тому, что сервер вернет ошибку. Для FTP протокола, значением по умолчанию является "anonymous".

***lpcPassword***
Указатель на строку с нуль-терминатором, которая содержит пароль для целей регистрации. Если оба - и lpcPassword, и lpcUsername равны NULL, функция использует пароль по умолчанию "anonymous". В случае FTP, паролем по умолчанию является имя e-mail пользователя. Если lpcPassword равено NULL, а lpcUsername им не является, функция использует пустой пароль.
***nService***
Тип сервиса для доступа. Этот параметр может иметь одно из приведенных ниже значений.

| Символьная константа | Dec-значение | Описание |
|---|---|---|
| INTERNET_SERVICE_FTP | 1 | FTP сервис. |
| INTERNET_SERVICE_GOPHER | 2 | Gopher сервис. |
| INTERNET_SERVICE_HTTP | 3 | HTTP сервис. |

Таблица 3.4. Значение параметра nService.

***nFlags***
Опции, характерные для используемого сервиса. Если nService равен INTERNET_SERVICE_FTP, INTERNET_FLAG_PASSIVE заставляет приложение использовать семантику пассивного FTP.
***nContext***
Указатель на переменную, которая содержит определяемое приложением значение, которое используется для идентификации контекста приложения для возвращаемого обработчика в обратных вызовах.
***Возвращаемое значение***
Возвращает действительный указатель к сессии FTP, Gopher или HTTP, если удалось соединиться с сервером; в противно случае вызов функции вернет NULL. Для получения расширенной информации об ошибке нужно вызвать функцию GetLastError. Кроме того, приложение может использовать функцию InternetGetLastResponseInfo для причины - почему доступ к сервису был отклонен.

***Примечания:***

Для FTP сайтов, InternetConnect устанавливает реальное соединение с сервером; для других, таких как Gopher, реальное соединение с сервером не устанавливается до тех пор, пока приложение не затребует характерную для соединения транзакцию.

Для обеспечения максимальной эффективности, приложения, использующие протоколы Gopher и HTTP должны стараться минимизировать вызовы фукнции InternetConnect и избегать вызовов этой функции для каждой транзакции, запрашиваемой пользователем. Одним из путей для осуществления этого является хранения небольшого кэша указателей, возвращаемых из вызовов функции InternetConnect. Когда пользователь делает запрос к серверам, к которым он ранее получал доступ, этот указатель сессии будет оставаться для него доступным.

После того, как работа приложения, использующего указатель, полученный в результате вызова InternetConnect, будет завершена, до своего закрытия оно должно закрыть его с помощью функции InternetCloseHandle.

```
BOOL FtpCreateDirectory(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszDirectory
);
```

Создает новый каталог на FTP-сервере.

**Параметры**

***nConnect_Handle***

Указатель на служебную структуру, полученный в результате вызова функции InternetConnect с использованием параметра nService, установленного в значение INTERNET_SERVICE_FTP.

***lpcDirectory***

Указатель на строку с нуль-терминатором, которая содержит имя каталога, который должен быть создан на FTP-сервере. В качестве имени может быть использован либо полностью квалифицированный путь, либо путь, относительно текущего каталога.

***Возвращаемое значение:***

В случае, если операция создания каталога завершилась успешно, то будет возвращен TRUE, в противном случае - FALSE. В случае ошибки, для получения полного сообщения об ошибке, вызовите GetLastError. Если сообщение об ошибке показывает, что FTP-сервер отклонил запрос на создание каталога, вызовите функцию InternetGetLastResponseInfo для определения причины.

***Примечание***

Приложение должно использовать функцию FtpGetCurrentDirectory для определения текущего рабочего каталога, вместо того, чтобы подразумевать, что удаленная система использует иерархическую схему наименований для каталогов.

```
BOOL FtpDeleteFile(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszFileName
);
```

Удаляет файл, хранящийся на FTP-сервере.

**Параметры**

***nConnect_Handle***

Указатель на служебную структуру, полученный в результате вызова функции InternetConnect с использованием параметра nService, установленного в значение INTERNET_SERVICE_FTP.

***lpcFileName***

Указатель на строку заканчивающуюся нулевым символом, которая содержит имя файла, подлежащего удалению.

***Возвращаемые значение***

В случае, если операция удаления файла завершилась успешно, то будет возвращен TRUE, в противном случае - FALSE. В случае ошибки, для получения подробного сообщения об ошибке, вызовите GetLastError.

```
HINTERNET FtpFindFirstFile(
  _In_   HINTERNET hConnect,
  _In_   LPCTSTR lpszSearchFile,
  _Out_  LPWIN32_FIND_DATA lpFindFileData,
  _In_   DWORD dwFlags,
  _In_   DWORD_PTR dwContext
);
```

Ищет указанный каталог в данной FTP сессии. Файл и каталог возвращаются приложению в виде структуру WIN32_FIND_DATA.

**Параметры**

***nConnect_Handle***

Указатель на служебную структуру, полученный в результате вызова функции InternetConnect с использованием параметра nService, установленного в значение INTERNET_SERVICE_FTP (1)

***lpcSearchStr***

Указатель на строку с нуль-терминатором, которая указывает действительный путь каталога или имя файла для файловой системы FTP-сервера. Строка может содержать модификаторы (*), однако в имени не допускаются пробелы. Если значение lpcSearchStrIf равно NULL или, если оно представляет собой пустую строку, функция ищет первый файл в текущем каталоге на сервере.

***lpcWIN32_FIND_DATA***

Указатель на структуру WIN32_FIND_DATA, которая получает информацию о найденном файле или каталоге. Структура приведена ниже:

```
typedef struct _WIN32_FIND_DATA {
  DWORD    dwFileAttributes;
  FILETIME ftCreationTime;
  FILETIME ftLastAccessTime;
  FILETIME ftLastWriteTime;
  DWORD    nFileSizeHigh;
  DWORD    nFileSizeLow;
  DWORD    dwReserved0;
  DWORD    dwReserved1;
  TCHAR    cFileName[MAX_PATH];
  TCHAR    cAlternateFileName[14];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA,
*LPWIN32_FIND_DATA;
```

Значения основных атрибутов файла приведены в таблице 3.5.

| Символьная константа | Dec-значение | Описание |
|---|---|---|
| FILE_ATTRIBUTE_ARCHIVE | 32 | Признак включения в архив. Приложения используют атрибут для резервного копирования файлов. |
| FILE_ATTRIBUTE_COMPRESSED | 2048 | Файл или директория подвергнут динамическому сжатию. |
| FILE_ATTRIBUTE_DIRECTORY | 16 | Значение определяет директорию. |
| FILE_ATTRIBUTE_ENCRYPTED | 16384 | Файл или директория зашифрован. |
| FILE_ATTRIBUTE_HIDDEN | 2 | Файл или директория скрыт. |
| FILE_ATTRIBUTE_NORMAL | 128 | Файл не содержит других атрибутов. Не может присутствовать в комбинации с другими атрибутами. |
| FILE_ATTRIBUTE_READONLY | 1 | Файл используется только для чтения. |
| FILE_ATTRIBUTE_SYSTEM | 4 | Файл или директория являются частью операционной системы. |

Таблица 3.5. Значения атрибутов файла.

Не все файловые системы могут фиксировать время создания и последнего доступа к файлу в едином формате. Например, на FAT, время создания файла имеет гранулярность в 10 миллисекунд, время записи - 2 секунды, а время доступа - 1 день (реально - дата доступа). На NTFS, время последнего доступа к файлу имеет гранулярность в 1 час

***nFlags***

Управляет поведением этой функции. Этот параметр может быть комбинацией следующих значений:

- INTERNET_FLAG_HYPERLINK (0x00000400)
- INTERNET_FLAG_NEED_FILE (0x00000010)
- INTERNET_FLAG_NO_CACHE_WRITE (0x04000000)
- INTERNET_FLAG_RELOAD (0x80000000)
- INTERNET_FLAG_RESYNCHRONIZE (0x00000800)

***nContext***

Указатель на переменную, которая определяет определенное приложением значение, которая связывает этот поиск с любыми данными приложения. Этот параметр используется только в том случае, если приложение уже вызвало InternetSetStatusCallback для установки статуса callback функции.

***Возвращаемое значение***

Возвращает указатель на служебную структуру запроса, если перечисление каталога началось успешно, в противно случае возвращает NULL. Для получения подробного сообщения об ошибке, вызовите GetLastError. Если GetLastError возвращает ERROR_INTERNET_EXTENDED_ERROR (12003), как в случае, если функция не нашла файла, отвечающего запрошенному имени, вызовите функцию InternetGetLastResponseInfo для получения расширенного текста ошибки, как это было уже было сказано ранее.

***Примечания***

Для функции FtpFindFirstFile, параметры файла, связанные со временем, возврщаемые в структуре WIN32_FIND_DATA указываются в местной временной зоне, а не в формате UTC.

Функция FtpFindFirstFile схожа с функцией FindFirstFile. Отметим, однако, что только одна функция FtpFindFirstFile может быть вызвана только во время данной FTP сессии. Функцию нельзя вызывать повторно до закрытия перечисления файлов и каталогов.

Это вызвано тем, что протокол FTP допускает только одно перечисление каталога на сессию.
После вызова функции FtpFindFirstFile и до момента вызова InternetCloseHandle, приложение не может вызвать FtpFindFirstFile повторно на данной FTP сессии. Если повторный вызов FtpFindFirstFile все-таки сделан, то функция завершается с ошибкой ERROR_FTP_TRANSFER_IN_PROGRESS (12110). При завершении приложения, использовавшего перечисление, возвращенное вызовом FtpFindFirstFile, оно должно закрыть его с помощью вызова функции InternetCloseHandle.
После успешного начала перечисления каталога с помощью функции FtpFindFirstFile, для продолжения перечисления может быть использована функция InternetFindNextFile.
Поскольку протокол FTP не предоставляет стандартных средств перечисления (enumerating), некоторая информация о файлах, такая как времена создания и доступа могут быть не всегда доступны или корректны. Когда это случается, FtpFindFirstFile и InternetFindNextFile заполняют места недоступной информации предположительным подсчетом на основе другой доступной информации. Например, даты создания и последнего доступа часто такие же, как и дата изменения файла.
Приложение не может вызвать функцию FtpFindFirstFile между вызовами функций FtpOpenFile и InternetCloseHandle.

```
BOOL InternetFindNextFile(
  _In_   HINTERNET hFind,
  _Out_  LPVOID lpvFindData
);
```

Продолжает поиск файла, начиная с результата, полученного в результате предудыщего вызова InternetFindNextFile или начатого вызовом функций FtpFindFirstFile или GopherFindFirstFile.
*Параметры*
***nConnect_Handle***
Указатель, полученный в результате вызова функции InternetConnect.
***lpcWIN32_FIND_DATA***
Указатель на буфер, (для FTP - это структура WIN32_FIND_DATA), который получает информацию о найденном файле или каталоге. (Структура приведена в описании

предыдущей функции - FtpFindFirstFile). В случае протокола Gopher используется структура GOPHER_FIND_DATA.

***Возвращаемое значение:***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError. Если функция не нашла соответствующего файла, то GetLastError вернет ERROR_NO_MORE_FILES.

```
BOOL FtpGetCurrentDirectory(
  _In_     HINTERNET hConnect,
  _Out_    LPTSTR lpszCurrentDirectory,
  _Inout_  LPDWORD lpdwCurrentDirectory
);
```

Возвращает информацию о текущем каталоге для данной FTP-сессии.

*Параметры*

***nConnect_Handle***

Указатель на служебную структуру, полученный в результате вызова функции InternetConnect.

***lpcDirectory***

Указатель на строку, которая получит абсолютный путь текущего каталога.

***nMax_Path***

Указатель на переменную, которая определяет длину буфера в символах TCHARs. Длина буфера должна учитывать пространство для символа окончания строки. Использование в качестве длины константы MAX_PATH достаточно для всех путей. При возврате функция заполняет переменную числом, определяющим количество символов, записанных в буфер.

***Возвращаемое значение:***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError.

***Примечание***

Если буфер lpcDirectory недостаточно велик, в nMax_Path будет записано число байтов, требуемых для сохранения полного имени текущего каталога.

```
BOOL FtpGetFile(
```

```
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszRemoteFile,
  _In_  LPCTSTR lpszNewFile,
  _In_  BOOL fFailIfExists,
  _In_  DWORD dwFlagsAndAttributes,
  _In_  DWORD dwFlags,
  _In_  DWORD_PTR dwContext
);
```

Возвращает файл с FTP-сервера и сохраняет его под указанным именем, создавая в процессе новый локальный файл.

**Параметры**

***nConnect_Handle***

Указатель на служебную структуру идентифицирующую FTP-сессию.

***lpcRemoteFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя запрашиваемого файла.

***lpcNewFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя файла, который будет создан на локальной системе.

***nFailIfExists***

Указывает, должна ли функция выполнять операцию, если локальный файл с указанным именем уже существует. Если параметр установлен в TRUE и локальный файл уже существует, функция завершится с ошибкой.

***nAttributes***

Аттрибуты для нового файла. Этот параметр может быть комбинацией флагов FILE_ATTRIBUTE_*, используемых функцией WinAPI CreateFile.

***nFlags***

Управляет тем, как функция будет обрабатывать загрузку с сервера. Первый набор значений флага указывает условия, при которых происходит перенос файла. Эти транспортные флаги могут быть использованы в комбинации со вторым набором значений, который управляет кэшированием. Приложение может выбрать одно из этих значений для переноса файла на локальную систему.

| Символьная константа | Hex-значение | Описание |
|---|---|---|
| FTP_TRANSFER_TYPE_ASCII | 0x00000001 | Передавать файл с использованием ASCII символов (с перекодировкой). |
| FTP_TRANSFER_TYPE_BINARY | 0x00000002 | Передавать файл, как двоичный поток данных |
| FTP_TRANSFER_TYPE_UNKNOWN | 0x00000000 | По умолчанию как FTP_TRANSFER_TYPE_BINARY. |

Таблица 3.6. Значения атрибутов nFlags.

Приведенные ниже флаги (таблица 3.7) определяют - как будет производиться кэширование файла. Любая комбинация из приведенных ниже флагов может быть использована с флагом типа переноса файла.

***nContext***

Указатель на переменную, которая определяет заданное приложением значение, которая связывает этот поиск с любыми данными приложения. Этот параметр используется только в том случае, если приложение уже вызвало InternetSetStatusCallback для установки статуса callback функции.

***Возвращаемые значение***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError.

***Примечания***

FtpGetFile представляет собой процедуру высокого уровня, которая обрабатывает все вычисления и накладные расходы, связанные с чтением файла из FTP-сервера и сохранением его локально. Приложение, которому нужны только данные о файле, или которому нужен больший контроль над переносом файла, должно использовать функции FtpOpenFile и InternetReadFile.

Если параметр nFlags установлен в FILE_TRANSFER_TYPE_ASCII, трансляция данных файла преобразует управляющие и форматирующие символы в локальные эквиваленты. Переносом по умолчанию является бинарный режим, где загруженный файл хранится в том же виде, каким он был на сервере.
Оба параметра, и lpcRemoteFile, и lpcNewFile имени могут иметь частично или полностью квалифицированные имена относительно текущего каталога.

| Символьная константа | Hex-значение | Описание |
|---|---|---|
| INTERNET_FLAG_HYPERLINK | 0x00000400 | Вызывает презагрузку, если в возврате с сервера не имеется срока истечения действительности документа (Expires) и нет флага LastModified, когда производится определение - необходима - ли загрузка элемента с сети. |
| INTERNET_FLAG_NEED_FILE | 0x00000010 | Служит основанием для создания временного файла, если файл не может быть кэширован. |
| INTERNET_FLAG_RELOAD | 0x80000000 | Принуждает загрузку запрашиваемого файла, объекта или листинга каталога с сервера, а не из кэша. |
| INTERNET_FLAG_RESYNCHRONIZE | 0x00000800 | Перезагружает HTTP ресурсы, если ресурс был изменен с момента времени последней загрузки. Все ресурсы FTP и Gopher перезагружаются. |

Таблица 3.7. Значения атрибутов nFlags, отвечающие за кэширования.

```
HINTERNET FtpOpenFile(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszFileName,
  _In_  DWORD dwAccess,
  _In_  DWORD dwFlags,
  _In_  DWORD_PTR dwContext
);
```

Инициирует доступ к удаленному файлу на FTP-сервере для чтения или записи.

