Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)**

Кафедра технологий электронного обучения (ТЭО)

МЕТОДЫ И ТЕХНОЛОГИИ РАЗРАБОТКИ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ

Учебное пособие

ТОМСК

2018

Кручинин Владимир Викторович, Морозова Юлия Викторовна
Методы и технологии разработки клиент-серверных приложений : Учебное пособие. – Томск: Томский государственный университет систем управления и радиоэлектроники, 2018. – 106 с.

Изложены основы разработки сетевых приложений, базирующихся на клиент/серверной модели. Даны основные понятия и логические основы компьютерных сетей. Рассмотрены: организация распределенных многопроцессорных и многопоточных приложений; средства синхронизации потоков и процессов. Приводятся методы и средства межпроцессорного обмена данными (сокеты, удаленный вызов процедур, логические каналы) Показаны примеры построения функциональных расширений WWW-сервера на основе интерфейсов CGI и ISAPI.

© Томский государственный
университет систем управления
и радиоэлектроники, 2018
© Кручинин В.В.,
Морозова Ю.В., 2018

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5
1 КОМПЬЮТЕРНЫЕ СЕТИ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ 6
1.1 Структура сети 6
1.2 Протоколы 8
1.3 Адреса и имена 9
1.4 Основные протоколы транспортного уровня UDP и TCP 10
1.5 Основные службы TCP/IP 10
1.6 Порт 11
2 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРНЫХ СЕТЕЙ 12
2.1 Модель «клиент/сервер» 12
2.2 Операционные системы 14
2.3 Серверное программное обеспечение 14
2.4 Клиентское программное обеспечение 16
3 ПРОГРАММНЫЕ ИНТЕРФЕЙСЫ 17
3.1 Сокеты 17
3.1.1 Основные понятия 17
3.1.2 Основные функции API сокетов 20
3.1.3 Простейшая реализация модели клиент/сервер на основе сокетов 20
3.1.4 Описание API-winsock2 21
3.2 Каналы (Pipes) 27
3.2.1 Создание каналов 27
3.2.2 Создание соединения с помощью именованных каналов 28
3.2.3 Передача данных по именованному каналу 29
3.2.4 Простейший пример реализации модели «клиент/сервер» 30
3.3 Удаленный вызов процедур (RPC – remote call procedure) 31
3.3.1 RPC для открытых систем 31
3.3.2 RPC для Windows 32
3.3.3 Пример создания сетевого приложения на основе RPC Windows 40
4 МНОГОПОТОЧНЫЕ ПРИЛОЖЕНИЯ 49
4.1 Процессы 49
4.2 Потоки (Thread) 52
4.3 Синхронизация потоков 54
4.4 Атомарный доступ 55
4.5 Критические секции 56
4.6 Синхронизация потоков в системном режиме 57
4.6.1 События (Events) 59
4.6.2 Ожидаемые таймеры 61
4.6.3 Семафоры 62
4.6.4 Мьютексы 63
4.7 Пулы потоков 64

4.7.1 Очередь асинхронных вызов функций.. 64
4.7.2 Использование порта завершения ввода/вывода 65
4.7.3 Пример организации пула потоков 66
5 ПРОСТЕЙШЕЕ СЕТЕВОЕ ПРИЛОЖЕНИЕ, ОСНОВАННОЕ НА СОКЕТАХ 68
5.1 Сервер 68
5.2 Клиентское приложение.. 70
6 РАЗРАБОТКА СЕТЕВЫХ ПРИЛОЖЕНИЙ НА ОСНОВЕ WWW-СЕРВЕРА 73
6.1 Обзор технологий 73
6.2 Программирование CGI-скриптов 75
6.2.1 Описание интерфейса 75
6.2.2 Взаимодействие WWW-сервера и CGI-программы 77
6.2.3 Переменные среды о сервере 78
6.3 Программный интерфейс ISAPI.. 82
6.4 Фильтры IIS 86
7 ПРОГРАММИРОВАНИЕ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ PYTHON 90
8 ПРОГРАММИРОВАНИЕ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA 97
ЗАКЛЮЧЕНИЕ 104
ЛИТЕРАТУРА.. 105

ВВЕДЕНИЕ

Развитие современных технических устройств не мыслим без внедрения технологий компьютерных сетей. Компьютерные сети являются необходимым элементом интеграции различного рода устройств и комплексов, в том числе и устройств промышленной электроники. Поэтому программирование компьютерных сетей является важным элементом подготовки современных инженеров и магистров, специализирующихся на проектировании сложных систем промышленной электроники.

Данный курс «Методы и технологии разработки клиент-серверных приложений» предназначен для изучения основ и принципов создания сетевого программного обеспечения, базирующегося на клиент/серверной модели.

Данный курс базируется на курсах «Информатика», «Программирование», «Технологии программирования», «Базы данных», «Компьютерные сети», «Основы вычислительной техники», «Операционные системы», «Введение в интернет»

Курс имеет следующую структуру:

1. Основные понятия и логические основы компьютерных сетей.
2. Организация распределенных многопроцессных и многопоточных приложений.
3. Средства синхронизации потоков и процессов.
4. Средства межпроцессоного обмена данными (сокеты, удаленный вызов процедур, логические каналы).
5. Структура и простой пример организации клиент/сервер-ного приложения.
6. Построение функциональных расширений WWW-сервера на основе интерфейсов CGI и ISAPI.
7. Программирование клиент-серверных приложений на языке Python.
8. Программирование клиент-серверных приложений на языке Java.

Для изучения данного курса необходимо знание и наличие практических навыков программирования на языке программирования C++ и Java для операционной системы MS Windows.

1 КОМПЬЮТЕРНЫЕ СЕТИ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

1.1 Структура сети

Компьютерная сеть – это совокупность компьютеров и других устройств, соединенных с помощью средств телекоммуникации с целью эффективной обработки данных и разделения ресурсов [14]. На рисунке 1.1 показаны основные элементы компьютерной сети.

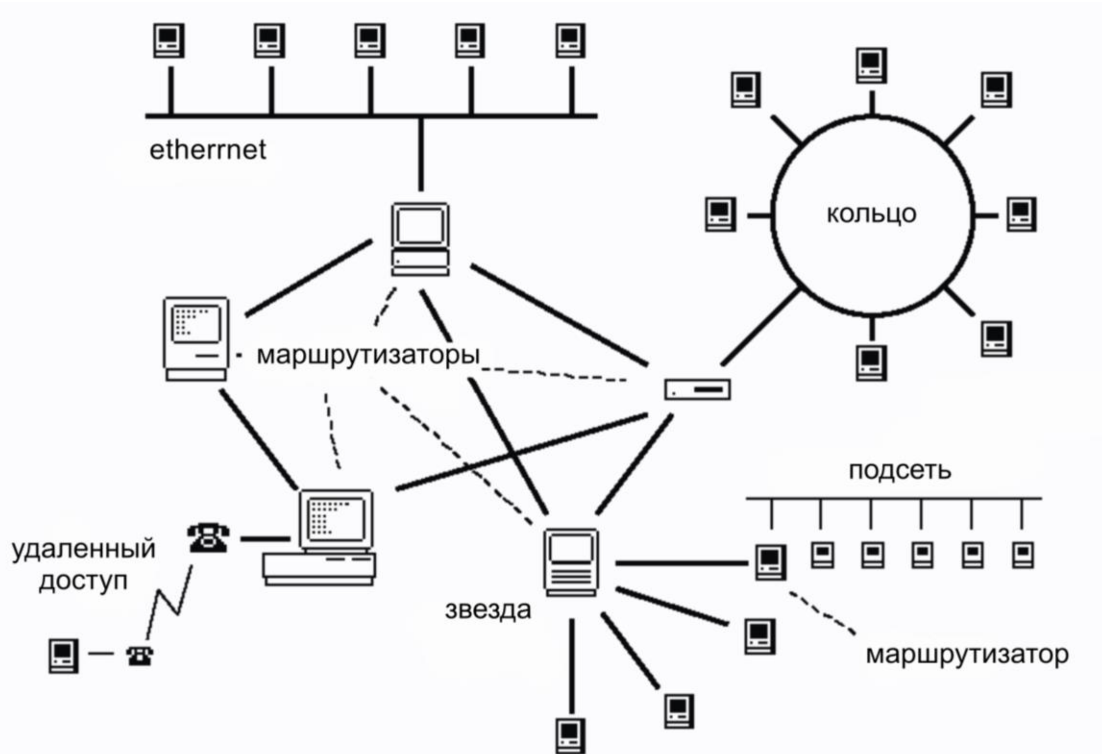

Рис. 1.1 – Структура компьютерной сети

Компьютерная сеть состоит из рабочих станций, серверов, сетевого оборудования, различных устройств (принтер, плоттер, и пр.).

Рабочая станция – это, как правило, персональный компьютер на котором работает клиент (пользователь компьютерной сети).

Сервер – специально выделенный в сети компьютер, обеспечивающий обслуживание запросов некоторого множества клиентов.

Подсеть – компьютерная сеть, входящая в другую компьютерную сеть.

По назначению выделяют:

- файл сервер предназначен для хранения и архивации файлов;
- сервер печати предназначен для управления одним или несколькими принтерами;
- почтовый сервер предназначен для хранения и обработки электронной почты;

- факс сервер предназначен для управления одним или несколькими факсами;
- телефонный сервер обычно обеспечивает связь между телефонной сетью и сетью Интернет;
- прокси сервер предназначен для обеспечения эффективной работы клиента в сети (например, в Интернет через прокси-сервер все пользователи некоторой локальной сети могут иметь доступ в Интернет используя один IP-адрес);
- сервер удаленного доступа обеспечивает доступ к компьютерной сети через модемные или другие линии связи;
- сервер приложений – сервер, обеспечивающий выполнение специфических программ, необходимых для конкретной группы пользователей;
- игровой сервер обеспечивает выполнение сетевых игр;
- веб сервер обеспечивает доступ и манипулирование HTML-документами в сети Интернет.
- сервер баз данных обеспечивает доступ и манипулирование к базам данных.