*Параметры*

***nConnect_Handle***

Указатель на служебную структуру идентифицирующий FTP-сессию.

***lpcRemoteFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя запрашиваемого файла.

***nAccessType***

Тип доступа к файлу. Этот параметр может принимать значение GENERIC_READ (0x80000000) или GENERIC_WRITE (0x40000000). Эти значения не могут комбинироваться.

***nFlags***

См. описание предыдущей функции - FtpGetFile

***nContext***

Указатель на переменную, которая определяет определенное приложением значение, которая связывает этот поиск с любыми данными приложения. Этот параметр используется только в том случае, если приложение уже вызвало InternetSetStatusCallback для установки статуса callback функции.

***Возвращаемое значение***

Возвращает указатель на файл, если вызов завершился успешно, в противном случае NULL. Для получения расширенной информации об ошибке вызовите GetLastError.

***Примечания***

После вызова функции FtpOpenFile и до вызова InternetCloseHandle, все другие вызовы FTP функций на том же самом указателе FTP сессии будут завершаться с ошибкой, и устанавливать сообщение об ошибке в

ERROR_FTP_TRANSFER_IN_PROGRESS (12110). При завершении приложения, которое использовало указатель, полученный в результате вызова FtpOpenFile, он должен быть закрыт приложением с помощью функции InternetCloseHandle.
В течение одной FTP сессии может быть одновременно открыт только один файл.

```
BOOL FtpPutFile(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszLocalFile,
  _In_  LPCTSTR lpszNewRemoteFile,
  _In_  DWORD dwFlags,
  _In_  DWORD_PTR dwContext
);
```

Сохраняет файл на FTP сервере.

*Параметры*

***nConnect_Handle***

Указатель на служебную структуру, идентифицирующий FTP-сессию.

***lpcLocalFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя файла, который будет послан с локальной системы на удаленную систему (FTP сервер).

***lpcNewRemoteFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя файла, который будет создан на удаленной системе.

***nFlags***

Аналогично флагам, указанным в описании функции FtpGetFile

***nContext***

Указатель на переменную, которая определяет определенное приложением значение, которая связывает этот поиск с любыми данными приложения. Этот параметр используется только в том случае, если приложение уже вызвало InternetSetStatusCallback для установки статуса callback функции.

***Возвращаемое значение***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError.

***Примечания***

FtpPutFile представляет собой процедуру высокого уровня, которая обрабатывает все вычисления и накладные расходы, связанные с чтением файла на локальной системе и сохранением его на FTP-сервере. Приложение, которому нужны только данные о файле, или которому нужен больший контроль над переносом файла, должно использовать функции FtpOpenFile и InternetWriteFile.

Если параметр nFlags установлен в FILE_TRANSFER_TYPE_ASCII, трансляция данных файла преобразует управляющие и форматирующие символы в локальные эквиваленты. Переносом по умолчанию является бинарный режим, где загруженный файл хранится в том же виде, каким он был на сервере.

Оба параметра, и lpcNewRemoteFile, и lpcLocalFile имени могут иметь частично или полностью квалифицированные имена относительно текущего каталога.

```
BOOL FtpRemoveDirectory(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszDirectory
);
```

Удаляет указанную директорию на FTP сервере.

*Параметры*

***nConnect_Handle***

Указатель на служебную структуру, идентифицирующий FTP сессию.

***lpcDirectory***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя подлежащего удалению каталога. Имя может быть как частично, так и полностью квалифицированным относительно текущего каталога.

***Возвращаемое значение***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError. Если сообщение об

ошибке указывает, что FTP-сервер отклонил запрос на удаление каталога, вызовите функцию InternetGetLastResponseInfo для определения причины.

***Примечание***

Приложение должно использовать функцию FtpGetCurrentDirectory для определения текущего каталога удаленного сайта, вместо того, чтобы подразумевать, что удаленная система использует иерархическую схему наименований для каталогов.

```
BOOL FtpRenameFile(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszExisting,
  _In_  LPCTSTR lpszNew
);
```

Переименовывает файл, хранящийся на FTP сервере.

*Параметры*

***nConnect_Handle***

Указатель на служебную структуру, идентифицирующий FTP сессию.

***lpcRemoteFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя подлежащего переименованию файла.

***lpcNewFile***

Указатель на строку, завершающуюся нулевым символом, которая содержит новое имя файла.

***Возвращаемое значение***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError.

```
BOOL FtpSetCurrentDirectory(
  _In_  HINTERNET hConnect,
  _In_  LPCTSTR lpszDirectory
);
```

Данная функция устанавливает текущую директорию на FTP сервере

*Параметры*

***nConnect_Handle***

Указатель на служебную структуру, идентифицирующий FTP сессию.

***lpcDirector***

Указатель на строку, завершающуюся нулевым символом, которая содержит имя каталога, который станет текущим рабочим каталогом. Параметр lpcDirectory может быть как частично, так и полностью квалифицированным именем относительно текущего каталога.

***Возвращаемое значение***

Возвращает TRUE, если вызов завершился успешно, в противном случае FALSE. Для получения расширенной информации об ошибке вызовите GetLastError. Если сообщение об ошибке указывает, что FTP-сервер отклонил запрос на изменение текущего каталога, вызовите функцию InternetGetLastResponseInfo для определения причины.

***Примечание***

Приложение должно использовать функцию FtpGetCurrentDirectory для определения текущего каталога удаленного сайта, вместо того, чтобы подразумевать, что удаленная система использует иерархическую схему наименований для каталогов.

Пример программы для работы с функциями FTP библиотеки WinINet с использованием Windows API.

```
#include "stdafx.h"
#include <windows.h>
#include <wininet.h>
#pragma comment(lib,"wininet")

void ShowError(void) {
        DWORD dwRetVal;
        dwRetVal = GetLastError();
        printf("\nFailed to Perform Operation.\nError Code:
%u\n", dwRetVal);
        DWORD dwInetError;
        DWORD dwExtLength = 1000;
```

```
        TCHAR *szExtErrMsg = NULL;
        TCHAR errmsg[1000];
        szExtErrMsg = errmsg;
        int returned = InternetGetLastResponseInfo(
&dwInetError, szExtErrMsg, &dwExtLength );
        printf("dwInetError: %d  Returned: %d\n", dwInetError,
returned);
        _tprintf(_T("Buffer: %s\n"), szExtErrMsg);
}

int _tmain(int argc, _TCHAR* argv[])
{
        HINTERNET hInternetSession, hFTPConect, hFindFile;
    hInternetSession = InternetOpen(TEXT("FTPExample"),
                                        INTERNET_OPEN_TYPE_PRECON
                                        FIG, NULL, NULL, 0);
        if (hInternetSession == NULL) {
                ShowError();
                return -1;
        }
    hFTPConect =
                                    InternetConnect(hInternetSes
                                    sion, TEXT("80.82.59.168"),
                                    INTERNET_DEFAULT_FTP_PORT,
                                    TEXT("anonymous"),
                                    TEXT(""),
                                    INTERNET_SERVICE_FTP, 0, 0);
        if (hFTPConect == NULL) {
                ShowError();
            InternetCloseHandle(hInternetSession);
                return -1;
        }
        WIN32_FIND_DATA FileData;
        hFindFile = FtpFindFirstFile(hFTPConect, TEXT("*.*"),
&FileData, 0, NULL);
        _tprintf(_T("%s\n"), FileData.cFileName);
        InternetCloseHandle(hFindFile);
        InternetCloseHandle(hFTPConect);
        InternetCloseHandle(hInternetSession);
        return 0;
}
```

Почему функция ShowError() выводит только код ошибки?

К сожалению функция Windows API FormatMessage при помощи которой можно получить текстовые описания ошибок по их коду не поддерживает коды ошибок, используемые функциями для работы с Internet. Поэтому единственный способ получить текстовое описание ошибки – посмотреть ее описание в wininet.h. При этом следует учесть, что ошибки там описаны следующим образом:

```
#define INTERNET_ERROR_BASE                    12000

#define ERROR_INTERNET_OUT_OF_HANDLES
(INTERNET_ERROR_BASE + 1)
#define ERROR_INTERNET_TIMEOUT
(INTERNET_ERROR_BASE + 2)
#define ERROR_INTERNET_EXTENDED_ERROR
(INTERNET_ERROR_BASE + 3)
#define ERROR_INTERNET_INTERNAL_ERROR
(INTERNET_ERROR_BASE + 4)
```

соответственно, прямой поиск по коду ошибки не даст требуемого результата.

**Задания:**

1. Создать FTP сервер. Создать пользователя с полными правами на сервере. Определить текущий директорий. Создать в нем заданный пользователем директорий. Отработать ситуацию существования создаваемого директория. Удалить созданный директорий (ранее существовавший не удалять!)
2. Создать FTP сервер. Создать пользователя с полными правами на сервере. Установить заданный пользователем текущий директорий и переименовать его. Если директорий не существует - создать его. Создать директорий по заданию пользователя. Отработать ситуацию существования создаваемого директория. Сделать его текущим. Восстановить прежний текущий директорий. Удалить созданный директорий (ранее существовавший не удалять!)

3. Создать FTP сервер. Создать пользователя с правами только на чтение на сервере. Вывести содержимое текущего директория, упорядоченное по именам файлов (использовать данные из WIN32_FIND_DATA)
4. Создать FTP сервер. Создать пользователя с правами только на чтение на сервере. Скопировать содержимое указанного директория FTP сайта на локальный носитель.
5. Создать FTP сервер. Создать пользователя с правами только на запись на сервере. Скопировать содержимое указанного директория с локального носителя на FTP сайт.
6. Создать FTP сервер. Разрешить только анонимный вход на сервер. Установить заданный пользователем текущий директорий и вывести его содержимое, упорядоченное по убыванию размера файлов. Восстановить прежний текущий директорий
7. Создать FTP сервер. Создать пользователя с правами на запись и чтение на сервере. Определить текущий директорий. Создать в нем заданный пользователем директорий. Отработать ситуацию существования создаваемого директория. Удалить созданный директорий (ранее существовавший не удалять!)
8. Создать FTP сервер. Создать пользователя с правами на запись и чтение на сервере. Установить заданный пользователем текущий директорий и переименовать его. Если директорий не существует - создать его.
9. Создать FTP сервер. Разрешить только анонимный вход на сервер. Вывести содержимое существовавшего ранее директория, упорядоченное по времени доступа к файлам (в порядке убывания)
10. Создать FTP сервер. Создать пользователя с правами на запись и чтение на сервере. Создать директорий по заданию пользователя. Отработать ситуацию существования создаваемого директория. Сделать его текущим. Если директорий существовал - вывести его содержимое, упорядоченное по убыванию размера файлов. Восстановить прежний текущий директорий. Удалить созданный директорий (ранее существовавший не удалять!)

11. Создать FTP сервер. Разрешить только анонимный вход на сервер. Вывести содержимое текущего директория, упорядоченное по времени создания файлов (использовать данные из WIN32_FIND_DATA)
12. Создать FTP сервер. Разрешить только анонимный вход на сервер. Вывести содержимое указанного директория, упорядоченное по времени доступа к файлам (использовать данные из WIN32_FIND_DATA)
13. * Реализовать программу, аналогичную редактору notepad, которая позволяла бы работать с файлами, расположенными на FTP непосредственно из программы.
14. * Разработать Файловый менеджер для FTP.
15. * Разработать программу, реализующую API высокого уровня для работы с FTP, работая только через библиотеку Winsock. Описание протокола - http://www.ietf.org/rfc/rfc959.txt.

## *Лабораторная работа №4. Изучение протоколов канального уровня. Разработка программы передачи данных с использованием протокола канального уровня.*

**Цель работы:** разобраться в основах работы протоколов канального уровня передачи данных. Разработать эмулятор физического уровня передачи данных для изучения работы канальных протоколов.

### Основные концепции протоколов сетевого уровня.

Для предоставления сервиса сетевому уровню уровень передачи данных должен использовать сервисы, предоставляемые ему физическим уровнем. Физический уровень принимает необработанный поток битов и пытается передать его по назначению. Этот поток не застрахован от ошибок. Количество принятых бит может быть меньше, равно или больше числа переданных бит; кроме того, значения принятых битов могут отличаться от значений переданных. Уровень передачи данных должен обнаружить ошибки и, если нужно, исправить их [2].

Обычно уровень передачи данных разбивает поток битов на отдельные кадры и считает для каждого кадра контрольную сумму. Когда кадр прибывает в пункт назначения, его контрольная сумма подсчитывается снова. Если она отличается от содержащейся в кадре, то уровень передачи данных понимает, что при передаче кадра произошла ошибка, и принимает меры (например, игнорирует испорченный кадр и посылает передающей машине сообщение об ошибке).

Разбиение потока битов на отдельные кадры представляет собой более сложную задачу, чем это может показаться на первый взгляд. Один из способов разбиения на кадры заключается во вставке временных интервалов между кадрами,

подобно тому, как вставляются пробелы между словами в тексте. Однако сети редко предоставляют гарантии сохранения временных параметров при передаче данных, поэтому возможно,

что эти интервалы при передаче исчезнут или, наоборот, будут добавлены новые интервалы.

Поскольку для отметки начала и конца кадра полагаться на временные параметры слишком рискованно, были разработаны другие методы. Рассмотрим четыре метода маркировки границ кадров:

1. Подсчет количества символов.

2. Использование сигнальных байтов с символьным заполнением.

3. Использование стартовых и стоповых битов с битовым заполнением.

4. Использование запрещенных сигналов физического уровня.

Первый метод формирования кадров использует поле в заголовке для указания количества символов в кадре. Когда уровень передачи данных на принимающем компьютере видит это поле, он узнает, сколько символов последует, и таким образом определяет, где находится конец кадра.

Недостаток такой системы в том, что при передаче может быть искажен сам счетчик. Например, если размер второго кадра из числа 5 станет из-за ошибки в канале числом 7, то принимающая машина потеряет синхронизацию и не сможет обнаружить начало следующего кадра. Даже если контрольная сумма не совпадет (скорее всего) и принимающий компьютер поймет, что кадр принят неверно, то он все равно не сможет определить, где начало следующего кадра. Запрашивать повторную передачу кадра также бесполезно, поскольку принимающий компьютер не знает, сколько символов нужно пропустить до начала повторной передачи. По этой причине метод подсчета символов практически не применяется.

Второй метод формирования кадров решает проблему восстановления синхронизации после сбоя при помощи маркировки начала и конца каждого кадра специальными байтами. В прошлом стартовые и стоповые байты отличались друг от друга, но в последнее время большинство протоколов перешло на использование в обоих случаях одного и того же байта, называемого флаговым. Таким образом, если приемник теряет синхронизацию, ему необходимо просто найти флаговый байт, с

помощью которого он распознает конец текущего кадра. Два соседних флаговых байта говорят о том, что кончился один кадр и начался другой.

Этот метод иногда приводит к серьезным проблемам при передаче бинарных данных, таких как объектные коды программ или числа с плавающей запятой. Вполне вероятно, что в передаваемых данных запросто может встретиться последовательность, используемая в качестве флагового байта. Возникновение такой ситуации, скорее всего, собьет синхронизацию. Одним из способов решения проблемы является добавление специального escape-символа (знака переключения кода, ESC) непосредственно перед случайно совпавшим с флаговым байтом внутри кадра. Уровень передачи данных получателя вначале убирает эти escape-символы, затем передает кадр на сетевой уровень. Такой метод называется символьным заполнением. Таким образом, настоящий флаг можно отличить от «подложного» по наличию или отсутствию перед ним ESC.

Следующий логичный вопрос: а что, если и символ ESC случайно окажется среди прочих данных? Решение такое же: вставить перед этим фиктивным escape-символом настоящий. Тогда любой одиночный ESC будет частью escape-последовательности, а двойной будет указывать на то, что служебный байт случайно оказался в потоке данных. В любом случае, байтовая последовательности после ее очищения от вставных символов в точности совпадает с исходной.