Сетевые устройства:

- хабы (hub) предназначены для организации рабочей группы;
- мосты (bridge) предназначены для соединения двух сегментов сети и локализации трафика в пределах каждого из них;
- переключатели (switch) предназначены для соединения нескольких сегментов локальной вычислительной сети;
- маршрутизатор (роутер) предназначен для объединения нескольких сетей различной конфигурации.

Классификация компьютерных сетей

Компьютерный сети можно классифицировать по ряду признаков. По размеру, охваченной территории выделяют:

- Локальные сети (LAN, Local Area Network).
- Территориальные сети (MAN, Metropolitan Area Network).
- Глобальные вычислительные сети (WAN, Wide Area Network).

По типу сетевой топологии встречаются:

- Шина.
- Звезда.
- Кольцо.
- Решётка.
- Смешанной топологии.

По архитектуре сети распределяют на:

- одноранговые (peer-to-peer), все компьютеры в сети одинаковые;
- клиент/сервер, когда часть компьютеров являются серверами, а другая – клиентами.

Модель взаимодействия «клиент/сервер» предполагает, что сервер находится в пассивном состоянии ожидания запроса клиентов. Клиент находится в активном состоянии, инициирует обращение к серверу на обслуживание. Затем клиент ждет ответа на посланный запрос, в то время как последний обрабатывается сервером.

1.2 Протоколы

Протоколом в компьютерных сетях называют набор правил для специфического типа связи.

Для обеспечения обмена данными между узлами сети необходимо выполнить следующие действия:

1) произвести пакетирование данных;
2) определить пути пересылки данных;
3) осуществить физическую пересылку;
4) регулировать скорости пересылки данных;
5) обеспечить полную сборку полученных данных, без потерянных частей;
6) осуществить проверку полученных данных на наличие дублированных фрагментов;
7) информировать отправителя о том, сколько было успешно передано данных;
8) обеспечить пересылку данных в нужное приложение;
9) произвести обработку ошибок.

В результате выполнения этих операций программное обеспечение существенно усложняется. Разбивая соответствующие действия по уровням, и записывая их в виде правил, получаем 7 уровневую модель взаимосвязи открытых компьютерных сетей (OSI) (табл. 1.1).

Интернет – всемирная система добровольно объединённых компьютерных сетей, построенная на использовании протокола IP и маршрутизации пакетов данных. Интернет образует глобальное информационное пространство, служит физической основой для Всемирной паутины и множества систем (протоколов) передачи данных.

Таблица 1.1

Уровень	Определение	Протоколы
Физический	Обеспечивает передачу информации в виде физических сигналов	• ISDN • RS-232

Канальный	Обеспечивает формирование и передачу кадров	• Ethernet • Token ring • Fibre Channel • HDLC
Сетевой	Обеспечивает управление передачей пакетов через промежуточные узлы сети	• ICMP • IP • IPX
Транспортный	Обеспечивает управление передачей данными между оконечными пунктами компьютерной сети	• SPX • TCP • UDP • RTCP
Сеансовый Презентационный Прикладной	Обеспечивают поддержание сеанса связи, позволяя приложениям взаимодействовать между собой длительное время.	• DNS • FTP • Gopher • HTTP • IMAP • IRC • LDAP • NTP • NNTP • POP3 • SSH • SMTP • Telnet • SNMP • PPP

1.3 Адреса и имена

Каждый сетевой узел должен иметь имя и адрес. Стратегия присваивания адресов и имен позволяет делегировать соответствующие полномочия. Схема имен и адресов Интернета позволяет: делегировать присвоение имен и адресов; именам отражать логическую структуру организации; присваивать адреса, отражающие топологию физической сети.

Иерархическая структура имен, например, *ie.tusur.ru*, состоит в следующем *ie* – кафедра, *tusur* – организация, *ru* – Российский домен.

В протоколе IP используются IP-адреса, которые идентифицируют рабочие станции, сервера и маршрутизаторы. IP-адрес должен быть уникальным. Соответствие между именем и IP-адресом узла сети осуществля-

ется с помощью специальных баз данных, хранящих пару имя-адрес. IP-адрес представлен четырьмя октетами. Например, 191.200.182.101.

IP протокол – это протокол сетевого уровня, обеспечивающий маршрутизацию данных в Интернете.

Взаимодействия в сети могут быть:

1) без установки соединения, например, как при отправке почтового письма, написал адрес и отправил письмо;

2) с установкой соединения, как в телефоне, позвонил, получил ответ, сделал запрос, получил ответ и т.д. закрыл соединение.

Кроме того, передача данных по сети может быть гарантированной или негарантированной.

1.4 Основные протоколы транспортного уровня UDP и TCP

В таблице 1.2 показаны основные протоколы транспортного уровня UDP и TCP.

Таблица 1.2 – Основные протоколы транспортного уровня UDP и TCP

UDP	TCP
User Datagram Protocol	Transmission control protocol
Не устанавливает соединения	Устанавливает соединение
Не гарантирует получение	Гарантирует получение
Посылает сообщение	Устанавливает потокоориентированную передачу данных
Простота организации	Сложность организации
Имеет преимущества для широковещательных и многоадресных рассылок	Хорошо интегрируется в модель “клиент-сервер”

1.5 Основные службы TCP/IP

Выделяются следующие основные службы TCP/IP:

1. Пересылка файлов осуществляется на основе протокола FTP (File Transfer Protocol). FTP обеспечивает доступ к удаленной файловой системе и позволяет переименовывать, удалять и копировать файлы и каталоги.

2. Удаленный доступ к компьютеру Telnet.

3. Электронная почта. Один из самых распространенных протоколов – простой протокол пересылки почты SMTP (Simple Mail Transfer Protocol).

4. Всемирная паутина WWW (*World Wide Web).*

5. NFS (Network File System) – сетевая файловая система. Обеспечивает доступ клиента к файлу, в таком режиме, если бы файл располагался на компьютере клиента.

6. Новости. Приложения, работающие с новостями для обслуживания локальных досок объявлений BBS (Bulletin Board System).

7. Служба имен DNS-система именования доменов. DNS представляет собой распределенную базу данных для имен и адресов узлов сети, которая распределена серверам данной сети. Протокол DNS разрешает клиенту послать запрос к базе данных локального сервера, а получить ответ от удаленного сервера.

8. Для управления сетью служит SMNP – простой протокол управления сетью. Этот протокол позволяет выявить состояние некоторого узла сети, его текущую нагрузку или получить список доступных в сети служб.

1.6 Порт

IP-адрес предоставляет возможность обращения к конкретному компьютеру сети. Однако, на одном компьютере может быть запущено несколько приложений, каждое из которых может использовать сообщения сети. Для того чтобы различать приложения на одном компьютере вводят понятие порта. *Порт* – это идентификатор, обеспечивающий однозначное соответствие между приложением и сетевыми операциями. Итак, порт это:

- шестнадцатиразрядное целое число;
- число уникальное на данном компьютере;
- IP+порт идентифицирует приложение в сети.

Служба	№ порта	Протокол	Комментарий
echo	7	UDP/TCP	Возвращает принятый символ
discard	9	UDP/TCP	Сброс всех входящих данных
daytime	13	UDP/TCP	Возвращает текущее время
chargen	19	UDP/TCP	Генератор символов
ftp	21	TCP	Порт обмена файлами
telnet	23	TCP	Порт для удаленной регистрации Telnet
smtp	25	TCP	Электронная почта
daytime	37	UDP/TCP	Текущий день
http	80	TCP	World Wide Web

2 ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОМПЬЮТЕРНЫХ СЕТЕЙ

2.1 Модель «клиент/сервер»

Модель «клиент/сервер» является доминирующей архитектурой современных компьютерных сетей. Поэтому в дальнейшем будем рассматривать только эту модель. Соответственно, программное обеспечение такой архитектуры разбивается на два класса: клиентское ПО, предназначенное для создания клиента и серверные ПО – для создания сервера.

Рассмотрим алгоритм работы сервера. Под сервером будем понимать компьютер, оснащенный серверным программным обеспечением. Далее, под сервером будем понимать компьютерную программу, относящуюся к серверному программному обеспечению. Тогда обобщенный алгоритм работы сервера следующий:

1. Инициализация.
2. Ожидание запроса клиента.
3. Создание механизма обработки запроса клиента (организация процесса или потока обработки).
4. Переход на шаг 2.
5. Завершение работы сервера.

На этапе инициализации сервера происходит чтение конфигурационных файлов, где записаны текущие параметры сервера. Это может быть карта текущих каталогов, открытие базы данных аутентификации, установка основного порта, установка параметров пула потоков, открытие журнала регистрации запросов и журнала ошибок.

На этапе ожидания запроса сервер находится в пассивном режиме. После получения запроса от клиента сервер пробуждается и запускает механизм обработки данного запроса.

Это механизм включает:

1) инициализацию механизма обработки;
2) получение запроса от клиента;
3) анализ запроса;
4) планирование обработки запроса;
5) выполнение запроса;
6) формирование ответа;
7) пересылку ответа клиенту;
8) завершение механизма обработки.