Главный недостаток этого метода заключается в том, что он тесно связан с 8-битными символами. Между тем не во всех кодировках один символ соответствует 8 битам. Например, UNICODE может использовать 16-битное кодирование. По мере развития сетевых технологий недостатки использования длины символьного кода в механизме формирования кадров становились все очевиднее. Поэтому потребовалось создание новой техники, допускающей использование символов произвольного размера.

Новый метод позволяет использовать кадры и наборы символов, состоящие из любого количества битов. Вот как это реализуется. Каждый кадр начинается и завершается специальной последовательностью битов, 01111110 (на самом деле это все тот же флаговый байт). Если в битовом потоке передаваемых данных

встретится пять идущих подряд единиц, уровень передачи данных автоматически вставит в выходной поток нулевой бит. Битовое заполнение аналогично символьному, при котором в кадр перед случайно встретившимся среди данных флагом вставляется escape-символ. Когда принимающая сторона встречает пять единиц подряд, за которыми следует ноль, она автоматически удаляет этот ноль. Битовое заполнение, как и символьное, является абсолютно прозрачным для сетевого уровня обоих компьютеров. Если флаговая последовательность битов (01111110) встречается в данных пользователя, она передается в виде 011111010, но в памяти принимающего компьютера сохраняется опять в исходном виде: 01111110.

Благодаря битовому заполнению границы между двумя кадрами могут быть безошибочно распознаны с помощью флаговой последовательности. Таким образом, если приемная сторона потеряет границы кадров, ей нужно всего лишь отыскать в полученном потоке битов флаговый байт, поскольку он встречается только на границах кадров и никогда — в данных пользователя.

Наконец, последний метод формирования кадров приемлем только в сетях, в которых физический носитель обладает некоторой избыточностью. Например, некоторые локальные сети кодируют один бит данных двумя физическими битами. Обычно бит 1 кодируется парой высокого и низкого уровней сигналов (отрицательный перепад), а бит 0 — наоборот, парой низкого и высокого уровней (положительный перепад). В такой схеме каждый передаваемый бит данных содержит в середине переход, благодаря чему упрощается распознавание границ битов. Комбинации уровней сигналов (низкий—низкий и высокий—высокий) не используются для передачи данных, но используются в качестве ограничителей кадров в некоторых протоколах.

Многие протоколы передачи данных для повышения надежности применяют комбинацию счетчика символов с другим методом формирования кадра. Когда прибывает кадр, для обнаружения его конца используется счетчик. Кадр воспринимается как правильный только в том случае, если соответствующий разделитель присутствует в нужной позиции и

совпадает контрольная сумма. В противном случае сканируется входной поток, в котором ищется следующий разделитель.

Решив проблему маркировки начала и конца кадра, мы сталкиваемся с новой проблемой: как гарантировать доставку сетевому уровню принимающей машины всех кадров и при этом расположить их в правильном порядке. Предположим, что отправитель просто посылает кадры, не заботясь о том, дошли ли они до адресата. Этого было бы достаточно для сервиса без подтверждений и без установления соединения, но не для ориентированного на соединение сервиса с подтверждениями.

Обычно для гарантирования надежной доставки поставщику посылается информация о том, что происходит на другом конце линии. Протокол требует от получателя посылать обратно специальные управляющие кадры, содержащие позитивные или негативные сообщения о полученных кадрах. Получив позитивное сообщение, отправитель узнает, что посланный им кадр успешно получен на том конце линии. Негативное сообщение, напротив, означает, что с кадром что-то случилось и его нужно передать снова.

Кроме того, посланный кадр может из-за неисправности оборудования или какой-нибудь помехи (например, шума) пропасть полностью. В этом случае принимающая сторона его просто не получит и, соответственно, никак не прореагирует, а отправитель может вечно ожидать положительного или .отрицательного ответа и так ничего и не получить.

Чтобы избежать зависаний сети в случае полной потери кадров, используются таймеры уровня передачи данных. После посылки кадра включается таймер и отсчитывает интервал времени, достаточный для получения принимающим компьютером этого кадра, его обработки и посылки обратно подтверждения. В нормальной ситуации кадр правильно принимается, а подтверждение посылается назад и вручается отправителю, прежде чем истечет установленный интервал времени, и только после этого таймер отключается.

Однако если либо кадр либо подтверждение теряется по пути, установленный интервал времени истечет, и отправитель получит сообщение о возможной проблеме. Самым простым решением для отправителя будет послать кадр еще раз. Однако

при этом возникает опасность получения одного и того же кадра несколько раз уровнем передачи данных принимающего компьютера и повторной передачи его сетевому уровню. Чтобы этого не случилось, необходимо последовательно пронумеровать отсылаемые кадры, так чтобы получатель мог отличить повторно переданные кадры от оригиналов.

Вопрос управления таймерами и порядковыми номерами, гарантирующими, что каждый кадр доставлен сетевому уровню принимающего компьютера ровно один раз, не больше и не меньше, является очень важной частью задачи, решаемой уровнем передачи данных.

Еще один важный аспект разработки уровня передачи данных (а также более высоких уровней) связан с вопросом о том, что делать с отправителем, который постоянно желает передавать кадры быстрее, чем получатель способен их получать. Такая ситуация может возникнуть, если у передающей стороны оказывается более мощный (или менее загруженный) компьютер, чем у принимающей. Отправитель продолжает посылать кадры на высокой скорости до тех пор, пока получатель не окажется полностью ими завален. Даже при идеально работающей линии связи в определенный момент получатель просто не сможет продолжать обработку все прибывающих кадров и начнет их терять. Очевидно, что для предотвращения подобной ситуации следует что-то предпринять.

В настоящее время применяются два подхода. При первом, называющемся управлением потоком с обратной связью, получатель отсылает отправителю информацию, разрешающую последнему продолжить передачу или, по крайней мере, сообщающую о том, как идут дела у получателя. При втором подходе, управлении потоком с ограничением, в протокол встраивается механизм, ограничивающий скорость, с которой передатчики могут передавать данные. Обратная связь с получателем отсутствует.

Известны различные схемы контроля потока с обратной связью, но большинство из них используют один и тот же принцип. Протокол содержит четко определенные правила, определяющие, когда отправитель может посылать следующий кадр. Эти правила часто запрещают пересылку кадра до тех пор,

пока получатель не даст разрешения, либо явно, либо неявно. Например, при установке соединения получатель может сказать: «Вы можете послать мне сейчас n кадров, но не посылайте следующие кадры, пока я не попрошу вас продолжать».

**Протоколы передачи данных.**

Знакомство с протоколами мы начнем с рассмотрения трех протоколов возрастающей сложности. Симулятор этих и следующих протоколов заинтересованные читатели могут найти на веб-сайте (см. предисловие). Прежде чем приступить к

изучению протоколов, полезно высказать некоторые допущения, лежащие в основе данной модели связи. Для начала мы предполагаем, что на физическом уровне, уровне передачи данных и сетевом уровне находятся независимые процессы,

общающиеся с помощью передачи друг другу сообщений. Во многих случаях процессы физического уровня и уровня передачи данных будут работать на процессоре внутри специальной сетевой микросхемы ввода-вывода, а процесс сетевого уровня — на центральном процессоре. Однако другие варианты реализации также возможны (например, три процесса внутри одной микросхемы ввода-вывода; физический уровень и уровень передачи данных, вызываемые как процедуры процессом сетевого уровня, и т. д.). В любом случае, представление трех уровней в виде отдельных процессов будет служить поддержанию концептуальной чистоты обсуждения, а также подчеркнет независимость уровней Другим ключевым допущением будет то, что машина А хочет послать на машину В длинный поток данных, используя надежный ориентированный на соединение сервис. Позднее мы рассмотрим случай, при котором одновременно машина В также хочет послать данные на машину А. Предполагается, что у машины А имеется бесконечный источник данных, готовых к отправке, и что ей никогда не требуется ждать готовности данных. Когда уровень передачи данных машины А запрашивает данные, сетевой уровень всегда готов их ему предоставить. (Это ограничение также будет потом отброшено.)

Также предполагается, что компьютеры не выходят из строя. При передаче могут возникать ошибки, но не проблемы, связанные с поломкой оборудования или случайным сбросом.

При рассмотрении уровня передачи данных пакет, передаваемый ему по интерфейсу сетевым уровнем, рассматривается как чистые данные, каждый бит которых должен быть доставлен сетевому уровню принимающей машины. Тот факт, что сетевой уровень принимающей машины может интерпретировать часть этого пакета как заголовок, не касается уровня передачи данных.

Получив пакет, уровень передачи данных формирует из пакета кадры, добавляя заголовок и окончание пакета передачи данных  Таким образом, кадр состоит из внедренного пакета, некоторой служебной информации (в заголовке) и контрольной суммы (в концевике). Затем кадр передается уровню передачи данных принимающей машины. Мы будем предполагать наличие соответствующих библиотечных процедур, например, to_physical_layer для отправки кадра и from_physical_layer для получения кадра. Передающая аппаратура вычисляет и добавляет контрольную сумму (создавая конечную часть кадра), так что уровень передачи данных может не беспокоиться об этом.

Вначале получатель ничего не должен делать. Он просто сидит без дела, ожидая, что что-то произойдет. В приводимых в данной главе примерах протоколов ожидание событий уровнем передачи данных обозначается вызовом процедуры wait_for_event(&event). Эта процедура возвращает управление, только когда что-то происходит (например, прибывает кадр). При этом переменная event сообщает, что именно случилось. Наборы возможных событий отличаются в разных протоколах и поэтому будут описываться для каждого протокола отдельно. Следует заметить, что в действительности уровень передачи данных не находится в холостом цикле ожидания событий, как мы предположили, а получает прерывание, когда это событие происходит. При этом он приостанавливает свои текущие процессы и обрабатывает пришедший кадр. Тем не менее, для простоты мы проигнорируем эти детали и предположим, что уровень передачи данных все свое время посвящает работе с одним каналом.

Когда приемная машина получает кадр, аппаратура вычисляет его контрольную сумму. Если она неверна (то есть при передаче возникли ошибки), то уровень передачи данных получает соответствующую информацию (event = cksum_err).

Если кадр прибывает в целости, уровню передачи данных об этом также сообщается (euew£=frame_arrival), после чего он может получить этот кадр у физического уровня с помощью процедуры from_physical_layer. Получив неповрежденный кадр, уровень передачи данных проверяет управляющую информацию, находящуюся в заголовке кадра, и если все в порядке, часть этого кадра передается сетевому уровню. Заголовок кадра не передается сетевому уровню ни при каких обстоятельствах.

Для запрета передачи сетевому уровню любой части заголовка кадра есть веская причина: поддержание полного разделения сетевого уровня и уровня передачи данных. До тех пор, пока сетевой уровень ничего не знает о формате кадра и протоколе уровня передачи данных, изменения формата и протокола не потребуют изменений программного обеспечения сетевого уровня. Поддержание строгого интерфейса между сетевым уровнем и уровнем передачи данных значительно упрощает разработку программ, так как протоколы различных уровней могут развиваться независимо.

В листинге показаны некоторые объявления (на языке С), общие для многих протоколов, обсуждаемых далее. Определены пять структур данных: boolean, seq_nr, packet, framejdnd и frame. Тип boolean представляет собой перечисляемый тип, переменные которого могут принимать значения true или false. Тип seq_nr является целым без знака, используемым для нумерации кадров. Эти последовательные номера могут принимать значения от 0 до числа MAX_SEQ включительно, которое определяется в каждом протоколе, использующем его. Тип packet является единицей информации, используемой при обмене информацией между сетевым уровнем и уровнем передачи данных одной машины или двумя равноранговыми сетевыми уровнями. В рассматриваемой модели пакет всегда состоит из MAX_PKT байт, хотя на практике он обычно имеет переменную длину.

Структура frame состоит из четырех полей: kind, seq, ack и info, — первые три из которых содержат управляющую

информацию, а последнее может содержать данные, которые необходимо передать. Эти три управляющих поля вместе называются заголовком кадра. Поле kind сообщает о наличии данных в кадре, так как некоторые протоколы отличают кадры, содержащие только управляющую информацию, от кадров, содержащих также и данные. Поля seq и ack используются соответственно для хранения последовательного номера кадра и подтверждения. Подробнее их использование будет описано далее. Поле данных кадра, info, содержит один пакет. В управляющем кадре поле info не используется. В реальной жизни используется поле info переменной длины, полностью отсутствующее в управляющих кадрах. Важно понимать взаимоотношения между пакетом и кадром. Сетевой уровень создает пакет, принимая сообщение от транспортного уровня и добавляя к нему заголовок сетевого уровня. Этот пакет передается уровню передачи данных, который включает его в поле info исходящего кадра. Когда кадр прибывает на пункт назначения, уровень передачи данных извлекает пакет из кадра и передает его сетевому уровню. Таким образом, сетевой уровень может действовать так, как будто машины обмениваются пакетами напрямую.

В листинге также перечислен ряд процедур. Это библиотечные процедуры, детали которых зависят от конкретной реализации, и их внутреннее устройство мы рассматривать не будем. Как уже упоминалось ранее, процедура wait_for_event представляет собой холостой цикл ожидания какого-нибудь события. Процедуры to_network_layer и from_network_layer используются уровнем передачи данных для передачи пакетов сетевому уровню и для получения пакетов от сетевого уровня соответственно. Обратите внимание: процедуры from_physical_layer и to_physical_layer используются для обмена кадрами между уровнем передачи данных и физическим уровнем, тогда как процедуры to_network_layer и from_network_layer применяются для передачи пакетов между уровнем передачи данных и сетевым уровнем. Другими словами, процедуры to_network_layer и from_network_layer относятся к интерфейсу между уровнями 2 и 3, тогда как процедуры from_physical_layer и to_physical_1ayer относятся к интерфейсу между уровнями 1 и 2.

В большинстве протоколов предполагается использование ненадежного канала, который может случайно потерять целый кадр. Для обработки подобных ситуаций передающий уровень передачи данных, посылая кадр, запускает таймер. Если за установленный интервал времени ответ не получен, таймер воспринимает это как тайм-аут, и уровень передачи данных получает сигнал прерывания.

В наших примерах протоколов это реализовано в виде значения event=timeout, возвращаемого процедурой wait_for_event. Для запуска и остановки таймера используются процедуры start_timer и stop_timer соответственно. Событие timeout может произойти, только если запущен таймер. Процедуру start_timer разрешается запускать во время работающего таймера. Такой вызов просто переинициализирует часы, чтобы можно было начать отсчет заново (до нового тайм-аута, если таковой будет иметь место).

Процедуры start_ack_timer и stop_ack_timer используются для управления вспомогательными таймерами при формировании подтверждений в особых обстоятельствах.

Процедуры enable_network_layer и disable_network_layer используются в более сложных протоколах, где уже не предполагается, что у сетевого уровня всегда есть пакеты для отправки. Когда уровень передачи данных разрешает работу сетевого уровня, последний получает также разрешение прерывать работу первого, когда ему нужно послать пакет. Такое событие мы будем обозначать как event=network_layer_ready. Когда сетевой уровень отключен, он не может инициировать такие события. Тщательно следя за включением и выключением сетевого уровня, уровень передачи данных не допускает ситуации, в которой сетевой уровень заваливает его пакетами, для которых не осталось места в буфере.

Последовательные номера кадров всегда находятся в пределах от 0 до MAX_SEQ (включительно). Число MAX_SEQ различно в разных протоколах. Для увеличения последовательного номера кадров на 1 циклически (то есть с обнулением при достижении числа MAX_SEQ) используется макрос inc.