Поскольку запросы от клиентов могут поступать в сравнительно короткие промежутки времени (практически одновременно) и в большом количестве, а обработка запроса может длится достаточно долго, то появляется необходимость распараллеливания процессов приема и обработки запросов. Таким образом, одним из главных требований к серверным программам является необходимость организации параллельной обработки

запросов клиентов, а требованием к компьютеру для организации сервера – достаточность вычислительных ресурсов для обработки запросов всех клиентов.

Клиент – это компьютер вычислительной сети, оснащенный клиентским программным обеспечением. Алгоритм работы клиентского программного обеспечения (далее, просто клиент) следующий:

1) инициализация работы;
2) ожидание ввода запроса и его редактирование;
3) ввод адреса (имени) сервера и передача запроса серверу;
4) ожидание ответа от сервера;
5) получение ответа;
6) обработка ответа;
7) вывод ответа клиенту;
8) переход на шаг 2;
9) завершение работы клиента.

Как видно из модели «клиент/сервер», решение некоторой задачи может быть распределено между сервером и клиентом. Причем основная нагрузка по решению может лежать на сервере, а на клиенте – только введение запроса и отображение ответа. В этом случае говорят о тонком клиенте. В другом случае клиент может решать задачу сам и по мере решения обращаться к тем или иным серверам. В этом случае говорят о толстом клиенте. В тех случаях, когда распределение функций между сервером и клиентом производится динамически, то говорят о смешанном клиенте.

В тех случаях, когда клиент или сервер при решении задачи обращается к другим серверам, говорят о многозвенной архитектуре системы. Например, на рисунке 2.1 представлена трехзвенная структура распределенной программной системы, основанной на модели «клиент/сервер».

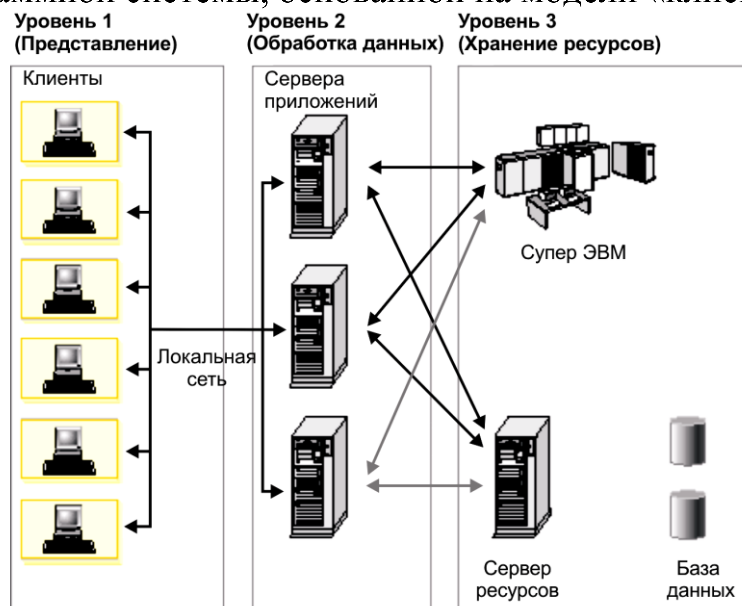

Рис. 2.1 – Трехзвенная организация распределенной программной системы, основанной на модели «клиент/сервер»

2.2 Операционные системы

Операционная система UNIX изначально создавалась для применения в компьютерных сетях.

Linux – это полноценная операционная система семейства UNIX, поддерживающая широкий спектр аппаратных средств, протокол TCP/IP, графический интерфейс пользователя, что позволяет использовать ее не только как сервер, но и как высокопродуктивную рабочую станцию.

Серверные ОС Microsoft: Windows NT 4/0 Server, Windows 2000 Server, Windows server 2003.

Клиентские ОС Microsoft: Windows NT 4/0 Worksration, Windows 2000 Professional, Windows XP Professional и др.

2.3 Серверное программное обеспечение

Telnet (terminal networking) доступ к компьютеру в сети через удаленный терминал (терминальный доступ). Одна из самых ранних сетевых служб, разработанная для UNIX систем, в настоящее время является стандартом для современных операционных систем. Модель протокола telnet показан на рисунке 2.2.

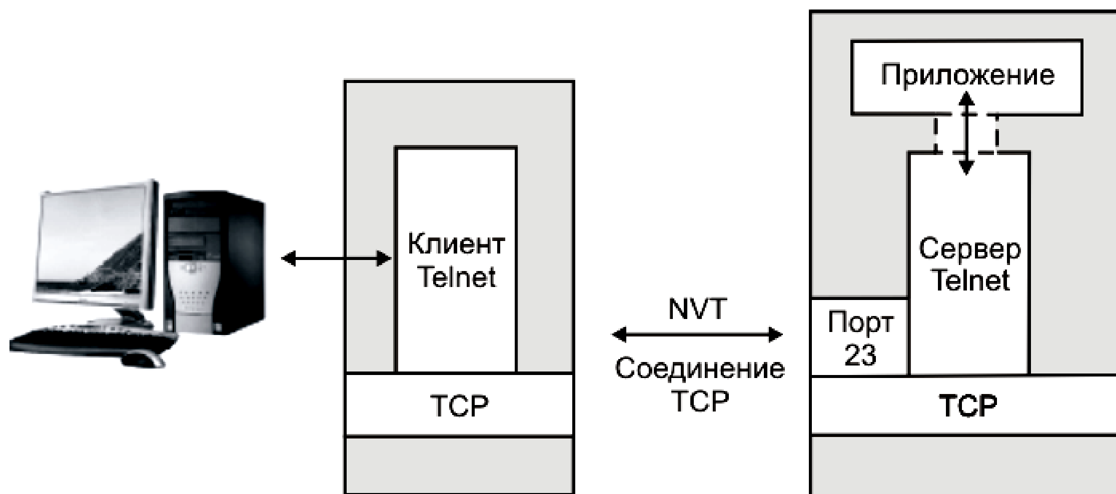

Рис. 2.2 – Модель протокола telnet

Пользователь с удаленного терминала запускает клиента Telnet и указывает ему адрес хоста. Далее, клиент Telnet оправляет запрос на указанный сервер Telnet. Работа клиента и сервера осуществляется по протоколу NVT.

NVT – протокол сетевого виртуального терминала (Network Virtual Terminal).

Веб-сервер – это сервер, принимающий HTTP-запросы от клиентов, обычно веб-браузеров, и выдающий им HTTP-ответы, обычно вместе с

HTML-страницей, изображением, файлом, медиа-потоком или другими данными. *Веб-серверы* – основа Всемирной паутины.

HTTP (Hypertext Transfer Protocol) – протокол пересылки гипертекста. Сам гипертекстовый документ (веб-страница) описывается на языке HTML (Hypertext Markup Language), который описывает:

- заголовки;
- подзаголовки;
- абзацы;
- ссылки с помощью URL;
- списки;
- изображения;
- формы для ввода данных;
- таблицы и формулы.

Клиенты получают доступ к веб-серверу по URL адресу нужной им веб-страницы или другого ресурса.

URL-адрес (универсальный указатель ресурсов) определяет:

- имя ресурса;
- местоположение ресурса;
- используемый протокол.

Архитектура HTTP достаточно проста. Клиент соединяется с сервером, извлекает страницу и закрывает соединение. WWW эффективно работает с текстовыми документами. Однако обрабатываемые изображения, видео и аудиоинформация может быть достаточно большой, поэтому требуется анализировать и оптимизировать объем и тип информации, которую передает WWW-сервер. Кроме того, сервер может вызвать специальные программы, которые генерируют HTML-документы.

Для доступа клиентов к внешним WWW серверам, расположенным в пределах зоны безопасности сети, используются прокси-сервер. Все запросы клиентов пересылаются на прокси-сервер, а он общается с внешним миром и передает ответы WWW-серверов.

Служба WWW реализуется поверх службы TCP. Работа WWW-сервера заключается в следующем:

- клиент соединяется с сервером;
- клиент посылает запрос;
- сервер отвечает на запрос, передает запрашиваемый документ вместе с указанием типа передаваемой информации.

Сервер может подстраиваться соответственно запросам клиента, анализируя возможности клиента, описанные в операторе accept. WWW сервер работает через порт TCP с номером 80. В HTTP используется три стандартных метода приема и передачи информации:

- GET – извлечение страницы;
- HEAD – запрос на вывод заголовка запрашиваемого документа;

- POST – оправка документа на сервер.

Дополнительными функциями многих веб-серверов являются:

- Ведение журнала сервера про обращения пользователей к ресурсам.
- Аутентификация пользователей.
- Поддержка динамически генерируемых страниц.
- Поддержка HTTPS для защищенных соединений с клиентами.

В настоящее время наиболее распространёнными веб-серверами, вместе занимающими около 90% рынка, являются:

- Apache – веб-сервер с открытым исходным кодом, наиболее часто используется;
- IIS – веб-сервер фирмы MicroSoft.

Почтовый сервер

Электронная почта – одна из самых популярных сетевых служб, основных на протоколе TCP/IP. Этот вид связи обеспечивает пользователям простой и удобный способ обмена сообщениями.

Электронная почта широка распространена, поэтому для нее было разработано несколько протоколов:

1. SMTP – простой протокол почтового обмена, является классическим стандартом и разработан для пересылки текстовых сообщений.
2. MIME – протокол многоцелевых почтовых расширений, допускает пересылку самых разнообразных документов.
3. POP – протокол почтового офиса, альтернативный протокол IMAP (протокол доступа к сообщениям Интернета).
4. ESMTP – расширенный SMTP.

Работу с клиентом осуществляет программа, называемая пользовательским агентом (User agent). Пользовательский агент позволяет создавать и манипулировать набором писем клиента, хранящихся в почтовом ящике. Для пересылки писем адресату используется другая программа, называемая агентом пересылки сообщений (MTA-Message Transfer Agent). Почта отправляется от клиента к адресату посредством промежуточных MTA.