Определение простых операций в виде макросов не снижает удобочитаемости программы, увеличивая при этом ее

быстродействие. К тому же, поскольку MAX_SEQ в разных протоколах будет иметь разные значения, то, определив инкремент в виде макроса, можно один и тот же код без проблем использовать в нескольких протоколах. Такая возможность крайне полезна для симулятора

Листинг 4.1. Общие объявления для последующих протоколов. Объявления располагаются в файле protocol.h

```
#define MAX_PKT 1024 /* определяет размер пакета в байтах */
typedef enum{false, true} boolean; /* тип boolean */
typedef unsigned int seq_nr; /* порядковые номера кадров или
подтверждений */
typedef struct {unsigned char data[MAX_PKT];} packet; /*
определение пакета */
typedef enum{data, ack, nak} frante_kind; /* определение
типа пакета */
typedef struct {
frame_kind kind; /* тип кадра */
seq_nr seq; /* порядковый номер */
seq_nr ack; /* номер подтверждения */
packet info; /* пакет сетевого уровня */
} frame;
/* ожидать события и вернуть тип события в переменной event
*/
void wait_for_event(event_type *event);
/* получить пакет у сетевого уровня для передачи по каналу
*/
void from_network_layer(packet *p);
/* передать информацию из полученного пакета сетевому уровню
*/
void to_network_layer(packet *p);
/* получить пришедший пакет у физического уровня и
скопировать его в r */
void from_physical_layer(frame *r);
/* передать кадр физическому уровню для передачи */
void to_physical_layer(frame *s);
/* запустить таймер и разрешить событие timeout */
void start_timer(seq_nr k);
/* остановить таймер и запретить событие timeout */
void stop_timer(seq_nr k);
/* запустить вспомогательный таймер и разрешить событие
ack_timeout */
```

```
void start_ack_timer(void);
/* остановить вспомогательный таймер и запретить событие ack
timeout */
void stop_ack_timer(void);
/* разрешить сетевому уровню инициировать событие
network_layer_ready */
void enable_network_layer(void);
/* запретить сетевому уровню инициировать событие network
layer ready */
void disable_network_layer(void);
/* макрос inc развертывается прямо в строке: Циклически
увеличить переменную k */
#define inc(k) if (k < MAX SEQ) k = k + 1; else k = 0
```

### Тривиальный симплексный протокол

В качестве первого примера мы рассмотрим самый простой протокол. Данные передаются только в одном направлении. Сетевой уровень на передающей и приемной сторонах находится в состоянии постоянной готовности. Временем обработки можно пренебречь. Размер буфера неограничен. И что лучше всего, канал связи между уровнями передачи данных никогда не теряет и не искажает кадры.

Протокол состоит из двух процедур, sender (отправитель) и receiver (получатель). Процедура sender работает на уровне передачи данных посылающей машины, а процедура receiver - на уровне передачи данных принимающей машины.

Ни последовательные номера, ни подтверждения не используются, поэтому MAX_SEQ не требуется. Единственным возможным событием является frame_arrival (то есть прибытие неповрежденного кадра).

Процедура sender представляет собой бесконечный цикл, начинающийся с оператора while, посылающий данные на линию с максимально возможной скоростью. Тело цикла состоит из трех действий: получения пакета (всегда обязательное) с сетевого уровня, формирования исходящего пакета с помощью переменной s и отсылки пакета адресату. Из служебных полей кадра данный протокол использует только поле info, поскольку другие поля

относятся к обработке ошибок и управлению потоком, которые в данном протоколе не применяются.

Процедура принимающей стороны ничуть не сложнее. Вначале она ожидает, пока что-нибудь произойдет, причем единственно возможным событием в данном протоколе может быть получение неповрежденного пакета. Когда пакет появляется, процедура wait_for_event возвращает управление, при этом переменной event присваивается значение frame_arrival (которое все равно игнорируется).

Обращение к процедуре from_physica1_1ayer удаляет вновь прибывший кадр из аппаратного буфера и помещает его в переменную r. Наконец, порция данных передается сетевому уровню, а уровень передачи данных отправляется ждать следующий кадр.

```
/* Протокол 1 обеспечивает только одностороннюю передачу
данных - от отправителя к получателю. Предполагается, что в
канале связи нет ошибок и что получатель способен мгновенно
обрабатывать получаемые данные. Соответственно, отправитель
в цикле передает данные на линию с максимально доступной для
него скоростью. */
typedef enum{frame_arrival} event_type;
#include "protocol.h"
void sender1(void)
{
        frame s; /* буфер для исходящего кадра */
        packt buffer; /* буфер для исходящего пакета */
        while (true) {
                from_network_layer(&buffer); /* получить у
сетевого уровня пакет для передачи */
                s.info = buffer; /* скопировать его в кадр s
для передачи */
                to_physical_1ayer(&s); /* послать кадр s */
        }
}
void receiver1(void)
{
        frame r;
        event_type event; /* заполняется процедурой ожидания
событий, но не используется здесь */
        while (true) {
```

```
            wait_for_event(&event); /* единственное
возможное событие - прибытие кадра,
            событие frame_arrival */
            from_physical_layer(&r): /* получить прибывший
кадр */
            to_network_layer(&r.info); /* передать данные
сетевому уровню */
        }
}
```

**Симплексный протокол с ожиданием**

Основная проблема, которую необходимо решить, — как предотвратить ситуацию, когда отправитель посылает данные быстрее, чем получатель может их обработать. То есть если получателю требуется время t, чтобы выполнить процедуры from_physica1_layer и to_network_layer, то отправитель должен передавать со средней скоростью меньшей, чем один кадр за интервал времени t. Более того, если мы предполагаем, что в принимающей аппаратуре не производится автоматической буферизации, то отправитель не должен посылать новый кадр до тех пор, пока старый кадр не будет считан процедурой from_physical_layer. В противном случае новый кадр окажется записанным поверх старого.

При некоторых обстоятельствах (например, при синхронной передаче, когда уровень передачи данных принимающей машины обрабатывает всего одну входную линию) может быть достаточно всего лишь вставить задержку в передающую программу протокола 1, снизив скорость его работы настолько, чтобы уберечь принимающую сторону от забивания данными. Однако в реальных условиях обычно каждый уровень передачи данных должен одновременно обрабатывать несколько линий, и время, необходимое на обработку одного кадра, может меняться в довольно значительных пределах. Если разработчик сети может рассчитать наихудшую ситуацию для приемника, он может запрограммировать настолько медленную работу отправителя, что даже при самой медленной обработке поступающих кадров получатель будет успевать их обрабатывать. Недостаток такого подхода в его консерватизме. В данном случае

использование пропускной способности будет намного ниже оптимального уровня. Исключением может быть только один случай — когда время реакции принимающего уровня передачи данных изменяется очень незначительно.

Лучшим решением данной проблемы является обратная связь со стороны получателя. Передав пакет сетевому уровню, получатель посылает небольшой управляющий пакет отправителю, разрешая тем самым посылать следующий кадр. Отправитель, отослав кадр, должен ждать разрешения на отправку следующего кадра.

```
/* Протокол 2 (с ожиданием) также обеспечивает только
одностороннюю передачу данных от отправителя к получателю.
Снова предполагается, что в канале связи нет ошибок.
Однако на этот раз емкость буфера получателя ограничена, и,
кроме того, ограничена скорость обработки данных получателя.
Поэтому протокол должен не допускать отправления данных
быстрее, чем получатель способен их обработать. */
typedef enum {frame_arrival} event_type;
#include "protocol.h"
void sender2(void)
{
frame s; /* буфер для исходящего кадра */
packet buffer; /* буфер для исходящего пакета */
event_type event; /* единственное возможное событие -
прибытие кадра (событие frame arrival)*/
        while (true) {
                from_network_layer(&buffer); /* получить у
сетевого уровня пакет для передачи */
                s.info = buffer; /* скопировать его в кадр s
для передачи */
                to_physical_layer(&s);
                wait_for_evint(&event); /* не продолжать, пока
на это не будет получено разрешения */
        }
}
void receiver2(void)
{
        frame r, s; /* буферы для кадров */
        event_type event; /* frame arrival является
единственным возможным событием */
```

```
        while (true) {
                wait_for_event(&event); /* единственное
возможное событие - прибытие кадра (событие frame_arrival)*/
                from_physical_layer(&r); /* получить прибывший
кадр */
                to_network_layer(&r.info); /* передать данные
сетевому уровню */
                to_physical_layer(&s); /* передать пустой кадр,
чтобы разбудить отправителя */
        }
}
```

**Симплексный протокол с контролем ошибок**

Теперь рассмотрим реальную ситуацию: канал связи, в котором могут быть ошибки. Кадры могут либо портиться, либо теряться. Однако мы будем предполагать, что если кадр будет поврежден при передаче, то приемная аппаратура определит это при подсчете контрольной суммы. Если кадр будет поврежден таким образом, что контрольная сумма совпадет, что очень маловероятно, то этот протокол (и любой другой протокол) передаст неверный пакет сетевому уровню.

На первый взгляд может показаться, что нужно лишь добавить таймер к протоколу 2. Получатель будет возвращать подтверждение только в случае получения правильных данных. Неверные пакеты будут просто игнорироваться. Через некоторое время у отправителя истечет интервал времени, и он отправит кадр еще раз. Этот процесс будет повторяться до тех пор, пока кадр, наконец, не прибудет в целости.

В приведенной выше схеме имеется один критический недостаток.

Рассмотрим следующий сценарий.

1. Сетевой уровень машины A передает пакет 1 своему уровню передачи данных. Пакет доставляется в целости на машину B и передается ее сетевому уровню. Машина B посылает кадр подтверждения назад на машину A.

2. Кадр подтверждения полностью теряется в канале связи. Он просто не попадает на машину A. Все было бы намного проще, если бы терялись только информационные — но не управляющие

— кадры, однако канал связи, к сожалению, не делает между ними большой разницы.

3. У уровня передачи данных машины А внезапно истекает отведенный интервал времени. Не получив подтверждения, он предполагает, что посланный им кадр с данными был поврежден или потерян, и посылает этот кадр еще раз.

4. Дубликат кадра прибывает на уровень передачи данных машины В и передается на сетевой уровень. Если машина А посылала на машину В файл, то частьэтого файла продублировалась, таким образом, копия файла на машине В будет неверной. Другими словами, протокол допустил ошибку.

5. Понятно, что необходим некий механизм, с помощью которого получатель смог бы отличать новый кадр от переданного повторно. Наиболее очевидным способом решения данной проблемы является помещение отправителем порядкового номера кадра в заголовке кадра. Тогда по номеру кадра получатель сможет понять, новый это кадр или дубликат.

Поскольку отводить в кадре много места под заголовок нежелательно, возникает вопрос: каково минимальное количество бит, достаточное для порядкового номера кадра? Единственная неопределенность в данном протоколе может возникнуть между кадром т и следующим за ним кадром n + 1. Если кадр n потерян или поврежден, получатель не подтвердит его и отправитель пошлет его еще раз. Когда он будет успешно принят, получатель пошлет отправителю подтверждение. Именно здесь находится источник потенциальной проблемы. В зависимости от того, будет получено подтверждение или нет, отправитель может послать кадр n или кадр n + 1.

Отправитель может начать посылать кадр n + 2 только когда получит подтверждение получения кадра n + 1. Но это означает, что кадр уже был отправлен и подтверждение его получения было отправлено и получено. Следовательно, неопределенность может возникнуть только между двумя соседними кадрами.

Таким образом, должно быть достаточно всего одного бита информации (со значением 0 или 1). В каждый момент времени получатель будет ожидать прибытия кадра с определенным порядковым номером. Кадр с неверным номером будет

отбрасываться как дубликат. Кадр с верным номером принимается, передается сетевому уровню, после чего номер следующего ожидаемого кадра увеличивается по модулю 2 (то есть 0 становится 1, а 1 — 0).

```
/* Протокол 3  обеспечивает симплексную передачу данных по
ненадежному каналу. */
#define MAX_SEQ 1
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void sender3(void)
{
       seq_nr next_frame_to_send; /* порядковый номер
следующего исходящего кадра */
       frame s; /* временная переменная */
       packet buffer; /* буфер для исходящего пакета */
       event_type event;
       next_frame_to_send = 0; /* инициализация исходящих
последовательных номеров */
       from_network_layer(&buffer); /* получить первый пакет
у сетевого уровня */
       while (true) {
              s.info = buffer; /* сформировать кадр для
передачи */
              s.seq = next_frame_to_send; /* вставить
порядковый номер в кадр */
              to_physical_layer(&s); /* послать кадр по
каналу */
              start_timer(s.seq); /* запустить таймер
ожидания подтверждения */
              wait_for_event(&event): /* ждать события
frame_arrival, cksum_err или timeout */
              if (event == frame_arrival) {
                     from_physical_layer(&s); /* получить
подтверждение */
                     if (s.ack = next_frame_to_send) {
                            from_network_layer(&buffer); /*
получить следующий пакет у сетевого уровня */
                            inc(next_frame_to_send); /*
инвертировать значение переменной next_frame_to_send */
                     }
              }
```

```
        }
}
void receiver3(void)
{
        seq_nr frame_expected;
        frame r, s:
        event_type event;
        frame_expected = 0;
        while (true) {
                wait_for_event(&event); /* ожидание возможных
событий: frame_arrival. cksum_err */
                if (event == frame_arrival) {
                        /* прибыл неповрежденный кадр */
                        from_physical_layer(&r); /* получить
прибывший кадр */
                        if (r.seq = frame_expected) { /* именно
этот кадр и ожидался */
                                to_network_layer(&r.info); /*
передать данные сетевому уровню */
                                inc(frame_expected); /* в
следующий раз ожидать кадр с другим порядковым номером */
                        }
                        s.ack = 1 - frame_expected; /* номер
кадра, для которого посылается подтверждение */
                        to_physical_layer(&s);
                }
        }
}
```

Данный протокол отличается от своих предшественников тем, что и отправитель, и получатель запоминают номера кадров. Отправитель запоминает номер следующего кадра в переменной next_frame_to_send, а получатель запоминает порядковый номер следующего ожидаемого кадра в переменной frame_expected. Перед бесконечным циклом в каждой процедуре размещена короткая фаза инициализации.

Передав кадр, отправитель запускает таймер. Если он уже был запущен, он настраивается на отсчет нового полного интервала времени. Период выбирается достаточно большим, чтобы даже в худшей ситуации кадр успел дойти до получателя, получатель успел его обработать и подтверждение успело вернуться к отправителю. Только по истечении отведенного

времени можно утверждать, чтопотерялся кадр или его подтверждение, а значит, необходимо послать дубликат.

Если время, после которого наступает тайм-аут, сделать слишком коротким, то передающая машина будет повторно посылать слишком много кадров, в которых нет необходимости. Хотя лишние кадры в данном случае не повлияют на правильность приема данных, они повлияют на производительность системы.

После передачи кадра отправитель запускает таймер и ждет какого-либо события. Возможны три ситуации: либо придет неповрежденный кадр подтверждения, либо будет получен поврежденный кадр подтверждения, либо просто истечет интервал времени. В первом случае отправитель возьмет у сетевого уровня следующий пакет и положит его в буфер поверх старого пакета. Кроме того, он увеличит порядковый номер кадра. Если же прибудет поврежденный кадр подтверждения или подтверждение не придет вовсе, то ни буфер, ни номер не будут изменены и будет послан дубликат кадра.

Когда неповрежденный кадр прибывает к получателю, проверяется его номер. Если это не дубликат, то кадр принимается и передается сетевому уровню, после чего формируется подтверждение. Дубликаты и поврежденные кадры на сетевой уровень не передаются.

**Дуплексный протокол передачи с минимальным размером буффера передачи**

Рассмотрим протокол с максимальным размером буффера, равным 1. Такой протокол использует метод ожидания, поскольку, отослав кадр, отправитель, прежде чем послать следующий кадр, должен дождаться подтверждения [8].

Данный протокол приведен ниже в листинге. Как и другие протоколы, он начинается с определения некоторых переменных. Переменная next_frame_to_send содержит номер кадра, который отправитель пытается послать. Аналогично переменная frame_expected хранит номер кадра, ожидаемого получателем. В обоих случаях возможными значениями могут быть только 0 и 1.

В нормальной ситуации только один уровень передачи данных может начинать передачу. Другими словами, только одна

из программ должна содержать обращения к процедурам to_physical_layer и start_timer вне основного цикла. Если оба уровня передачи данных начинают одновременно передавать, возникает непростая ситуация. Начинающая машина получает первый пакет от своего сетевого уровня, создает из него кадр и посылает его. Когда этот (или другой) кадр прибывает, получающий уровень передачи данных проверяет, не является ли этот кадр дубликатом, аналогично протоколу 3. Если это тот кадр, который ожидался, он передается сетевому уровню и окно получателя сдвигается вверх.

Поле подтверждения содержит номер последнего полученного без ошибок кадра. Если этот номер совпадает с номером кадра, который пытается передать отправитель, последний понимает, что этот кадр успешно принят получателем, и что он может пересылать следующий кадр. В противном случае он должен продолжать попытки переслать тот же кадр.

```
 /* Протокол 4 является дуплексным и более надежным, чем
протокол 3.
*/
#define MAX_SEQ 1
typedef enum {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"
void protocol4 (void)
{
seq_nr next_frame_to_send; /* только 0 или 1 */
seq_nr frame_expected;     /* только 0 или 1 */
frame r, s; /* временные переменные */
packet buffer; /* текущий посланный пакет */
event_type event;
next_frame_to_send = 0; /* номер следующего кадра в
исходящем потоке */
frame_expected = 0;     /* номер ожидаемого кадра */
from_network_layer(&buffer); /* получить первый пакет у
сетевого уровня */
s.info = buffer;            /* подготовить первый кадр для
передачи */
s.seq = next_frame_to_send;  /* вставить порядковый номер в
кадр */
s.ack = 1 - frame_expected;  /* подтверждение */
```

```
to_physical_layer(&s);          /* послать кадр */
start_timer(s.seq); /* запустить таймер ожидания
подтверждения */
while (true) {
        wait_for_event(&event); /* ждать возможного события:
frame_arrival, cksum_err или timeout */
        if (event = frame_arrival) {
                from_physical_layer(&r); /* кадр прибыл в
целости получить кадр */
                if (r.seq == frame_expected) {
                        /* обработать входящий поток кадров */
                        to_network_layer(&r.info); /* передать
пакет сетевому уровню */
                        inc(frame_expected); /* инвертировать
порядковый номер кадра, ожидаемого в следующий раз */
                }
                if (r.ack == next_frame_to_send) { /*
обработать исходящий поток кадров */
                        from_network_layer(&buffer); /* получить
следующий пакет у сетевого уровня */
                        inc(next_frame_to_send): /*
инвертировать порядковый номер посылаемого кадра */
                }
        }
        s.info = buffer; /* подготовить кадр для передачи */
        s.seq = next_frame_to_send; /* вставить порядковый
номер в кадр */
        s.ack = 1 - frame_expected; /* порядковый номер
последнего полученного кадра */
        to_physical_layer(&s); /* передать кадр */
        start_timer(s.seq); /* запустить таймер ожидания
подтверждения */
}
}
```

**Конвейерный протокол передачи данных**

В следующем листинге показан конвейерный протокол, в котором принимающий уровень передачи данных принимает кадры по порядку. Все кадры, следующие за ошибочным, игнорируются.

В данном протоколе не используется предположение, что у сетевого уровня всегда есть неограниченное количество пакетов для отсылки. Когда у сетевого уровня появляется пакет, который он хочет отправить, уровень инициирует событие network_layer_ready. Однако чтобы ограничить количество неподтвержденных пакетов числом MAX_SEQ, уровень передачи данных должен иметь возможность отключать на время сетевой уровень. Для этой цели служит пара библиотечных процедур — enable_network_layer и disable_network_layer. Обратите внимание на то, что в любой момент времени может быть MAX_SEQ, а не MAX_SEQ + 1 неподтвержденных пакетов, хотя различаются MAX_SEQ + 1 порядковых номеров: от 0 до MAX_SEQ. Чтобы понять, почему необходимо такое ограничение, рассмотрим сценарий с MAX__SEQ = 7.

1. Отправитель посылает кадры с 0 по 7.

2. Подтверждение для кадра 7 наконец прибывает к отправителю.

3. Отправитель посылает следующие восемь кадров, снова с номерами с 0 по 7.

4. Еще одно подтверждение для кадра 7 прибывает к отправителю.

Но вот вопрос: все восемь кадров, входящих во второй набор, благополучно дошли до адресата или все они потерялись (включая игнорированные кадры после ошибочного)? В обоих случаях получатель отправит кадр 7 в качестве подтверждения. У отправителя нет способа отличить один случай от другого. По этой причине максимальное количество неподтвержденных кадров должно быть ограничено числом MAX_SEQ.

Хотя в протоколе 5 кадры, поступившие после ошибки, не буферизируются получателем, тем не менее, отправитель должен хранить отправленные кадры в своем буфере, пока не получит на них подтверждение. Если поступает подтверждение на кадр n, кадры n - 1, n - 2 (то есть все предыдущие кадры) автоматически считаются подтвержденными. Эта особенность наиболее важна в случае потери или повреждения какого-либо кадра с подтверждением. Получив подтверждение, уровень передачи данных проверяет, не освободился ли у него какой-нибудь буфер. Если буфер освобождается, то заблокированному ранее сетевому

уровню можно снова разрешить инициировать события network_layer_ready.

Для этого протокола предполагается, что всегда есть обратный трафик, по которому можно отправлять подтверждение. Если же это условие не выполняется, то никакие подтверждения отосланы не будут. Протокол 4 не нуждается в подобном допущении, поскольку за каждым принятым кадром следует обратный, даже если только что уже был отправлен какой-то кадр в сторону отправителя.

В следующем протоколе проблема отсутствия обратного трафика будет решена гораздо элегантней.

Поскольку протокол 5 хранит несколько неподтвержденных кадров, ему требуется несколько таймеров, по одному на кадр. Для каждого кадра время считается независимо от других. Все таймеры могут симулироваться программно, использую единственные аппаратные часы, вызывающие периодические прерывания.

Данные таймеров могут храниться в программе в виде связанного списка, каждый узел которого будет хранить количество временных интервалов системных часов, оставшихся до истечения срока ожидания, номер кадра и указатель на следующий узел списка.

```
/* Протокол 5 (конвейерный) допускает наличие нескольких
неподтвержденных кадров. Отправитель может передать до
MAX_SEQ кадров, не ожидая подтверждения. Кроме того, в
отличие от предыдущих протоколов, не предполагается, что у
сетевого уровня всегда есть новые пакеты. При появлении
нового пакета сетевой уровень инициирует событие
network_layer_ready */
#define MAX_SEQ 7 /* должно быть 2^n-1 */
typedef enum {frame_arrival, cksum_err, timeout,
network_layer_ready} event_type;
#include "protocol.h"
static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
/* возвращает true, если a <=b < c циклично: иначе false */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b
< c) && (c < a))) {
       return(true);
  } else {
```

```
        return(false);
    }
}
static void send_data(seq_nr framejir, seq_nr
frame_expected, packet buffer[])
{
/* подготовить и послать информационный кадр */
frame s; /* временная переменная */
s.info = buffer[frame_nr]; /* вставить пакет в кадр */
s.seq = frame_nr; /* вставить порядковый номер в кадр */
s.ack = (frame_expected + MAX_SEQ) %(MAX_SEQ + 1) ; /*
подтверждение, посылаемое в кадре данных */
to_physical_layer(&s); /* послать кадр по каналу */
start_timer(frame_nr); /* запустить таймер ожидания
подтверждения */
}
void protocol5(void)
{
seq_nr next_frame_to_send; /* MAXj SEQ > 1; используется для
исходящего потока */
seq_nr ack_expected; /* самый старый неподтвержденный кадр
*/
seq_nr frame_expected; /* следующий кадр, ожидаемый во
входящем потоке */
frame r; /* временная переменная */
packet buffer[MAX_SEQ+1]; /* буферы для исходящего потока */
seq_nr nbuffered; /* количество использующихся в данный
момент выходных буферов */
seq_nr i; /* индекс массива буферов */
event_type event;
enable_network_layer(); /* разрешить события
network_layer_ready */
ack_expected = 0; /* номер следующего ожидаемого входящего
подтверждения */
next_frame_to_send = 0; /* номер следующего посылаемого
кадра */
frame_expected = 0; /* номер следующего ожидаемого входящего
кадра */
nbuffered = 0; /* вначале буфер пуст */
while (true) {
        wait_for_event(&event); /* четыре возможных события:
см. event_type выше */
        switch(event) {
```

```
            case network_layer_ready: /* у сетевого уровня
есть пакет для передачи */
                    /* получить, сохранить и передать новый
кадр */

       from_network_layer(&buffer[next_frame_to_send]); /*
получить новый пакет у сетевого уровня */
                    nbuffered = nbuffered + 1 ; /* увеличить
окно отправителя */
                    send_data(next_frame_to_send.
frame_expected. buffer); /* передать кадр */
                    inc(next_frame_to_send); /* увеличить
верхний край окна отправителя */
                    break;
            case frame arrival: /* прибыл кадр с данными
или с подтверждением */
                    from_physical_layer(&r); /* получить
пришедший кадр у физического уровня*/
                    if (r.seq = frame_expected) {
                            /* кадры принимаются только по
порядку номеров */
                            to_network_layer(&r.info); /*
передать пакет сетевому уровню */
                            inc(frame_expected); /*
передвинуть нижний край окна получателя */
                    }
                    /* подтверждение для кадра n
подразумевает также кадры n - 1. n – 2... */
                    while (between(ack_expected, r.ack ,
next_frame_to_send)) {
                            /* отправить подтверждение вместе
с информационным кадром */
                            nbuffered = nbuffered - 1; /* в
буфере на один кадр меньше */
                            stop_timer(ack_expected); /* кадр
прибыл в целости; остановить таймер */
                            inc(ack_expected); /* уменьшить
окно отправителя */
                    }
                    break;
            case cksum_err: break; /* плохие кадры просто
игнорируются */
```

```
            case timeout: /* время истекло; передать
повторно все неподтвержденные кадры */
                    next_frame_to_send = ack_expected; /*
номер первого посылаемого повторно кадра */
                    for (i = 1; i <= nbuffered; i++) {
                            send_data(next_frame_to_send,
frame_expected. buffer); /* переслать повторно 1 кадр */
                            inc(next_frame_to_send); /*
приготовиться к пересылке следующего кадра */
                    }
        }
        if (nbuffered < MAX_SEQ) {
                enable_network_layer( ) ;
        } else {
                disable_network_layer();
        }
}
}
```

**Дуплексный протокол передачи с выборочным повтором**

Протокол 5 хорошо работает, если ошибки встречаются нечасто, однако при плохой линии он тратит впустую много времени, передавая кадры по два раза. В качестве альтернативы можно использовать протокол, в котором получатель буферизирует кадры, принятые после потерянного или испорченного кадра. Такой протокол не отвергает кадры только лишь потому, что предыдущий кадр был поврежден или потерян.

В этом протоколе и отправитель, и получатель работают с окнами допустимых номеров кадров. Размер окна отправителя начинается с нуля и растет до некоторого определенного уровня, MAX_SEQ. Размер окна получателя, напротив, всегда фиксированного размера, равного MAX_SEQ. Получатель должен иметь буфер для каждого кадра, номер которого находится в пределах окна. С каждым буфером связан бит, показывающий, занят буфер или свободен. Когда прибывает кадр, функция between проверяет, попал ли его порядковый номер в окно. Если да, то кадр принимается и хранится в буфере. Это действие производится независимо от того, является ли данный кадр

следующим кадром, ожидаемым сетевым уровнем. Он должен храниться на уровне передачи данных до тех пор, пока все предыдущие кадры не будут переданы сетевому уровню в правильном порядке.

Способность протокола принимать кадры в произвольном порядке вызывает некоторые проблемы, отсутствовавшие в предыдущих протоколах, в которых все пакеты принимались строго по порядку номеров. Проще всего проиллюстрировать это на примере. Предположим, что порядковый номер кадра состоит из 3 бит, так что отправитель может посылать до семи кадров, прежде чем перейти в режим ожидания подтверждения.

Отправитель передает кадры с 0 по 6. Окно получателя позволяет ему принимать любые кадры с номерами от 0 по 6 включительно. Все семь кадров прибывают успешно, поэтому получатель подтверждает их прием и передвигает окно для приема кадров с номерами 7, 0, 1, 2, 3, 4 и 5. Все семь буферов помечаются как свободные. Именно в этот момент происходит какое-нибудь бедствие — например, молния ударяет в телефонный столб и стирает все подтверждения. Отправитель, не Дождавшись подтверждений, посылает повторно кадр 0. К сожалению, кадр 0 попадает в новое окно и поэтому принимается получателем. Получатель снова отправляет подтверждение для кадра 6, поскольку были приняты все кадры с 0 по 6. Отправитель с радостью узнает, что все переданные им кадры успешно достигли адресата, поэтому он тут же передает кадры 7, 0, 1, 2, 3, 4 и 5. Кадр 7 принимается получателем, и содержащийся в нем пакет передается сетевому уровню.

Сразу после этого принимающий уровень передачи данных проверяет наличие кадра 0, обнаруживает его и передает внедренный пакет сетевому уровню. Таким образом, сетевой уровень получает неверный пакет; это означает, что протокол со своей задачей не справился.

Причина неудачи в том, что при сдвиге приемного окна новый интервал допустимых номеров кадров перекрыл старый интервал. Соответственно, присылаемый набор кадров может содержать как новые кадры (если все подтверждения были получены), так и повторно высланные старые кадры (если

подтверждения были потеряны). У принимающей стороны нет возможности отличить одну ситуацию от другой.

Решением данной проблемы является предоставление гарантии того, что в сдвинутом положении окно не перекроет исходное окно. Для этого размер окна не должен превышать половины от количества порядковых номеров. Например, если для порядковых номеров используются 4 бита, они должны изменяться в пределах от 0 до 15. В таком случае в любой момент времени только восемь кадров могут быть неподтвержденными. Таким образом, если будут получены кадры с 0 по 7 и будет передвинуто окно для приема кадров с 8 по 15, получатель сможет безошибочно отличить повторную передачу (кадры с 0 по 7) от новых кадров (с 8 по 15). Поэтому в протоколе 6 применяется окно размером (MAX_SEQ + 1 )/2.

Возникает новый вопрос: сколько буферов должно быть у получателя? Ни при каких условиях он не должен принимать кадры, номера которых не попадают в окно. Поэтому количество необходимых буферов равно размеру окна, а не

диапазону порядковых номеров. В приведенном выше примере для 4-битовых порядковых номеров требуется восемь буферов с номерами от 0 до 7. Когда прибывает кадр r, он помещается в буфер r mod 8. Обратите внимание на то, что хотя i и (r + 8), взятые по модулю 8, «соревнуются» за один и тот же буфер, они никогда не оказываются в одном окне одновременно, потому что это привело бы к увеличению размера окна по крайней мере до 9.

По этой же причине количество необходимых таймеров также равно числу буферов, а не диапазону порядковых номеров. Таким образом, удобно связать каждый таймер со своим буфером. Когда интервал времени истекает, содержимое буфера высылается повторно.

В протоколе 5 предполагалось, что загрузка канала довольно высока. Когда прибывал кадр, он не подтверждался сразу. Вместо этого подтверждение высылалось на встречном информационном кадре. Если обратный поток информации был невелик, подтверждения задерживались на довольно большой период времени. Если в одном направлении посылалось много

информации, а во встречном — вообще ничего, то высылалось только MAX__SEQ пакетов, после чего протокол останавливался.

В протоколе 6 эта проблема решена. По прибытии кадра с данными процедура start_ack_timer запускает вспомогательный таймер. Если таймер сработает раньше, чем появится кадр с данными для передачи, то будет послан отдельный кадр с подтверждением. Прерывание от вспомогательного таймера называется событием ack_timeout. При такой организации возможен однонаправленный поток данных, так как отсутствие встречных информационных кадров, на которых можно было бы отправлять подтверждения, больше не является препятствием.

Для этого требуется всего один таймер. При вызове процедуры start_ack_timer, если таймер уже был запущен, его параметры переустанавливаются на отсчет полного интервала времени.

Важно, что период времени вспомогательного таймера должен быть существенно короче интервала ожидания подтверждения. При этом условии подтверждение в ответ на полученный правильный кадр должно приходить прежде, чем у отправителя истечет период ожидания и он пошлет этот кадр второй раз.

Протокол 6 использует более эффективную стратегию обработки ошибок, чем протокол 5. Как только у получателя появляются подозрения, что произошла ошибка, он высылает отправителю отрицательное подтверждение (NAK). Получатель может делать это в двух случаях: если он получил поврежденный кадр или если прибыл кадр с номером, отличным от ожидаемого (возможность потери кадра). Чтобы избежать передачи нескольких запросов на повторную передачу одного и того же кадра, получатель должен запоминать, был ли уже послан NAK для данного кадра. В протоколе 6 для этой цели применяется переменная no_nak, принимающая значение true, если NAK для ожидаемого кадра (с номером frame_expected) еще не был послан. Если NAK повреждается или теряется по дороге, в этом нет большой беды, так как у отправителя в конце концов истечет период ожидания положительного подтверждения и он, так или иначе, вышлет недостающий кадр еще раз. Если после того как NAK будет выслан и потерян прибудет не тот кадр, переменной

no_nak опять будет присвоено true и будет запущен вспомогательный таймер. Когда время истечет, будет послано положительное подтверждение (ACK) для восстановления синхронизации текущих состояний отправителя и получателя.