2.4 Клиентское программное обеспечение

Для каждой службы TCP/IP было разработано достаточно много разнообразных клиентских программ. Так для просмотра веб-страниц используется браузер, для отправки и приема электронной почты используется программа под называнием «почтовый агент», другие носят названия «клиент Telnet» или «клиент FTP».

3 ПРОГРАММНЫЕ ИНТЕРФЕЙСЫ

3.1 Сокеты

3.1.1 Основные понятия

Объединение IP-адреса и номера порта называется адресом сокета (разъема или вилки). Соединение TCP или UDP полностью идентифицируется адресом сокета. Понятие сокета является фундаментальным для программирования сетевых приложений, поскольку является основой для разработки сетевых приложений и сетевых служб операционных систем.

Сокет обеспечивает идентификацию конечного программного процесса для получения или отправки данных по сети. В модели «клиент/сервер» сокетов должно быть два: сокет на стороне клиента и сокет на стороне сервера.

Рассмотрим модель «клиент/сервер» с использованием сокетов. Первоначально (рис. 3.1) сеть находится в исходном состоянии, на каждом компьютере выполняются некоторые приложения и каждый компьютер имеет IP-адрес.

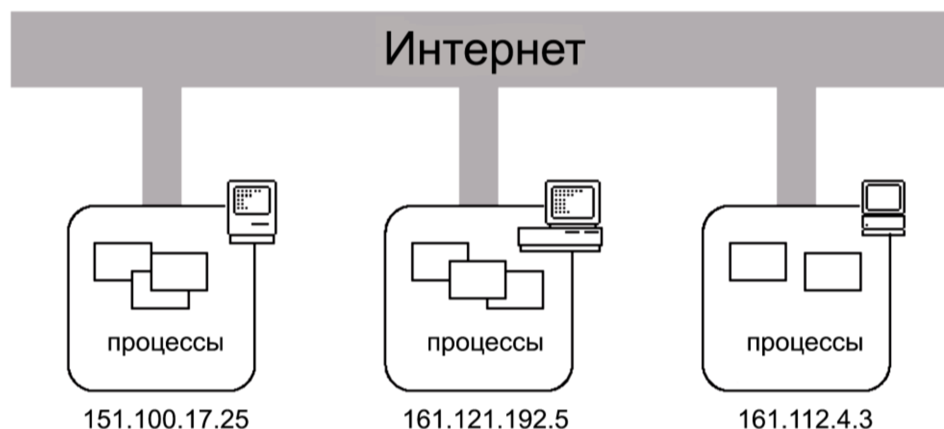

Рис. 3.1 – Исходное состояние сети, все компьютеры имею IP адрес

Далее на компьютере (161.112.192.5) запускается серверный процесс, выделяется порт 21, устанавливается очередь запросов клиентов (рис. 3.2). Серверный процесс переходит в режим ожидания.

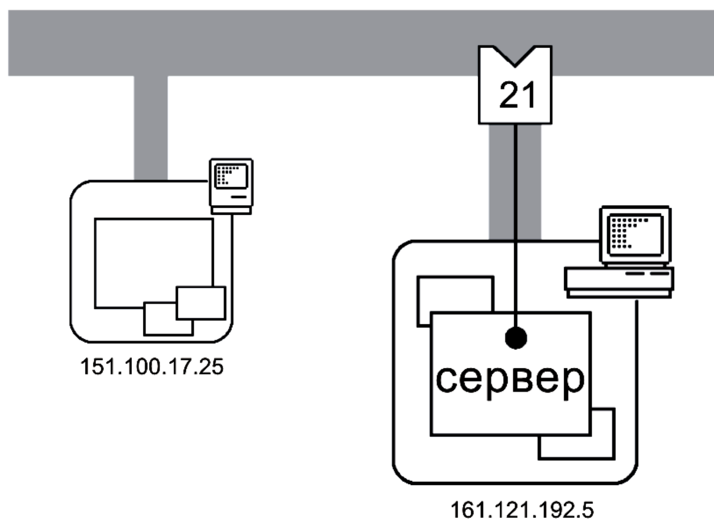

Рис. 3.2 – Компьютер (161.112.192.5) становится сервером, устанавливая очередь запросов по порту 21

Затем (рис. 3.3) на компьютере с адресом (151.100.17.25) запускается клиентский процесс, который по номеру порта 2397 формирует запрос к серверу с адресом (161.112.192.5) и портом 21.

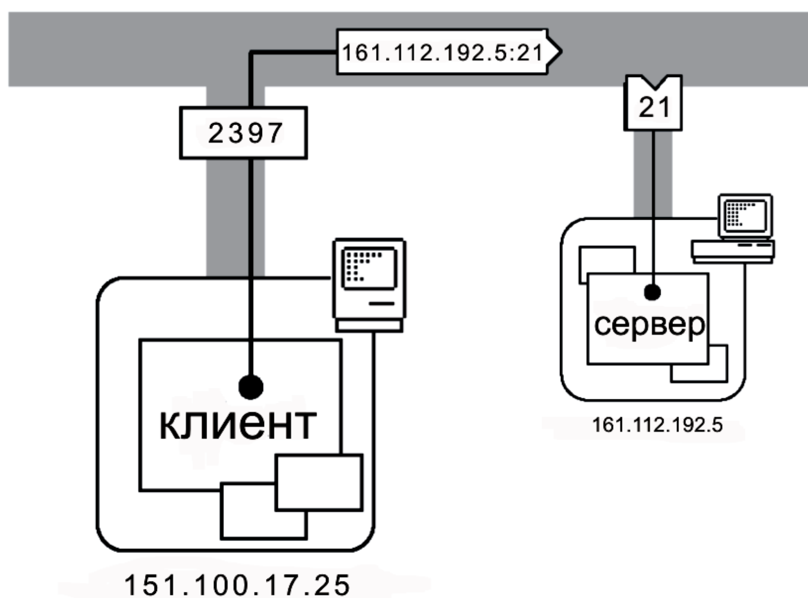

Рис. 3.3 – Компьютер с адресом (151.100.17.25) становится клиентом и делает запрос к серверу (161.112.192.5) порт 21

Производится соединение (рис. 3.4) и сервер запускает дочерний процесс (или поток), который обрабатывает запрос клиента через соединение. Порт 21 снова переходи в режим ожидания.

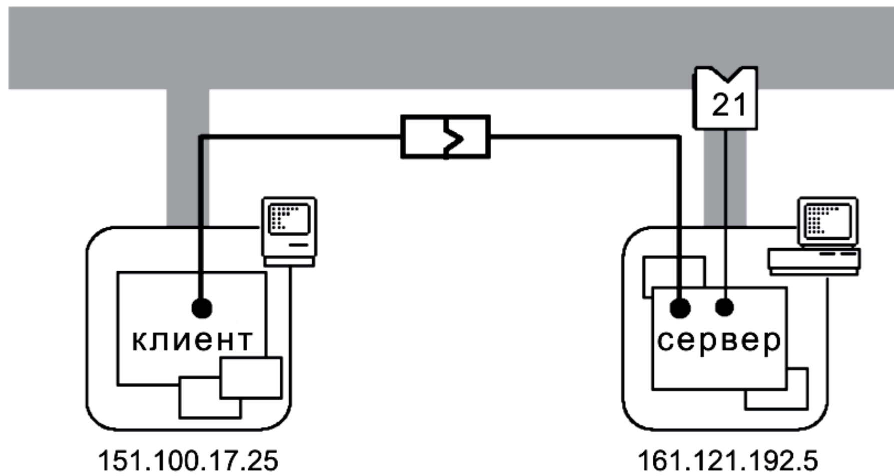

Рис. 3.4 – Организация соединения клиента с сервером

Аналогично производится соединение сервера с другим клиентом (рис. 3.5).

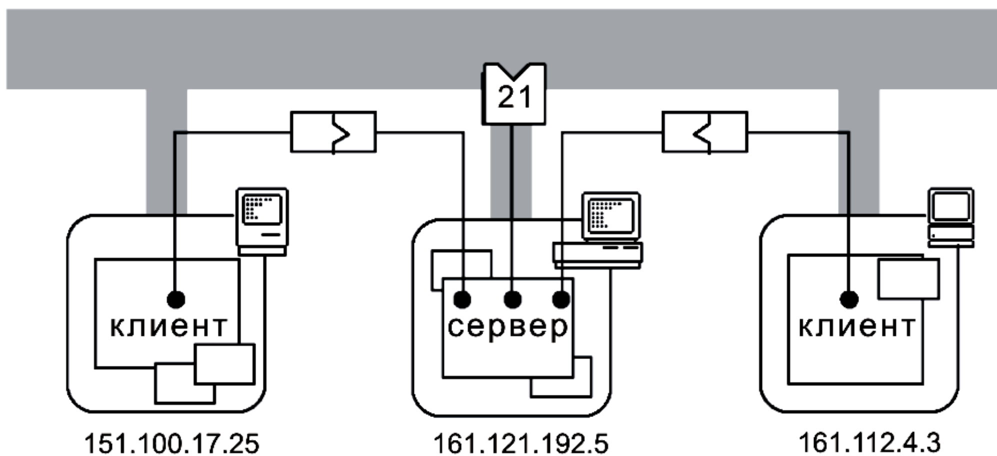

Рис. 3.5 – Соединяются еще несколько клиентов

Сетевые операционные системы (UNIX, Linux, Windows и пр.) поддерживают протоколы TCP/IP и др. и, соответственно, имеют программные интересы (API):

1. UNIX – сокеты.
2. Apple Mac – MacTCP.
3. Windows – интерфейс WinSock.

В операционных системах UNIX службы TCP/IP являются частью ядра операционной системы, в Mac и Windows это расширения соответствующих драйверов и DLL

Основные функции API сокетов делятся на следующие классы:

1) создание, закрытие и удаление сокетов;
2) прием и передача данных;
3) установка и управление соединениями;

4) манипулирование адресами хостов;
5) обслуживание портов;
6) преобразование данных;
7) обработка ошибок;
8) вспомогательные функции.