В некоторых ситуациях время, необходимое для прохождения кадра по каналу, его обработки и отсылки обратно подтверждения, остается практически неизменным. В таких ситуациях отправитель может поставить время ожидания несколько больше ожидаемого интервала между отправкой кадра и получением подтверждения. Однако если это время меняется в широких пределах, перед отправителем возникает непростой выбор значения времени ожидания. Если выбрать слишком короткий интервал, то увеличится риск ненужных повторных передач, следовательно, снизится производительность канала. При выборе слишком большого значения протокол будет тратить много времени на ожидания, что также снизит пропускную способность.

В любом случае пропускная способность на что-то тратится. Если встречный поток данных нерегулярен, то время прихода подтверждений также будет непостоянным, уменьшаясь при наличии встречного потока и увеличиваясь при его отсутствии. Переменное время обработки кадров получателем также может вызвать ряд проблем. Итак, если среднеквадратичное отклонение интервала ожидания подтверждения невелико по сравнению с самим интервалом, то таймер может быть установлен довольно «туго» и польза от отрицательных подтверждений (NAK) не очень высока. В противном случае таймер может быть установлен довольно «свободно», и отрицательные подтверждения могут существенно ускорить повторную передачу потерянных или поврежденных кадров.

С вопросом тайм-аутов и отрицательных подтверждений тесно связана проблема определения кадра, вызвавшего тайм-аут. В протоколе 5 это всегда кадр с номером ack_expected, поскольку он является старшим. В протоколе 6 нет столь простого способа определить кадр, интервал ожидания которого истек. Предположим, были переданы кадры с 0 по 4, то есть список неподтвержденных кадров выглядит так: 01234 (от первого к последнему). Теперь допустим, что у кадра 0 истекает интервал

ожидания и он передается повторно, затем посылается кадр 5 (новый), потом интервал ожидания истекает у кадров 1 и 2, и посылается кадр 6 (также новый). В результате список неподтвержденных кадров принимает вид 3405126, начиная с самого старого и заканчивая самым новым. Если весь встречный поток данных потеряется, интервалы ожидания этих семи кадров истекут именно в таком порядке. Чтобы не усложнять и без того непростой пример протокола, мы не показываем подробностей управления таймером. Вместо этого предполагается, что переменной oldest_frame при наступлении тайм-аута присваивается номер кадра, интервал времени которого истек.

```
/* Протокол 6 (выборочный повтор) принимает кадры в любом
порядке, но передает их
сетевому уровню, соблюдая порядок. С каждым неподтвержденным
кадром связан таймер. При
срабатывании таймера передается повторно только этот кадр, а
не все неподтвержденные
кадры, как в протоколе 5. */
#define MAX_SEQ 7 /* должно быть 2^n-1 */
#define NR_BUFS ((MAX_SEQ + 1)/2)
typedef enum{frame_arrival, cksum_err, timeout,
network_layer_ready, ack_timeout} event_type;
#inciude "protocol.h"
boolean no_nak = true; /* отрицательное подтверждение (nak)
еще не посылалось */
seq_nr oldest_frame = MAX_SEQ+1; /* начальное значение для
симулятора */
static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
       /* то же. что и в протоколе 5*/
       return ((a <= b) && (b < c)) || ((c < a) && (a <= b))
|| ((b < c) (c < a));
}
static void send_frame(frame_kind fk, seq_nr frame_nr,
seq_nr frame_expected, packet buffer[])
{
/* сформировать и послать данные, а также положительное или
отрицательное подтверждение */
frame s; /* временная переменная */
s.kind = fk; /* kind == data, ack, или nak */
if (fk == data) s.info = buffer[frame_nr % NR_BUFS];
```

```
s.seq = frame_nr; /* имеет значение только для
информационных кадров */
s.ack = (frame_expected + MAX_SEQ) %(MAX_SEQ + 1);
if (fk == nak) no_iak = false; /* пришел nak */
to_physical_layer(&s); /* переслать кадр */
if (fk == data) start_timer(frame_nr %NR_BUFS);
stop_ack_timer(); /* отдельный кадр с подтверждением не
нужен */
}
void protocol6(void)
{
seq_nr ack_expected; /* нижний край окна отправителя */
seq_nr next_frame_to_send; /* верхний край окна отправителя
+ 1 */
seq_nr frame_expected; /* нижний край окна получателя */
seq_nr too_far; /* верхний край окна получателя + 1 */
int i; /* индекс массива буферов */
frame r; /* временная переменная */
packet out_buf[NR_BUFS]; /* буферы для исходящего потока */
packet in_buf[NR_BUFS]; /* буферы для входящего потока */
boolean arrived[NR_BUFS]; /* входящая битовая карта */
seq_nr nbuffered; /* количество использующихся в данный
момент выходных буферов */
event_type event;
enable_network_layer(); /* инициализация */
ack_expected = 0; /* следующий номер подтверждения во
входном потоке */
next_frame_to_send = 0; /* номер следующего посылаемого
кадра */
frame_expected = 0; /* номер следующего ожидаемого входящего
кадра */
too_far = NR_BUFS; /* верхний край окна получателя + 1 */
nbuffered = 0; /* вначале буфер пуст */
for (i = 0; i < NR_BUFS; i ++) {
        arrived[i] = false
}
while (true) {
        wait_for_event(&event); /* пять возможных событий: см.
event_type выше */

        switch(event) {
                case network_layer_ready: /* получить,
сохранить и передать новый кадр */
```

```
                    nbuffered = nbuffered + 1 ; /* увеличить
окно отправителя */

       from_network_layer(&out_buf[next_frame_to_send %
NR_BUFS]); /* получить новый пакет у сетевого уровня */
                    send_frame(data, next_frame_to_send.
frame_expected, out_buf); /* передать кадр */
                    inc(next_frame_to_send); /* увеличить
верхний край окна отправителя */
                    break;
              case frame_arrival: /* прибыл кадр с данными
или с подтверждением */
                    from_physical_layer(&r); /* получить
пришедший кадр у физического уровня*/
                    if (r.kind == data) {
                          /* кадр прибыл в целости */
                          if ((r.seq != frame_expected) &&
no_nak) {
                                send_frame(nak, 0,
frame_expected, out_buf);
                          }else{
                                start_ack_timer();
                          }
                          if (between(frame_expected,
r.seq. too_far) &&(arrived[r.seq%NR_BUFS] ==false)) {
                                /* кадры могут приниматься
в любом порядке */
                                arrived[r.seq % NR_BUFS] =
true; /* пометить буфер как занятый */
                                in_buf[r.seq % NR_BUFS] =
r.info; /* поместить данные в буфер */
                                while
(arrived[frame_expected % NR_BUFS]) {
                                      /* передать пакет
сетевому уровню и сдвинуть окно */

       to_network_layer(&in_buf[frame_expected % NR_BUFS]);
                                      no_nak = true;

       arrived[frame_expected % NR_BUFS] =false;
                                      inc(frame_expected);
/* передвинуть нижний край окна получателя */
```

```
                                        inc(too_far); /*
передвинуть верхний край окна получателя */
                                        start_ack_timer();
/* запустить вспомогательный таймер на случай, если
потребуется пересылка подтверждения отдельным кадром */
                                    }
                                }
                        }
                        if((r.kind==nak) &&
between(ack_expected,(r.ack+1)%(MAX_SEQ+1),next_frame_to_sen
d)) {
                                send_frame(data, (r.ack+1) %
(MAX_SEQ + 1), frame_expected, out_buf);
                        }
                        while (between(ack_expected, r.ack,
next_frame_to_send)) {
                                nbuffered = nbuffered - 1; /*
отправить подтверждение вместе с информационным кадром */
                                stop_timer(ack_expected %
NR_BUFS); /* кадр прибыл в целости */
                                inc(ack_expectid); /* передвинуть
нижний край окна отправителя */
                        }
                        break;
                case cksum_err:
                        if (no_nak) send_frame(nak, 0,
frame_expected, out_buf);
                        break; /* поврежденный кадр */
                case timeout:
                        send_frame(data, oldest_frame,
frame_expected, out_buf);
                        break; /* время истекло */
                case ack_timeout:
                        send_frame(ack, 0, frame_expected,


out_buf); /* истек период ожидания «попутки» для


подтверждения: послать подтверждение */
        }
        if (nbuffered < NR_BUFS) {
                enable_network_layer();
```

```
        }else {
                disable_network_layer();
        }
}
}
```

**Задания:**

1. Разработать программное обеспечение, для демонстрации работы тривиального симплексного протокола, используя в качестве среды передачи данных файловую систему.
2. Разработать программное обеспечение, для демонстрации работы симплексного протокола с ожиданием, используя в качестве среды передачи данных файловую систему.
3. Разработать программное обеспечение, для демонстрации работы симплексного протокола с контролем ошибок, используя в качестве среды передачи данных файловую систему.
4. Разработать программное обеспечение, для демонстрации работы дуплексного протокола передачи с минимальным размером буффера передачи, используя в качестве среды передачи данных файловую систему.
5. Разработать программное обеспечение, для демонстрации работы конвеерного протокола передачи данных, используя в качестве среды передачи данных файловую систему.
6. Разработать программное обеспечение, для демонстрации работы Дуплексного протокола передачи с выборочным повтором, используя в качестве среды передачи данных файловую систему.
7. Разработать программное обеспечение, для демонстрации работы тривиального симплексного протокола, используя в качестве среды передачи данных механизм сокетов.
8. Разработать программное обеспечение, для демонстрации работы симплексного протокола с ожиданием, используя в качестве среды передачи данных механизм сокетов.
9. Разработать программное обеспечение, для демонстрации работы симплексного протокола с контролем ошибок,

используя в качестве среды передачи данных механизм сокетов.

10. Разработать программное обеспечение, для демонстрации работы дуплексного протокола передачи с минимальным размером буффера передачи, используя в качестве среды передачи данных механизм сокетов.
11. Разработать программное обеспечение, для демонстрации работы конвеерного протокола передачи данных, используя в качестве среды передачи данных механизм сокетов.
12. Разработать программное обеспечение, для демонстрации работы Дуплексного протокола передачи с выборочным повтором, используя в качестве среды передачи данных механизм сокетов.

## *Лабораторная работа №5. Изучение работы программ-снифферов.*

**Цель работы:** изучить основы работы программ-снифферов. При помощи программы-сниффера WareShark осуществить перехват трафика и выполнить его анализ.

**Задачи решаемые с помощью снифферов**

В работе компьютерной сети и сетевого стека узлов иногда возникают проблемы, причины которых трудно обнаружить общеизвестными утилитами сбора статистики (такими например, как netstat) и стандартными приложениями на основе протокола ICMP (ping, traceroute/tracert и т.п.). В подобных случаях для диагностики неполадок часто приходится использовать более специфичные средства, которые позволяют отобразить (прослушать) сетевой трафик и проанализировать его на уровне единиц передачи отдельных протоколов («сниффинг», sniffing).

Анализаторы сетевых протоколов или «снифферы» являются исключительно полезными инструментами для исследования поведения сетевых узлов и выявления неполадок в работе сети. Разумеется, как и всякое средство, например острый нож, сниффер может быть как благом в руках системного администратора или инженера по информационной безопасности, так и орудием преступления в руках компьютерного злоумышленника [1].

Целью атак с использованием сниффера является похищение информации, обычно такой, как идентификационные номера пользователей, данные о функционировании сети, номера кредитных карт и.т.д.

Подобное специализированное программное обеспечение обычно использует «беспорядочный» (promiscuos) режим работы сетевого адаптера компьютера-монитора (в частности, для перехвата трафика сетевого сегмента, порта коммутатора или маршрутизатора). Как известно, суть данного режима сводится к обработке всех приходящих на интерфейс фреймов, а не только

предназначенных MAC-адресу сетевой карты и широковещательных, как это происходит в обычном режиме.

**Применение снифферов на разных сетевых уровнях**

Важно помнить о том, что атаки с использованием сниффера могу затрагивать диапазон от 1 до 7 уровня OSI. Если говорить о физическом соединении, кто-либо, уже имеющий доступ к внутренней локальной сети (чаще всего это работник компании), может использовать программы для прямого захвата сетевого трафика. Применяя техники спуфинга, взломщик, находящийся за пределами атакуемой сети, может вести перехват пакетов на уровне межсетевого экрана и и похищать данные. В последнее время все чаще используется атака, направленная на перехват данных беспроводных сетей, при которой задача атакующего упрощается, так как нужно всего лишь находиться в радиусе действия сети для сбора информации о ней и проникновения в нее.

Использование сниффера в сетях, работающих по протоколу TCP/IP подразумевает захват, декодирование, исследование и интерпретацию данных, передающихся в пакетах по сети.

Для понимания того, для чего взломщики используют снифферы, нам следует знать о том, какие данные они могут получить из сети. Рисунок 1 иллюстрирует уровни OSI и ту информацию, которой взломщик может завладеть на каждом уровне, успешно использовав сниффер.

| Уровень | Действие |
|---|---|
| Прикладной | Захват имен пользователей и паролей |
| Представительский | Захват трафика сессий SSL/TLS |
| Сеансовый | Захват трафика Telnet/FTP |
| Транспортный | Захват TCP-сессий и UDP-трафика |
| Сетевой | Захват IP-адресов и номеров портов |
| Канальный | Захват MAC-адресов и ARP-запросов |

| | |
|---|---|
| Физический | Получение сведений о сети |

Таблицп 5.1. Распределение уровней OSI

Пакет TCP/IP содержит информацию, необходимую для соединения двух сетевых интерфейсов. Он содержит такие поля с информацией об исходном и целевом IP-адресах, номерах портов, номере пакета и типе протокола. Каждое из этих полей является необходимым для функционирования различных уровней сетевого стека и особенно приложений, относящихся к прикладному уровню (уровню 7 OSI), обрабатывающих принятые данные.

По своей природе протокол TCP/IP занимается только тем, что проверяет, сформирован ли пакет, добавлен ли он в Ethernet-фрейм и доставлен ли он от отправителя по сети к адресату. Однако, в этом протоколе не имеется механизмов для контроля безопасности данных. Таким образом, задача по установлению того, не произошло ли вмешательство в передачу данных, перекладывается на высшие уровни сетевого стека.

**Способы перехвата траффика**

В зависимости от того, каково нахождение взломщиков в сети, где производится перехват данных, они используют программы для захвата или программы для исследования пакетов.

В отношении технической стороны захвата пакетов следует помнить о том, что программы, осуществляющие захват пакетов всегда работают в promiscous-режиме, что делает возможным захват и сохранение всех данных, передающихся по сети, с их помощью. Это также означает то, что даже если пакет не предназначается для сетевого интерфейса, на котором работает сниффер, он все равно будет захвачен, сохранен и проанализирован.

В ходе атак, заключающихся в захвате пакетов, используются снифферы, являющиеся либо программным обеспечением с открытым исходным кодом, либо коммерческим программным обеспечением. В целом, существуют три пути перехвата трафика в сети:

- Использование внешней сети для перехвата траффика;

- Использование внутренней сети для перехвата траффика;
- Использование безпроводной сети для перехвата траффика.

**Похищение паролей (Web password sniffing)**

Как становится ясно из названия, в ходе атаки перехватываются данные HTTP-сессий и из них выделяются идентификаторы пользователей и пароли, которые похищаются. Хотя для защиты HTTP-сессий от таких атак и разработан протокол SSL, существует множество сайтов во внутренних сетях, использующих стандартное менее безопасное шифрование. Достаточно просто перехватить данные зашифрованные по алгоритмам Base64 или Base128 и получить пароль, применив специальное программное обеспечение. В современных снифферах присутствует функция захвата и получения информации, передаваемой в рамках SSL-сессий, но использовать эту функцию непросто.