3.1.2 Основные функции API сокетов

- socket() – создать новый сокет и вернуть его дескриптор;
- bind() – ассоциировать с данным сокетом номер порта и IP-адрес;
- listen() – установить очередь для запросов на соединение;
- accept() – принять запрос на соединение;
- connect() – инициировать соединение с удаленным хостом;
- recv() принять данные от сокета с заданным дескриптором;
- send() – послать данные на сокет, заданный данным дескриптором;
- close() – закрыть соединение с данным сокетом;
- gethostbyname() – получить IP-адрес компьютера по сетевому имени.

3.1.3 Простейшая реализация модели клиент/сервер на основе сокетов

Рассмотрим работу сервера на рисунке 3.6.

1. Функция socket(). Создается сокет, идентифицируется тип связи (TCP, UDP и др), создается структура TCB для управления передачей данных и возвращается дескриптор.
2. Функция bind(). Сервер устанавливает IP-адрес и порт.
3. Функция listen(). Организуется очередь запросов.
4. Организация цикла приема и обработки запросов клиентов:

a. Функция accept() – принимается запрос из очереди. Данная функция создает новый сокет – сокет клиента на сервере и возвращает его дескриптор.

b. В соответствии с логикой соединения сервер передает или принимает данные с помощью функций recv()/send().

c. Клиент в соответствии с этой логикой передает или принимает данные с помощью функций recv()/send().

d. Функция close(). После обмена данными клиент и сервер закрывают соответствующие сокеты.

5. После обработки сервер переходит на обработку следующего запроса, если он есть в очереди или переходит в пассивный режим ожидания.

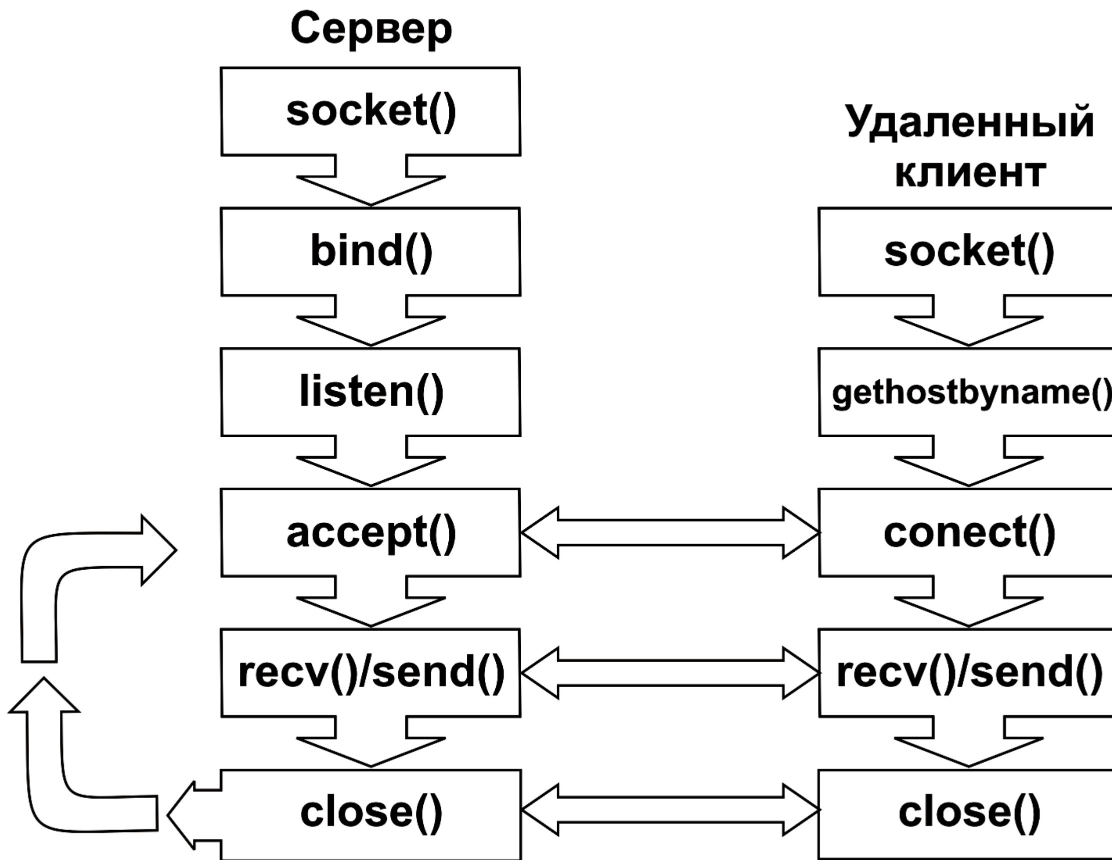

Рис. 3.6 – Простейшая реализация модели клиент/сервер

на основе функций API сокетов

Рассмотрим работу клиента (рис. 3.6).

1. Функция socket(). Создается сокет клиента.

2. Функция gethostbyname() определяет IP-адрес сервера по его имени.

3. Функция connect(). Клиент указывает IP-адрес и порт сервера. TCP пытается соединиться с сервером.

4. Клиент, в соответствии с этой логикой, передает или принимает данные с помощью функций recv()/send().

5. Функция close(). После обмена данными клиент закрывает соответствующий сокет.

Если клиент желает явно определить применяемый далее локальный порт, то он должен вызвать функцию bind(). В нашем случае служба TCP сама определяет номер порта клиента.

3.1.4 Описание API-winsock2

Фирма Microsoft разработала свой интерфейс для работы с сокетами. Это интерфейс имеет имя winsock. Приложение, которое работает с сокетами в ОС Windows должно подключать специальную библиотеку WinSock.DLL.

Для подключения библиотеки необходимо воспользоваться функцией WSAStartup, пример записан ниже:

```
if(WSAStartup(MAKELONG(1, 1), &info) == SOCKET_ERROR) {
MessageBox(NULL,"Could not initialize socket library.",
              "Startup", MB_OK);
    return 1;
     }
```

Для закрытия библиотеки используется функция WSACleanup().

Функция socket()

Функция socket предназначена для создания сокета. Прототип этой функции записан ниже:

```
SOCKET WSAAPI
socket (
IN int af,
IN int type,
IN int protocol
);
```

где af – адресное семейство, которое может иметь следующие значения:

- AF_UNIX – для UNIX систем;
- AF_INET – адресное семейство, поддерживающее взаимодействие по протоколам IPv4;
- AF_INET6 – взаимодействие по протоколам IPv6;

type – тип сокета, может быть установлено два основных типа:

- SOCK_STREAM – потоковый (соответствует протоколу TCP), режим установления соединения.
- SOCK_DGRAM – передача данных в виде датаграмм в режиме без установления соединения (соответствует протоколу UDP).

protocol – конкретный протокол, который используется в данном адресном семействе. По умолчанию его значение равно 0.

Если функция завершилась успешно, то будет возвращено значение нового дескриптора, в противном случае будет возращено значение INVALID_SOCKET. Код ошибки можно узнать, используя функцию WSAGetLastError().

Пример создания сокета:

```
SOKCET s = socket (AF_INET, SOCK_STREAM, 0);
```

Функция bind

Функция bind связывает созданный сокет с локальным адресом. Прототип функции следующий:

```
int WSAAPI
bind (
IN SOCKET s,
IN const struct sockaddr FAR* name,
IN int namelen
);
```

s – дескриптор несвязанного сокета;

name – указатель на структуру, содержащую адрес сокета.

Эта структура определена следующим образом:

```
struct sockaddr {
u_short sa_family;
char sa_data[14];
};
```

namelen – длинна структуры name в байтах.

Например, для протокола TCP/IP необходимо заполнить структуру sockaddr_in, и передать ее в bind.