**Похищение TCP-сессий (TCP session stealing)**

Этот метод заключается в простом захвате трафика между IP-адресом отправителя и адресата, проходящего через сетевой интерфейс в promiscous-режиме. Такие подробности, как номера портов, типы служб, порядковые номера TCP и сами данные интересуют взломщиков в первую очередь. После захвата достаточного количества пакетов, опытные взломщики могут самостоятельно создавать TCP-сессии, вводя в заблуждение узлы, являющиеся источником и адресатом пакетов, а также осуществлять атаку перехвата с участием человека (man-in-the-middle) в отношении активной TCP-сессии.

**Захват данных приложений (Application-level sniffing)**

Обычно из захваченных пакетов данных можно получить некоторое количество информации относительно приложений, осуществляющих обмен данными, и на основе этой информации провести другие атаки или просто похитить эту информацию. Например, протокол захвата данных может быть исследован с целью идентификации операционной системы, анализа SQL-запросов, получения информации о TCP-портах, специфических для приложения, и.т.д. С другой стороны, создание списка

приложений, исполняющихся на сервере является хорошим началом атаки в отношении этих приложений.

**Захват данных в локальной сети (LAN sniff)**

Сниффер, работающий во внутренней сети может захватывать данные со всего диапазона IP-адресов. Это помогает злоумышленнику получить данные о функционировании сети, такие, как список активных узлов, список открытых портов, данные об оборудовании серверов и другие. Как только получен список открытых портов, становится возможной атака на основе эксплуатации уязвимостей отдельных служб, работающих на определенных портах [11].

**Захват информации об используемых протоколах (Protocol sniff)**

Этот метод подразумевает захват данных, относящихся к различным протоколам, используемым в сети. Сначала создается список протоколов на основе захваченных данных. Этот список в будущем может использоваться для захвата данных, относящихся к отдельным протоколам. Например, если данных, относящихся к протоколу ICMP не было обнаружено во время захвата, считается, что этот протокол заблокирован. Однако, если при захвате обнаружены UDP-пакеты, отдельный сниффер для UDP-трафика начинает использоваться для захвата и расшифровки трафика, относящегося к Telnet, PPP, DNS и другим приложениям.

**Захват ARP-трафика (ARP sniff)**

В ходе этого популярного метода атаки злоумышленник захватывает как можно больший объем данных для создания таблицы соответствия IP-адресов MAC-адресам. Эта таблица впоследствии может быть использована для подмены ARP-записей (APR poisoning), спуфинг-атак или для эксплуатации уязвимостей маршрутизатора [10].

**Роль снифферов**

Использование сниффера считается типом "пассивной" атаки, при котором атакующие не могут быть замечены в сети. Это обстоятельство затрудняет определение наличия данной атаки и, поэтому, этот тип атаки является опасным.

Использование сниффера помогает взломщикам либо получить информацию непосредственно из сетевого трафика, либо получить данные о работе сети, которые могут быть использованы

для подготовки последующих атак. Взломщики очень часто прибегают к использованию сниффера, так как возможно длительное его использование без риска быть разоблаченным.

Современные снифферы предназначаются для диагностики сетей, но при этом также могут быть использованы и для взлома. В таблице 5.2 приведены основные примеры, описывающую законные и незаконные примеры использования снифферов.

| Примеры законного использования | Примеры незаконного использования |
|---|---|
| Захват пакетов<br>Анализ использования трафика в сети<br>Преобразования пакетов для анализа данных<br>Диагностика сетей | Похищение пользовательских идентификаторов и паролей<br>Похищение данных, относящихся к сообщениям электронной почты и служб мгновенных сообщений<br>Похищение данных с применением спуфинга<br>Похищение средств и причинение ущерба репутации |

Таблица 5.2. Цели использования снифферов

**Сниффер Wireshark**

Утилита Wireshark является широко известным инструментом перехвата и интерактивного анализа сетевого трафика, фактически, стандартом в промышленности и образовании. К ключевыми особенностям Wireshark можно отнести: многоплатформенность (Windows, Linux, Mac OS, FreeBSD, Solaris и др.); возможности анализа сотен различных протоколов; поддержку как графического режима работы, так и интерфейса командной строки (утилита tshark); мощную систему фильтров трафика; экспорт результатов работы в форматы XML, PostScript, CSV и т. д.

Немаловажным фактом является также то, что Wireshark — это программное обеспечение с открытым исходным кодом, распространяемое под лицензией GNU GPLv2, т. е. Вы можете свободно использовать этот продукт по своему усмотрению.

**Установка Wireshark**

Последнюю версию Wireshark для операционных систем Windows и OS X, а также исходный код можно скачать с сайта проекта. Для дистрибутивов Linux и BSD-систем, данный продукт обычно доступен в стандартных или дополнительных репозиториях. Расматриваемые в данном издании снимки экранов сделаны с версии 1.10.10 Wireshark для Windows. Более ранние версии программы, которые можно найти в репозиториях Unix-подобных операционных систем, также можно успешно использовать, так как Wireshark давно уже стабильный и функциональный продукт.

Работа Wireshark базируется на библиотеке Pcap (Packet Capture), предоставляющей собой прикладной интерфейс программирования для реализации низкоуровневых функций взаимодействия с сетевыми интерфейсами (в частности перехвата и генерации произвольных единиц передачи сетевых протоколов и протоколов локальных сетей). Библиотека Pcap является также основой таких известных сетевых средств, как tcpdump, snort, nmap, kismet и т. д. Для Unix-подобных систем Pcap обычно присутствует в стандартных репозиториях программного обеспечения. Для семейства операционных систем Windows существует версия Pcap, которая называется Winpcap. Ее можно скачать с сайта проекта. Впрочем, обычно в этом нет необходимости, так как библиотека Winpcap включена в пакет установки Wireshark для Windows.

Процесс установки программы не сложен для любой операционной системы, с поправкой, разумеется, на специфику используемой Вами платформы. Например, Wireshark в Debian/Ubuntu устанавливается так, что непривилегированные пользователи по умолчанию не имеют права перехватывать пакеты, поэтому программу нужно запускать с использованием механизма смены идентификатора пользователя sudo (или же произвести необходимые манипуляции согласно документации стандартного DEB-пакета).

**Основы работы с Wireshark**

Пользовательский интерфейс Wireshark (рисунок 5.2.) построен на основе библиотеки GTK+ (GIMP Toolkit). Главное окно программы включает следующие элементы: меню, панели

инструментов и фильтров просмотра, список пакетов, детальное описание выбранного пакета, отображение байтов пакета (в шестнадцатеричной форме и в виде текста) и строку состояния:

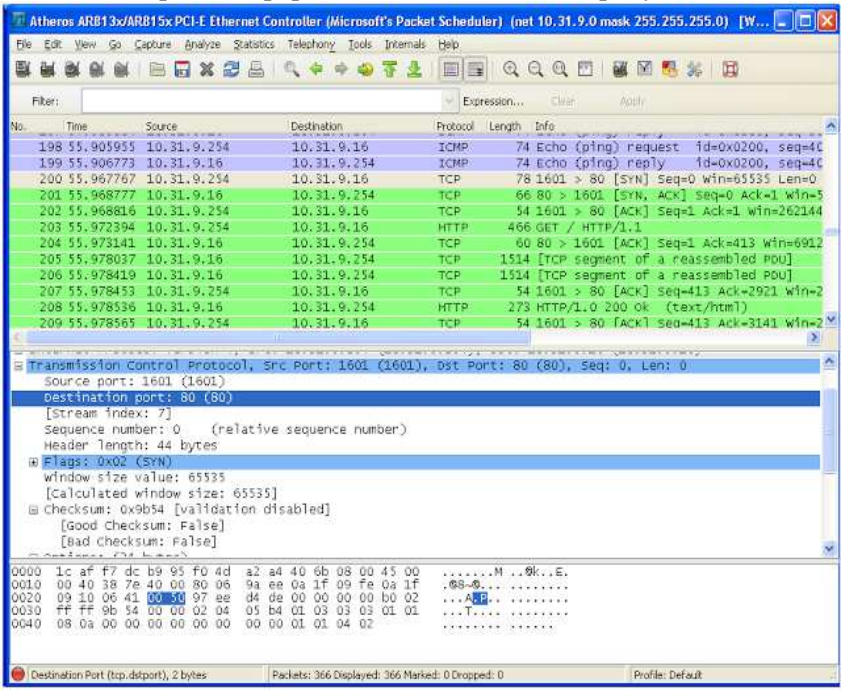


Рисунок 5.2: Основное окно программы Wireshark.

Следует отметить, что пользовательский интерфейс программы хорошо проработан, достаточно эргономичен и вполне интуитивен, что позволяет пользователю сконцентрироваться на изучении сетевых процессов, не отвлекаясь по мелочам. Кроме того, все возможности и подробности использования Wireshark детально описаны в руководстве пользователя. Поэтому мы уделим основное внимание функциональным возможностям рассматриваемого продукта.

Итак, эргономика Wireshark отражает многоуровневый подход к обеспечению сетевых взаимодействий. Все сделано таким образом, что, выбрав сетевой пакет из списка, пользователь получает возможность просмотреть все заголовки (слои), а также

значения полей каждого слоя сетевого пакета, начиная от обертки — кадра Ethernet, непосредственно IP-заголовка, заголовка транспортного уровня и данных прикладного протокола, содержащихся в пакете.

Исходные данные для обработки могут быть получены Wireshark в режиме реального времени или импортированы из файла дампа сетевого трафика, причем несколько дампов для задач анализа можно «на лету» объединить в один.

Проблема поиска необходимых пакетов в больших объемах перехваченного трафика решается двумя типами фильтров: сбора трафика (capture filters) и его отображения (display filters). Фильтры сбора Wireshark основаны на языке фильтров библиотеки Pcap, т.е. синтаксис в данном случае аналогичен синтаксису утилиты tcpdump. Фильтр представляет собой серию примитивов, объединенных, если это необходимо, логическими функциями (and, or, not). Часто используемые фильтры можно сохранять в профиле для повторного использования.

На рисунке 5.3. показан профиль фильтров сбора Wireshark.

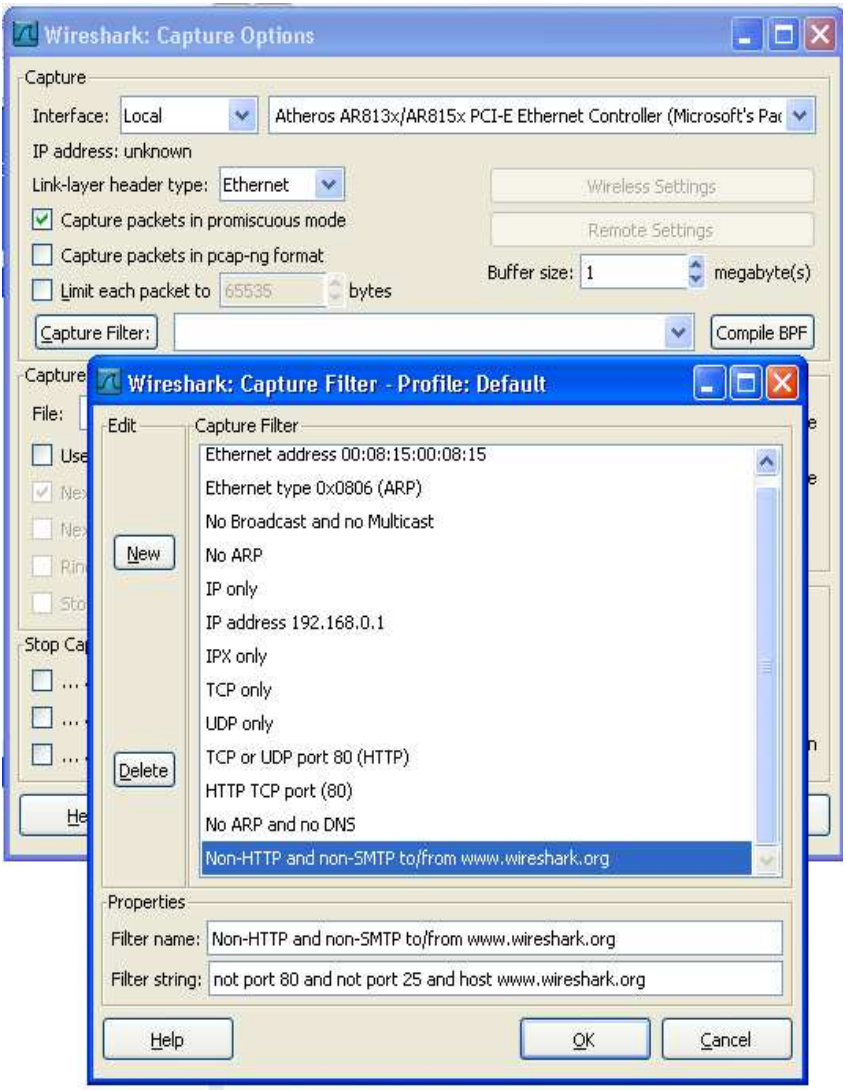


Рисунок 5.3: Работа с профилями фильтров в Wireshark.

Анализатор сетевых пакетов Wireshark также имеет свой простой, но многофункциональный язык фильтров отображения. Значение каждого поля в заголовке пакета может быть

использовано как критерий фильтрации (например, ip.src — IP-адрес источника в сетевом пакете, frame.len — длина Ethernet-фрейма и т.д.). С помощью операций сравнения значения полей можно сопоставлять заданным величинам (например, frame.len < 10), а несколько выражений объединять логическими операторами (например: ip.src==10.0.0.5 and tcp.flags.fin). Хорошим помощником в процессе конструирования выражений является окно настройки правил отображения (Filter Expression) – рисунок 5.4.:

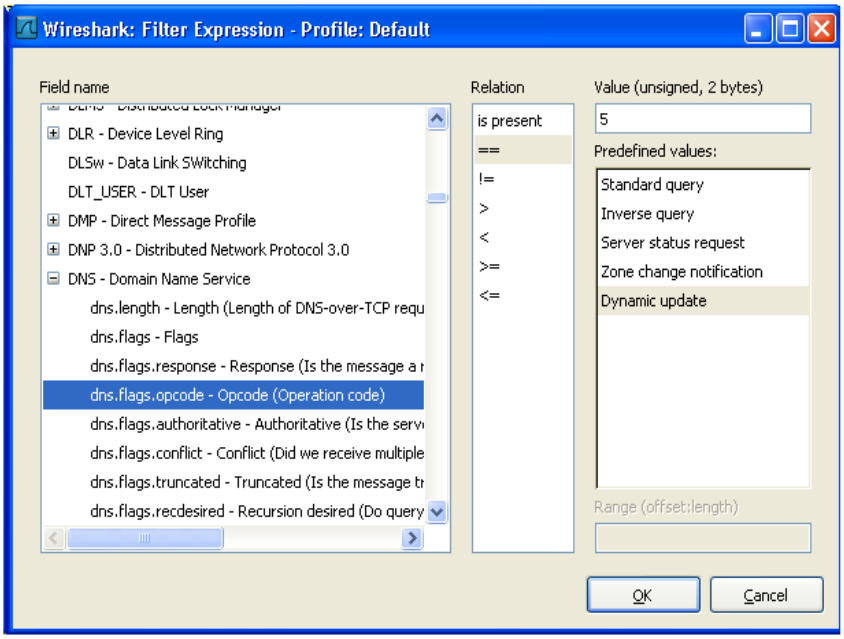


Рисунок 5.4: Конструирование выражений фильтра в утилите Wireshark.

**Средства анализа сетевых пакетов**

Если протоколы без установления соединения возможно исследовать простым просмотром отдельных пакетов и расчетом статистики, то изучение работы ориентированных на соединение протоколов существенно упрощается при наличии

дополнительных возможностей анализа хода сетевых взаимодействий.

Одной из полезных функций Wireshark является пункт «Follow TCP Stream» (буквально, «Следовать за TCP-потоком») подменю анализа «Analyze», позволяющий извлечь данные прикладного протокола из TCP-сегментов потока, которому принадлежит выбранный пакет рисунок 5.5:

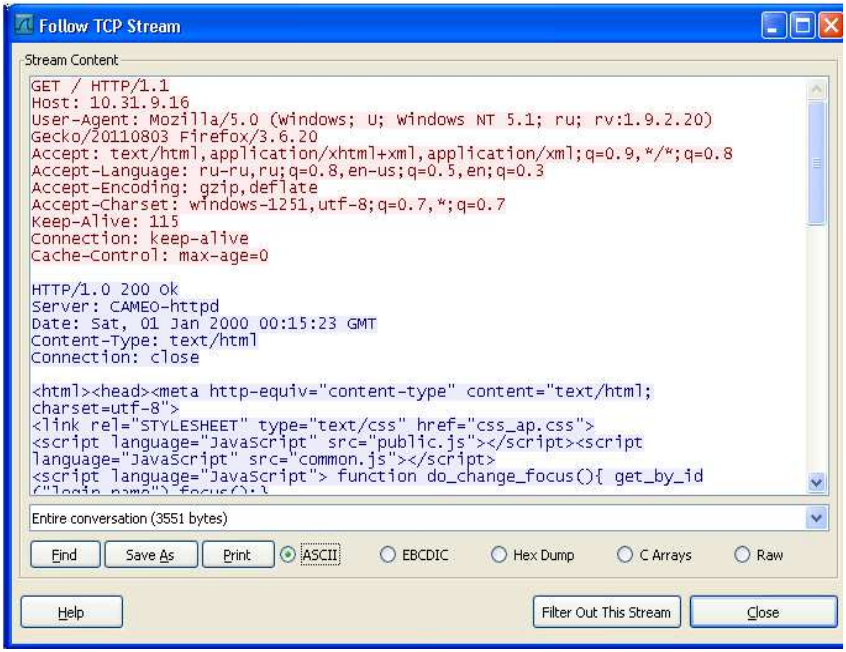


Рисунок 5.5: Извлечение данных из TCP потока в Wireshark.

Еще один интересный пункт подменю анализа - «Expert Info Composite», вызывающий окно встроенной экспертной системы Wireshark, которая попытается обнаружить ошибки и замечания в пакетах, автоматически выделить из дампа отдельные соединения и охарактеризовать их. Данный модуль находится в процессе разработки и совершенствуется от версии к версии программы.

В подменю статистики «Statistics» собраны опции, позволяющие рассчитать всевозможные статистические характеристики изучаемого трафика, построить графики интенсивности сетевых потоков, проанализировать время отклика сервисов и т.д. Так, пункт «Protocol Hierarchy» отображает статистику в виде иерархического списка протоколов с указанием процентного отношения к общему трафику, количества пакетов и байт, переданных данным протоколом.

Функция «Endpoint» дает многоуровневую статистику по входящему/исходящему трафику каждого узла. Пункт «Conversations» (буквально, «разговоры») позволяет определить объемы трафика различных протоколов (канального, сетевого и транспортного уровня модели взаимодействия открытых систем), переданного между взаимодействовавшими друг с другом узлами. Функция «Packet Lengths» отображает распределение пакетов по их длине.

Пункт «Flow Graph...» представляет потоки пакетов в графическом виде. При этом, при выборе элемента на графике становится активным соответствующий пакет в списке в главном окне программы – рисунок 5.6.

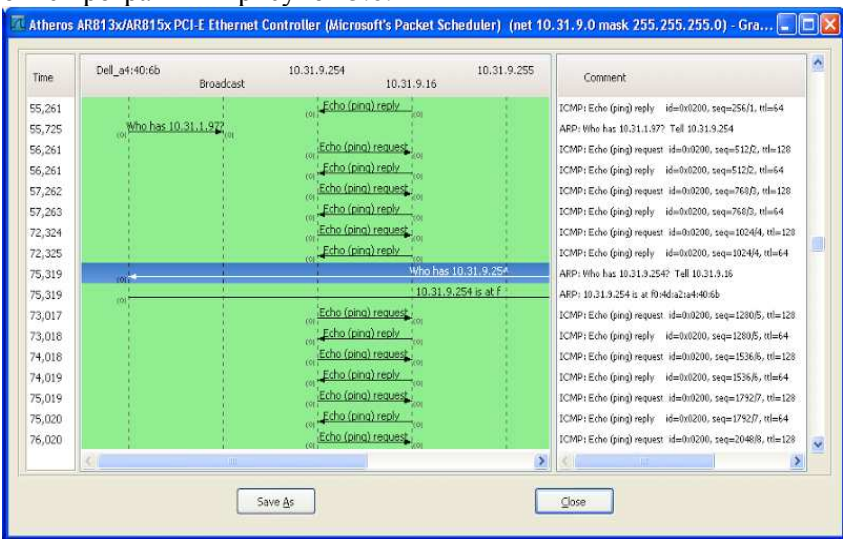


Рисунок 5.6: Графическая визуализация пакетов данных в Wireshark.

Отдельное подменю в последних версиях Wireshark отведено IP-телефонии. В подменю «Tools» есть пункт «Firewall ACL Rules», для выбранного пакета попытается создать правило межсетевого экрана (в версии 1.6.x поддерживаются форматы Cisco IOS, IP Filter, IPFirewall, Netfilter, Packet Filter и Windows Firewall).

Программа также имеет встроенный интерпретатор легковесного языка программирования Lua. Используя Lua, Вы можете создавать собственные «декодеры» протоколов и обработчики событий в Wireshark.

Анализатор сетевых пакетов Wireshark является примером Opensource-продукта, успешного как в рамках платформы Unix/Linux, так популярного среди пользователей Windows и Mac OS X.

Существуют консольные варианты снифферов. Для Windows используется утилита WinDump или TShark, для UNIX-систем – tcpdump [11].

**Задания:**

1. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов с HTTP трафиком и провести его анализ.
2. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов с FTP трафиком и провести его анализ.
3. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов с SMTP трафиком и провести его анализ.
4. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов с POP3 трафиком и провести его анализ.
5. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов приложения, работающего с базой Microsoft SQL и провести его анализ.
6. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить

захват пакетов приложения, работающего с базой MySQL и провести его анализ.

7. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов приложения, работающего с базой PostgreSQL и провести его анализ.
8. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов с любым TCP трафиком и провести его анализ.
9. Освоить интерфейс работы с программой Wireshark. Изучить основные принципы ее работы. Осуществить захват пакетов с любым UDP трафиком и провести его анализ.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

### Обязательная литература

1. Столингс, В. Основы защиты сетей. Приложения и стандарты [Текст] / В. Столингс. – М.: Издательство Вильямс, 2002. – 432 с.

2. Таненбаум, Э. Компьютерные сети. [Текст] / Э. Таненбаум, Д. Уэзеролл 5-е изд. – СПб.: Издательство Питер, 2012. – 960 с.

3. Олифер, В.Г. Компьютерные сети. Принципы, технологии, протоколы. [Текст] / В. Г. Олифер, Н.А. Олифер. 4-е изд. – СПб.: Издательство Питер, 2010. -944 с.

4. Кравец, О.Я. Сети ЭВМ и телекоммуникации [Текст]: учеб. пособие / О.Я. Кравец. – Воронеж : Научная книга, 2010. – 225 с.

5. Новиков, Ю.В. Локальные сети: архитектура, алгоритмы, проектирование [Текст] / Ю.В. Новиков, С.В. Кондратенко. –М. : Издательство «ЭКОМ», 2002. – 560 с.

6. Кузьменко, Н.Г. Компьютерные сети и сетевые технологии [Текст] / Н.Г. Кузьменко – М.: Наука и техника, 2013. – 368 с.

7. Кузин, А.В. Компьютерные сети [Текст] / А.В. Кузин – М. Форум-Инфра-М, 2011. – 192 с.

8. Олифер, В.Г. Основы компьютерных сетей [Текст]: учеб. пособие / В. Г. Олифер, Н.А. Олифер. – СПб.: Издательство Питер, 2009. -352 с.

9. Норткат, С. Обнаружение нарушений безопасности в сетях [Текст] / С. Норткат, Д. Новак. – М.: Издательство Вильямс, 2003. – 448 с.

10. Смит, Р.Э. Аутентификация: от паролей до открытых ключей [Текст] / Р. Э. Смит – М.: Издательство Вильямс, 2002. – 432 с.

11. Собел, М.Г. Linux. Администрирование и системное программирование [Текст] / М.Г. Собел – СПб.: Издательство Питер, 2011. – 880 с.

12. Касперски, К. Секреты поваров компьютерной кухни или ПК: решение проблем [Текст] / К. Касперски.- М. :BHV. 2003. – 360 с.

**Рекомендуемая литература**

1. Авдеев В.А. Периферийные устройства: интерфейсы, схемотехника, программирование. — М.: ДМК Пресс, 2009.

2. Агуров П. Практика программирования USB. - СПб.: Питер, 2006.

3. Алексеев Е.Г., Богатырев С.Д. Информатика. Мультимедийный электронный учебник - http://inf.e-alekseev.ru/text/Processor.html

4. Аппаратные средства персональных компьютеров все про компьютерное железо - http://www.about-pc.narod.ru/part2/proc19.html

5. Архипкин В.Я., Архипкин Я.В. Bluetooth. Технические требования. Практическая реализация. Приложения. – М.: Мобильные коммуникации, 2004.

6. Гладкий А.В. Периферийные устройства. - http://abc.vvsu.ru/Books/lb_perefustr/page0005.asp

7. Гук М., Юров В. Процессоры Pentium 4, Athlon и Duron. - СПб.: Питер, 2002.

8. Гук М.Ю. Аппаратные средства IBM PC. Энциклопедия. 3-е изд. — СПб.: Питер, 2006.

9. Гук М.Ю. Процессоры intel от 8086 до Pentium 4. – СПб.: “Питер Паблишинг”. – 2002.

10. Исследование эффективности совместного использования общего и разделенного L2-кэша современных двухъядерных процессоров. -http://www.ixbt.com/cpu/rmmt-l2-cache.shtml

11. Каган Б.. Электронные вычислительные машины и системы. – М.: Энергоатомиздат, 1991.

12. Калабеков Б.А. Цифровые устройства и микропроцессорные системы – М.: Телеком, 2000.

13. Каталог продукции - http://www.intel.com/cd/products/services/emea/rus/sitemap/384117.htm?iid=subhdr-RU+proddev_all

14. Корнеев В.В., Киселев А.В. Современные микропроцессоры. - СПб.: БХВ-Петербург, 2003.

15. Кравец О.Я. Практикум по вычислительным сетям и телекоммуникациям: Учеб. пособие. - Изд. 4-е, исправл. - Воронеж: Научная книга, 2009.

16. Кравец О.Я. Сети ЭВМ и телекоммуникации: структура и

организация: Учеб. пособие. - Уфа: УГАТУ, 2004.

17. Кравец О.Я. Сети ЭВМ и телекоммуникации: Учеб. пособие. - Воронеж: «Научная книга», 2010.

18. Кравец О.Я., Гараев Р.А. Сети ЭВМ и телекоммуникации: современные технологии: Учеб. пособие. - Уфа: УГАТУ, 2004.

19. Кравец О.Я., Подвальный Е.С., Титов В.С., Ястребов А.С. Архитектура вычислительных систем с элементами конвейерной обработки: Учеб. пособие. – СПб.: Политехника, 2009.

20. Кравец О.Я., Подвальный Е.С., Толпинская Н.Б., Садовой Н.Н. Вычислительные комплексы и системы: компоненты, технологии, реализация: Учеб. пособие. - Ростов н/Д: Издательский центр ДГТУ, 2007.

21. Кравец О.Я., Подвальный Е.С., Хисамутдинов Р.А. Вычислительные комплексы и системы: архитектура, конвейеризация, параллелизм: Учеб. пособие. - Уфа: УГАТУ, 2004.

22. Кравец О.Я., Подвальный Е.С., Хисамутдинов Р.А. Вычислительные комплексы и системы: компоненты, технологии, реализация: Учеб. пособие. - Уфа: УГАТУ, 2004.

23. Кравец О.Я., Сафонов А.И. Методология анализа и проектирования специализированных многозвенных клиент-серверных систем. - Воронеж: «Научная книга», 2010.

24. Новая линейка от Intel. - http://www.intel.com/cd/products/

25. О разрядности процессоров. - http://www.ixbt.com/cpu/cpu-bitness.shtml

26. Опадчий Ю.Ф., Глудкин О.Л., Гуров А.И. Аналоговая и цифровая электроника. – М.: "Горячая Линия - Телеком", 2000.

27. Производство современных процессоров. Технологический экскурс. - http://www.thg.ru/cpu/cpu_production/print.html

28. Рассел Борланд. Running Word 6.0 для Windows (Русская редакция). - М.: ТОО Channel Trading Ltd., 1995.

29. Таненбаум Э. Архитектура компьютеров. - СПб.: Питер, 2007.

30. Токхейм Р. Основы цифровой электроники. - М.: Мир, 1988.

31. Угрюмов Е.П. Цифровая схемотехника. Учеб. пособие для вузов. - БХВ-Петербург, 2004.

32. Ушаков Н.Н. Технология производства ЭВМ. - М.: Высшая школа, 1991.

33. Фигурнов В.Э. IBM PC для пользователя. – М.: «Финансы», 2001.

34. Хорошевский В.Г. Архитектура вычислительных систем. - М.: МГТУ, 2005.

35. Хульцебош Ю. USB в электронике. - СПб.: Питер, 2009.
36. Цилькер Б.Я., Орлов С.А. Организация ЭВМ и систем. - СПб.: Питер, 2006.
37. Цифровая и вычислительная техника. Учебное пособие для высших учебных заведений. - М.; Радио и связь, 1991.
38. Что готовит нам рынок процессоров в 2010 году? - http://analytics.modnews.ru/view/964
39. Шестаков А.П. - http://comp-science.narod.ru/KR/K_1_LR_S.html
40. Экономическая информатика и вычислительная техника/ Под ред. В.П.Косарева, Ю.М.Королева. - М.: Перспектива, 2000.
41. Электронный учебно-методический комплекс Архитектура компьютера - http://arxitektura-pk.26320-004georg.edusite.ru/p31aa1.html
42. Электронный учебно-методический комплекс Архитектура компьютера - http://arxitektura-pk.26320-004georg.edusite.ru/p29aa1.html
43. 3dnews. - http://www.3dnews.ru/news/k-tretemu-kvartalu-2011-goda-bolee-70-protsessorov-intel-budut-32-nm
44. 3dnews. - http://www.3dnews.ru/news/v-2011-godu-amd-imeet-vse-shansi-zanyat-25-rinka-protsessorov/
45. http://modnews.ru/analytics/view/964
46. http://www.voip-tel.ru/technology.html
47. http://ru.wikipedia.org/wiki/ Тактовая_частота
48. http://ru.wikipedia.org/wiki/CISC
49. http://ru.wikipedia.org/wiki/Кэш
50. http://ru.wikipedia.org/wiki/Процессор
51. http://wifi.ab.ru/faq
52. http://www.intersv.ru/user/faq_adsl/
53. http://www.ixbt.com/mainboard.shtml
54. http://www.ixbt.com/video/vidfaq.html
55. http://www.ixbt.com/video/monitor_guide.html
56. http://www.ixbt.com/storage/mdfaq.html
57. http://www.ixbt.com/storage/boot-man1.shtml
58. http://www.ixbt.com/storage/cdfaq.html
59. http://www.lankey.ru/?id=453
60. http://www.network.xsp.ru/3_1.php
61. http://www.network.xsp.ru/3_2.php
62. http://www.xdsl.ru/articles/adsl.htm

## СОДЕРЖАНИЕ



---


Учебное издание

Марк Викторович **Локшин**
Сергей Александрович **Рыков**

**ОСНОВЫ КОМПЬЮТЕРНЫХ СЕТЕЙ. ПРОГРАММИРОВАНИЕ И ПРОТОКОЛЫ. ЛАБОРАТОРНЫЙ ПРАКТИКУМ**

Учебное пособие

Издание публикуется в авторской редакции

Дизайн обложки С.А.Кравец

---

Подписано в печать 16.10.2014. Формат 60x84 1/16.
Усл. печ.л. 8,0. Заказ 000. Тираж 500 экз.

---

ООО Издательство «Научная книга»
394077, Россия, г.Воронеж, ул. 60-й Армии, 25-120
http://www.sbook.ru/

Отпечатано с готового оригинал-макета
в ООО «Цифровая полиграфия»
394036, г. Воронеж, ул. Ф. Энгельса, 52.
Тел.: (473)261-03-61

128