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

struct sockaddr_in sin;

//очищаем sin
memset ((char *)&sin, '\0', sizeof(sin));
//устанавливаем значения полей
sin.sin_family = AF_INET; //имя семейства адресов
//адрес, устанавливаемый по умолчанию
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = SRV_PORT; //номер порта

//вызываем bind и обрабатываем ошибку
if(bind (s, (struct sockaddr *)&sin, sizeof(sin))!=0)
       {
          code=WSAGetLastError();
          MessageBox(NULL, "ErrBind", "Bind", MB_OK);
          return 0;
       }
```

Если установлено значение адреса INADDR_ANY, то присвоенный адрес можно узнать с помощью функции getsockname().

Функция closesocket()

Функция closesocket() предназначена для закрытия соединения. Прототип функции следующий:

```
int WSAAPI
closesocket (
```

```
IN SOCKET s
);
```

Функция accept()

Функция accept () обеспечивает прием соединения для запроса на соединение от некоторого сокета. Прототип функции записан ниже.

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

где s – дескриптор принимающего сокета; addr – указатель на структуру, в которую будет занесен адрес сокета, запрашивающего соединение (обычно, это сокет клиента); addrlen – указатель, по которому будет записана длина структуры addr.

Если не обнаружено ошибок, то функция accept вернет значение дескриптора сокета для организации передачи данных на сокет клиента. В противном случае, вернет значение INVALID_SOCKET. Код ошибки можно узнать, используя функцию WSAGetLastError().

Например,

```
s_new = accept(s, (struct sockaddr *)&from_sin, &from_len);
```

где s – дескриптор сокета сервера; s_new – дескриптор сокета соединения с клиентом; from_sin – структура, содержащая адрес клиента; from_len – переменная, содержащая длину структуры from_sin.

Функция connect()

Функция connect() предназначена для организации соединения. Прототип этой функции приведен ниже:

```
int WSAAPI
connect (
IN SOCKET s,
IN const struct sockaddr FAR* name,
IN int namelen
);
```

где *s* – дескриптор сокета; *name* – структура, хранящая имя сокета к которому необходимо присоединиться; *namelen* – длина структуры name.

Функция gethostbyname()

Функция gethostbyname() определяет адрес хоста по сетевому имени. Прототип функции следующий:

```
struct hostent FAR * gethostbyname (
```

```
    const char FAR * name
   );
```
где name – имя хоста.

Структура hostent следующая

```
struct hostent {
    char FAR *        h_name;
    char FAR * FAR * h_aliases;
    short            h_addrtype;
    short            h_length;
    char FAR * FAR * h_addr_list;
};
```
где h_name – официальное имя хоста; h_aliases – массив альтернативных имен хоста; h_addrtype – тип адреса; h_length – длина; h_addr_list – список адресов хоста.

Функция gethostbyaddr() возвращает структуру типа hostent указанному адресу машины addr, длина которого равна len, а тип адреса равен type. Прототип функции следующий:

```
struct hostent FAR * gethostbyaddr (
    const char FAR * addr,
    int len,
    int type
   );
```
где addr – адрес хоста; len – длинна адреса в байтах; type – тип адреса может быть только AF_INET.

Функция getsockname()

Функция getsockname() предназначена для получения локального имени сокета. Прототип функции следующий:

```
     int WSAAPI
getsockname (
IN SOCKET s,
OUT struct sockaddr FAR* name,
IN OUT int FAR* namelen
);
```
s – дескриптор сокета; name – структура, которая будет содержать имя сокета; *namelen* – длина структуры в байтах.

Функция listen()

Функция listen() предназначена для установки прослушивания канала связи. Протопип функции следующий:

```
int WSAAPI
listen (
```

```
IN SOCKET s,
IN int backlog
);
```
где s – дескритор сокета; backlog – максимальный размер очереди для запросов на установление соединения.

Например,

```
if(listen(s, 3)!=0)
       {
          code=WSAGetLastError();
          MessageBox(NULL, "ErrListen", "Listen", MB_OK);
          return 0;
       }
```

Функция resv()

Функция resv() осуществляет прием данных от передающего сокета. Прототип функции следующий:

```
int WSAAPI
recv (
IN SOCKET s,
OUT char FAR* buf,
IN int len,
IN int flags
);
```
где s – дескриптор описывающий сокет соединения; *buf* – буфер для приема данных; *len* – длина буфера; *flags* – флаги, определяющие способ приема данных.

Функция send()

Функция send() предназначена для передачи данных на присоединенный сокет. Прототип функции следующий:

```
int WSAAPI
send (
IN SOCKET s,
IN const char FAR * buf,
IN int len,
IN int flags
);
```
где *s* – дескриптор идентифицирующий присоединенный сокет; *buf* – буфер содержащий данные для отправки; *len* – длина данных в буфере *buf*; *flags* – флаги, описывающие способы передачи данных.

Например,

```
send (s, buf, lstrlen(buf), 0);
```

Обработка ошибок

```
code=WSAGetLastError();
```

3.2 Каналы (Pipes)

Каналы (Pipes) обеспечивают простой и надежный способ обмена информацией между двумя приложениями. Программирование с использованием каналов основано на клиент/серверной модели. Обычно серверное приложение создает канал и ждет запросов клиентских приложений. Клиентское приложение присоединяется к этому каналу и посылает запросы.

3.2.1 Создание каналов

В Win32 API имеется два вида каналов: поименованные и анонимные. Использование анонимных каналов ограничено. При создании анонимного канала приложение получает два дескриптора, один из которых затем передается в другое приложение, что необходимо для организации связи.

Наиболее часто анонимные каналы используются для организации передачи информации между родительским и дочерними процессами, или между дочерними процессами одного родительского процесса. Поскольку анонимный канал имеет дескрипторы, но не имеет имени, то он может быть использован только на локальной ЭВМ.

Для взаимодействия приложений, находящихся в сети на разных ЭВМ, необходимо использовать именованные каналы. Имя канала записывается специальным образом:

\\<имя хоста>\\pipe\\<имя канала>.

Например,

\\ws317-2\\pipe\\mypipe

Анонимный канал создается с помощью функции API CreatePipe. Эта функция возвращает два дескриптора, один на чтение, другой на запись.

Именованный канал создается функцией CreateNamedPipe. Для создания клиентского приложения имя канала должно иметь следующую структуру:

hostname\\pipe*pipename*

Поскольку имя канала должно быть связано с данным компьютером, оно имеет специальный вид:

\\.\pipe*pipename*

Имя хоста заменено на точку.

При создании именованного канала можно указать направление передачи данных от сервера к клиенту, от клиента серверу и двунаправленное соединение. Для указания направления передачи данных используется специальный флаг, который имеет следующие значения:

1. PIPE_ACCESS_DUPLEX – двунаправленная передача данных.
2. PIPE_ACCESS_INBOUND – работа канала на прием.
3. PIPE_ACCESS_OUTBOUND – работа канала на передачу.

Именованный канал поддерживает операции асинхронного ввода/вывода, а анонимный канал этот режим работы не поддерживает. Именованный канал может работать в двух модах: байтовой и ориентированной на сообщения. При первой моде передача данных осуществляется как поток байт, при второй моде – как поток сообщений. Канал, открытый как читающий, может работать как в байтовой моде, так и в моде приема сообщений. Именованный канал, открытый с модой потока сообщений, может работать и с потоком сообщений, и с потоком байт. Именованный канал, открытый с модой потока байт, может работать только с потоком байт.

Операция чтения из канала может быть в двух режимах: чтение с блокированием, когда, вызвав функцию чтения, приложение ждет данных из канала и чтение без блокирования, когда приложение не переходит в режим «ожидание» (блокированный и неблокированный ввод/вывод). Режим, в котором работает канал на чтение, может быть установлен или изменен с помощью функции SetNamedPipeHandleState. Для определения текущего режима используется функция GetNamedPipeHandleState. Более подробную информацию об именованном канале можно узнать, вызвав функцию GetNamedPipeInfo.

3.2.2 Создание соединения с помощью именованных каналов

Когда создается анонимный канал, то приложение может передавать данные в оба конца, используя соответствующие дескрипторы. При создании именованного канала серверное приложение открывает только один конец канала, на котором он ждет появление клиентских запросов. Другой конец именованного канала должен открыть клиентское приложение. Структура взаимодействия серверного и клиентского приложений для организации передачи данных с помощью именованного канала показана на рисунке 3.7.

Серверное приложение создает именованный канал, используя функцию CreateNamePipe, и переходит в режим ожидания, вызвав функцию ConnectNamedPipe.

Клиентское приложение с помощью функции WaitNamedPipe может определить, имеется ли для соединения серверный именованный канал. Если канал имеется, то клиентское приложение создает соединение, по-

средством вызвава функцию CreateFile. При этом вместо имени файла указывается имя канала (\\хост\\pipe\\namepipe).

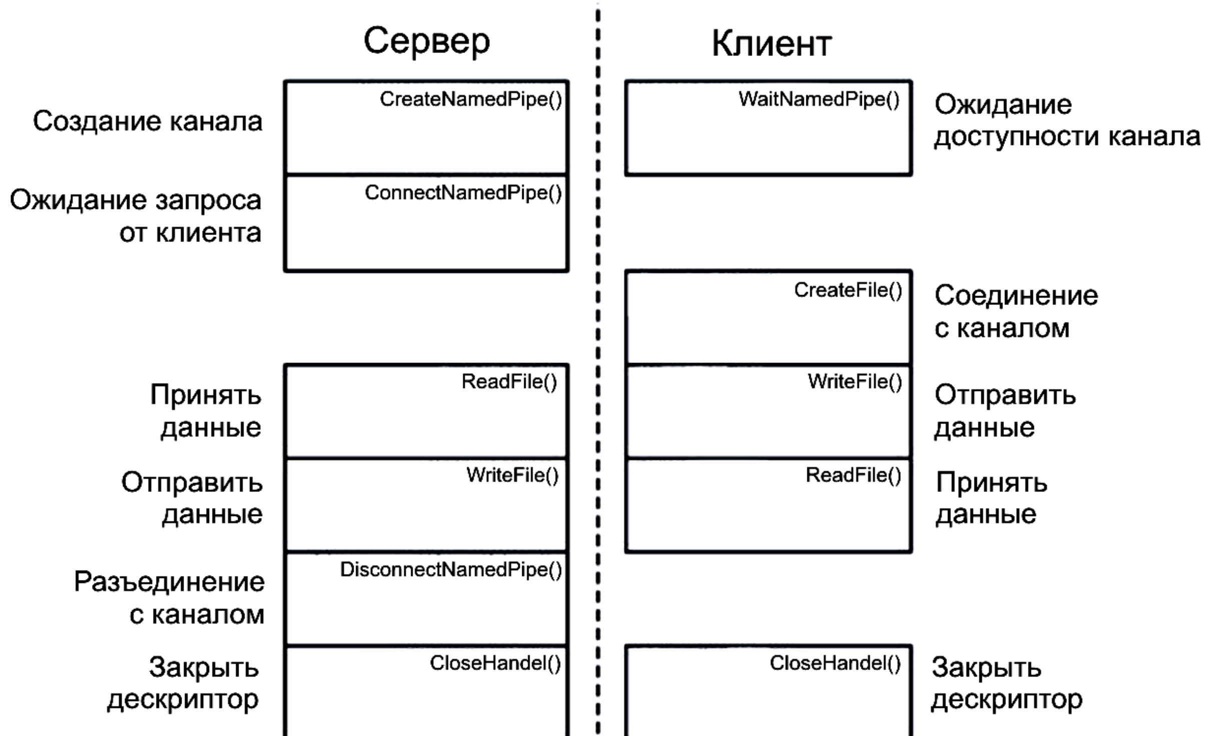

Рис. 3.7 – Структура взаимодействия серверного и клиентского приложений

Как только у клиента срабатывает CreateFile, у серверного приложения срабатывает функция ConnectNamePipe. Далее сервер и клиент могут обмениваться данными с помощью функций ReadFile и WriteFile.

Для завершения соединения серверное приложение вызывает функцию DisconnectNamedPipe. Как только срабатывает эта функция происходит разрыв соединения. Если клиент за этот период не принял все данные, то не принятая часть будет потеряна. Если необходимо чтобы данные клиентом были приняты все до разрыва соединения, на сервере используется функция FlushFileBuffers, обеспечивающая передачу всех данных клиенту. После вызова функции DisconnectNamedPipe серверное приложение вновь может перейти в режим ожидания запроса клиента, вызвав функцию ConnectNamedPipe.

После разрыва соединения, клиентское приложение должно вызвать функцию CloseHandle, чтобы обеспечить закрытие соответствующего именованного канала.

3.2.3 Передача данных по именованному каналу

Передача данных по каналу в блокированном режиме производится с помощью функций ReadFile и WriteFile. Для использования асинхронного режима необходимо использовать функции ReadFileEx и WriteFileEx.

Для быстрой и эффективной передачи сообщений с помощью именованного канала необходимо установить соответствующую моду и использовать функцию TransactNamedPipe, которая на стороне клиента производит запись в канал и осуществляет чтение из канала.

3.2.4 Простейший пример реализации модели «клиент/сервер»

Рассмотрим пример создания серверного и клиентского приложений для передачи сообщения «Hello». Листинг серверного приложения будет следующий:

```
#include <windows.h>
void main(void)
{
    HANDLE hPipe;
    DWORD dwRead;
    char c = -1;
    //создаем именованный канал \\pipe\\hello односторонний,
   // будем ждать запросов от клиента
    hPipe = CreateNamedPipe("\\\\.\\pipe\\hello",
        PIPE_ACCESS_INBOUND, PIPE_WAIT,
PIPE_UNLIMITED_INSTANCES,
          256, 256, 1000, NULL);
   //ожидаем соединения
    ConnectNamedPipe(hPipe, NULL);
    //читаем байты от клиента
    while (c != '\0')   {

        ReadFile(hPipe, &c, 1, &dwRead, NULL);
        if (dwRead > 0 && c != 0) putchar ;
    }
    // разрываем соединение
    DisconnectNamedPipe(hPipe);
    //закрываем серверный канал
    CloseHandle(hPipe);
}
```

Серверное приложение создает именованный канал «\\pipe\\hello» в режиме блокирования (PIPE_WAIT) и односторонней передачей данных (PIPE_ACCESS_INBOUND) от клиента серверу. Далее сервер вызывает функцию ConnectNamedPipe и переходит в режим ожидания для обработки запросов от клиентов.

Обнаружив соединение, сервер пытается прочитать данные из канала от клиента, выводит прочитанные байты в стандартный вывод (standard output). После записи данных сервер разрывает соединение и закрывает дескриптор канала.

Ниже приведен листинг простого клиента.

```
#include <windows.h>
#include <string.h>
#include <stdio.h>
void main(int argc, char *argv[]){

    HANDLE hPipe;
    DWORD dwWritten;
    char *pszPipe;
   if (argc != 3)   { //если неверное число параметров
   printf("Делай: %s <имя хоста> <строка печати>\n", argv[0]);
        exit(1);
    }
//распределяем память
    pszPipe = malloc(strlen(argv[1]) + 14);
    sprintf(pszPipe, "\\\\%s\\pipe\\hello", argv[1]);
//ожидание ответа от сервера
    WaitNamedPipe(pszPipe, NMPWAIT_WAIT_FOREVER);
//создается соединение
    hPipe = CreateFile(pszPipe, GENERIC_WRITE, 0, NULL,
                  OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
//освобождается память
    free(pszPipe);
//передаются данные на сервер
    WriteFile(hPipe, argv[2], strlen(argv[2])+1, &dwWritten,
NULL);
//закрывается канал
    CloseHandle(hPipe);
}
```

Вызов программы для передачи данных:

```
pipec MYHOST "Hello, World!"
```

3.3 Удаленный вызов процедур (RPC – remote call procedure)

3.3.1 RPC для открытых систем

Впервые механизм удаленного вызова процедур был разработан компанией Sun Microsystems, который опубликован в документе RFC 1831. Основная схема удаленного вызова показана на рисунке 3.8. Клиентская программа производит вызов удаленной процедуры, формируя запрос. Этот запрос посылается на сервер, где производится анализ запроса и вызов соответствующей процедуры. Результат работы удаленной процедуры преобразуется в ответ, клиентская программа получает ответ, преобразует его в нужный формат и далее продолжается выполнение программы.

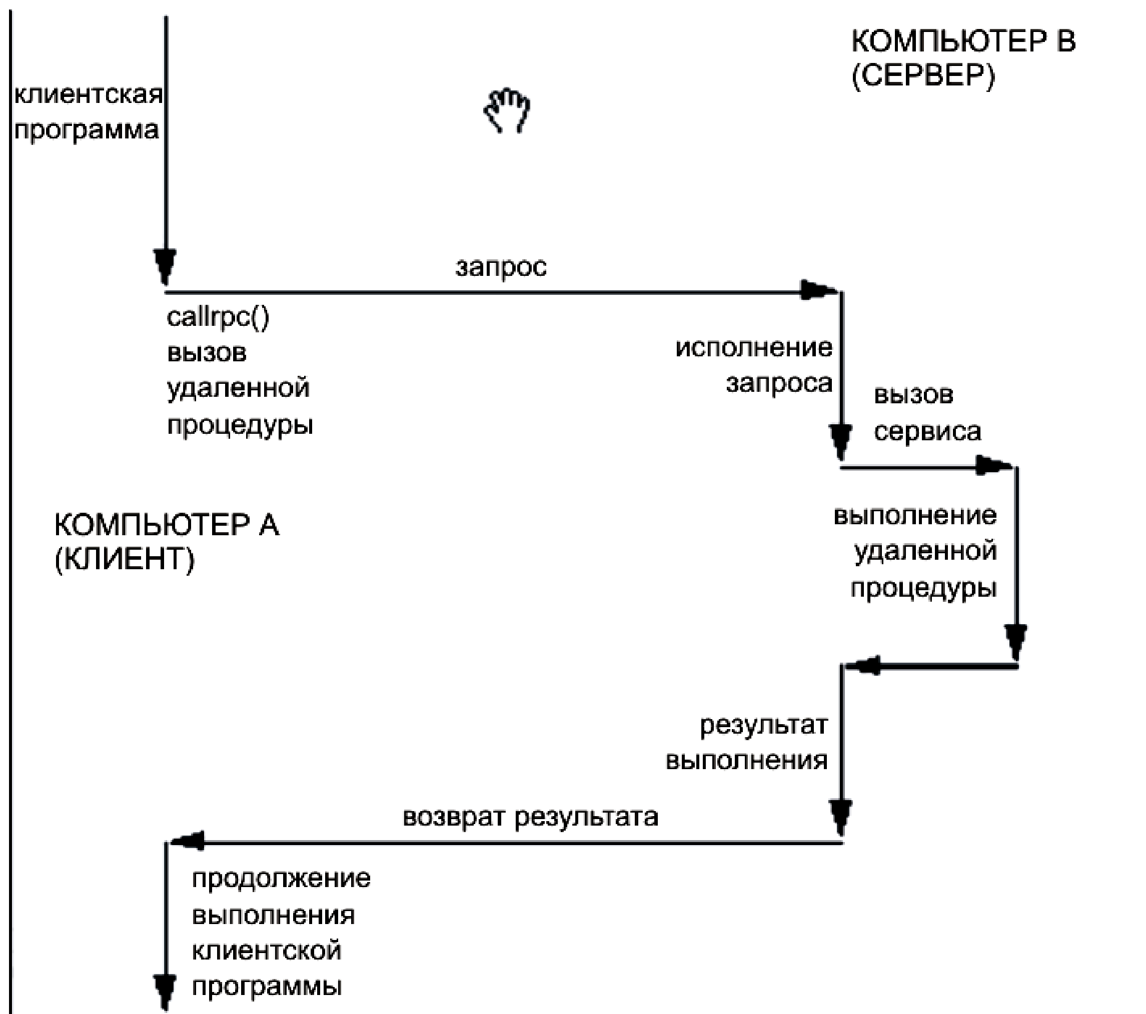

Рис. 3.8 – Основная схема удаленного вызова процедур
для открытых систем

3.3.2 RPC для Windows

Удаленный вызов процедур (далее RPC) – специальный механизм сетевых операционных систем, позволяющий вызывать процедуры, размещенные на удаленных серверах. Клиент и сервер находятся в разных адресных пространствах. Это означает, что обычный механизм вызова процедур в этом случае нельзя использовать, поскольку для размещения параметров у каждого процесса имеется своя собственная память. На рисунке 3.9 представлена архитектура механизма работы RPC.

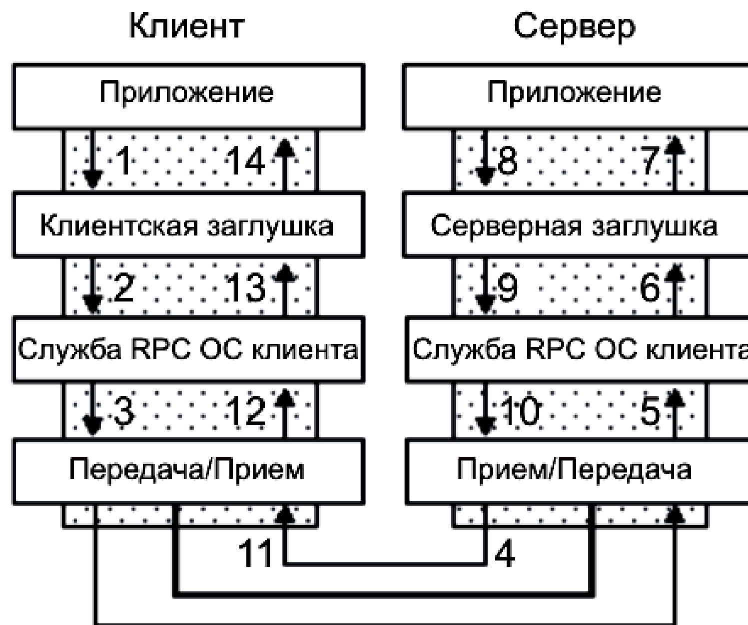

Рис. 3.9 – Архитектура механизма работы RPC

Для вызова удаленной процедуры у клиентского процесса имеется специальная заглушка (Client Stub), которая:

- принимает адреса параметров в адресном пространстве клиентского процесса;
- передает значения входных параметров;
- ожидает передачи выходных параметров от сервера;

Серверное приложение выполняет следующие шаги по вызову процедуры:

1. Системное программное обеспечение службы RPC (Server Runtime Library) принимает запрос от клиента и вызывает серверную заглушку удаленный процедуры.
2. Серверная заглушка принимает значения входных параметров из сетевого буфера и производит их преобразование в заданный формат удаленной процедуры.
3. Серверная заглушка вызывает необходимую процедуру данного сервера.

Затем выполняется удаленная процедура на сервере. Если есть выходные параметры, то процедура формирует их. После того как удаленная процедура завершила свое выполнение, производятся следующие шаги:

1. Удаленная процедура возвращает управление и выходные параметры в серверную заглушку.
2. Серверная заглушка преобразует выходные параметры и посредством службы RPC сервера передает их в сеть.

Клиентский компьютер завершает процесс приема выходных данных от сервера и передачу их в клиентское приложение:

1. Службы RPC (Clientr Run-time Library) принимают выходные данные от удаленной процедуры.

2. Клиентская заглушка принимает выходные данные, преобразует их и записывает в память в адресном пространстве клиентского приложения.

3. Клиентская заглушка возвращает управление клиентскому приложению.

3.3.2.1 Подготовка сервера к соединению

При запуске серверное приложение должно выполнить следующие шаги (рис. 3.10):

1. Зарегистрировать данный интерфейс удаленной процедуры в службах RPC сервера.

2. Создать информацию о соединении, на основе которой осуществляется вызов удаленной процедуры.

3. Зарегистрировать конечную точку.

4. Организовать прослушивание для обнаружения запросов клиентских вызовов.

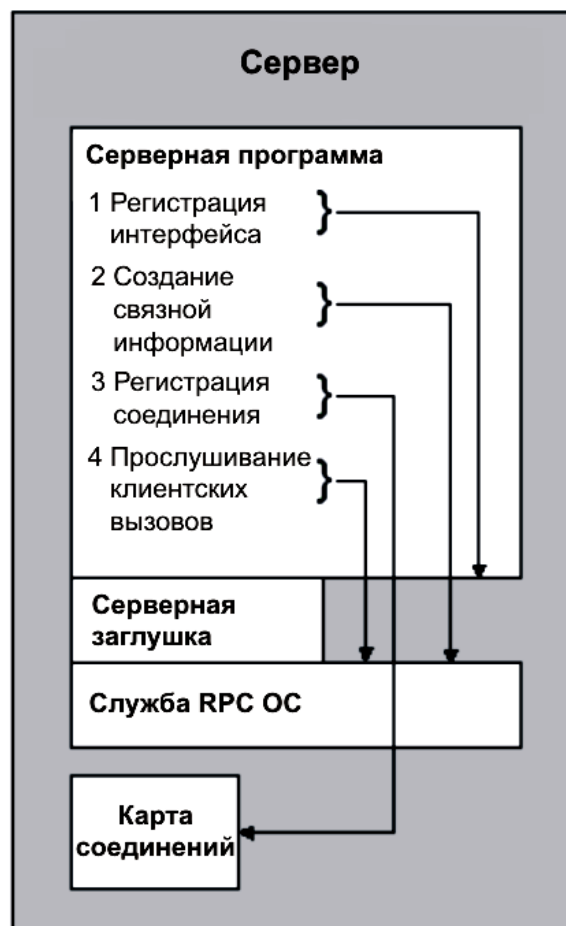

Рис. 3.10 – Подготовка сервера к соединению

Рассмотрим подробнее каждый из перечисленных шагов.

1. Регистрация интерфейса

Для регистрации интерфейса механизма вызова удаленных процедур необходимо вызвать функцию **RpcServerRegisterIf**. Например,

```
RPC_STATUS status;
status = RpcServerRegisterIf(MyInterface_v1_0_s_ifspec, NULL,
NULL);
```

Первый параметр этой функции – структура интерфейса, сгенерированная компилятором MIDL на основе файла описания интерфейса (расширение .idl). Остальные два параметра обычно равны NULL.

Необходимо отметить, что MIDL генерирует две структуры, одну для клиента, другую для сервера. Структура, передаваемая в RpcServerRegisterIf, должна быть серверной.

2. Установка механизма связывания

Служба RPC Windows позволяет использовать различные механизмы связи между различными процессами в сети. Для указания механизма связывания между клиентом и сервером Служба RPC Windows имеет некоторое множество функций. Большинство серверных программ могут использовать все протоколы, имеющиеся в сети. Для указания на то, что сервер использует некоторый механизм, служит функция **RpcServerUseAllProtseqs**. Например,

```
RPC_STATUS status;
status = RpcServerUseProtseq(
    L"ncacn_ip_tcp",  //используется протокол TCP/IP
    RPC_C_PROTSEQ_MAX_REQS_DEFAULT,    //  параметр, зависящий
от протокола
    NULL);// здесь всегда NULL.
```

Первый параметр необходим для указания протокола. Например, для протокола TCP/IP необходимо указать строку «ncacn_ip_tcp», для локального использования указать строку «ncalrpc».

Второй параметр служит для установки параметра соответствующего протокола. Значение RPC_C_PROTSEQ_MAX_ REQS_DEFAULT означает, что по умолчанию используется значение параметра, имеющееся в службе RPC Windows.

Третий параметр необходим для указания параметров защиты. Если никакой защиты не требуется, то он равен NULL.

Для указания протоколов можно также воспользоваться функциями RpcServerUseAllProtseqs, RpcServerUseProtseqEx, RpcServerUseProtseqEp, или RpcServerUseProtseqEpEx.

После указания протокола связи в службе RPC Windows необходимо установить конечную точку соединения (endpoints). Для этого необходимо

запросить таблицу таких точек с помощью вызова функции RpcServer-InqBindings. Например:

```
RPC_STATUS status;
RPC_BINDING_VECTOR *rpcBindingVector;
status = RpcServerInqBindings(&rpcBindingVector);
```

Эта функция возвращает указатель rpcBindingVector на структуру RPC_BINDING_VECTOR, которая динамически создается данной функцией. При завершении работы сервера необходимо освободить память с помощью вызова функции RpcBindingVectorFree. Описание структуры дано ниже:

```
#define rpc_binding_vector_t RPC_BINDING_VECTOR
typedef struct _RPC_BINDING_VECTOR
{
  unsigned long Count;
  RPC_BINDING_HANDLE BindingH[1];
} RPC_BINDING_VECTOR;
```

3. Регистрация конечного соединения

Регистрация конечного соединения для данной серверной программы осуществляется с помощью функции RpcEpRegister. Например,

```
RPC_STATUS status;
status = RpcEpRegister(
    MyInterface_v1_0_s_ifspec,
    rpcBindingVector,
    NULL,
    NULL);
```

Здесь первый параметр – структура данных описания зарегистрированного интерфейса, второй параметр – указатель на таблицу конечных точек соединений.

3.3.2.2 Обслуживание клиентских вызовов

После регистрации интерфейса, формирования необходимой информации для создания соединений с клиентами, следует запустить механизм обслуживания клиентских вызовов. Этот механизм запускается с помощью вызова функции RpcServerListen. Например,

```
RPC_STATUS status;
status = RpcServerListen(
    1,
    RPC_C_LISTEN_MAX_CALLS_DEFAULT,
    0);
```

3.3.2.3 Соединение клиента с сервером

Рассмотрим организацию работы клиентской программы для вызова удаленной процедуры. Клиентская программа должна выполнить следующие шаги (рис. 3.11):

1) создать дескриптор соединения;
2) вызвать клиентскую заглушку;
3) найти сервер;
4) послать запрос;
5) ожидать ответа.

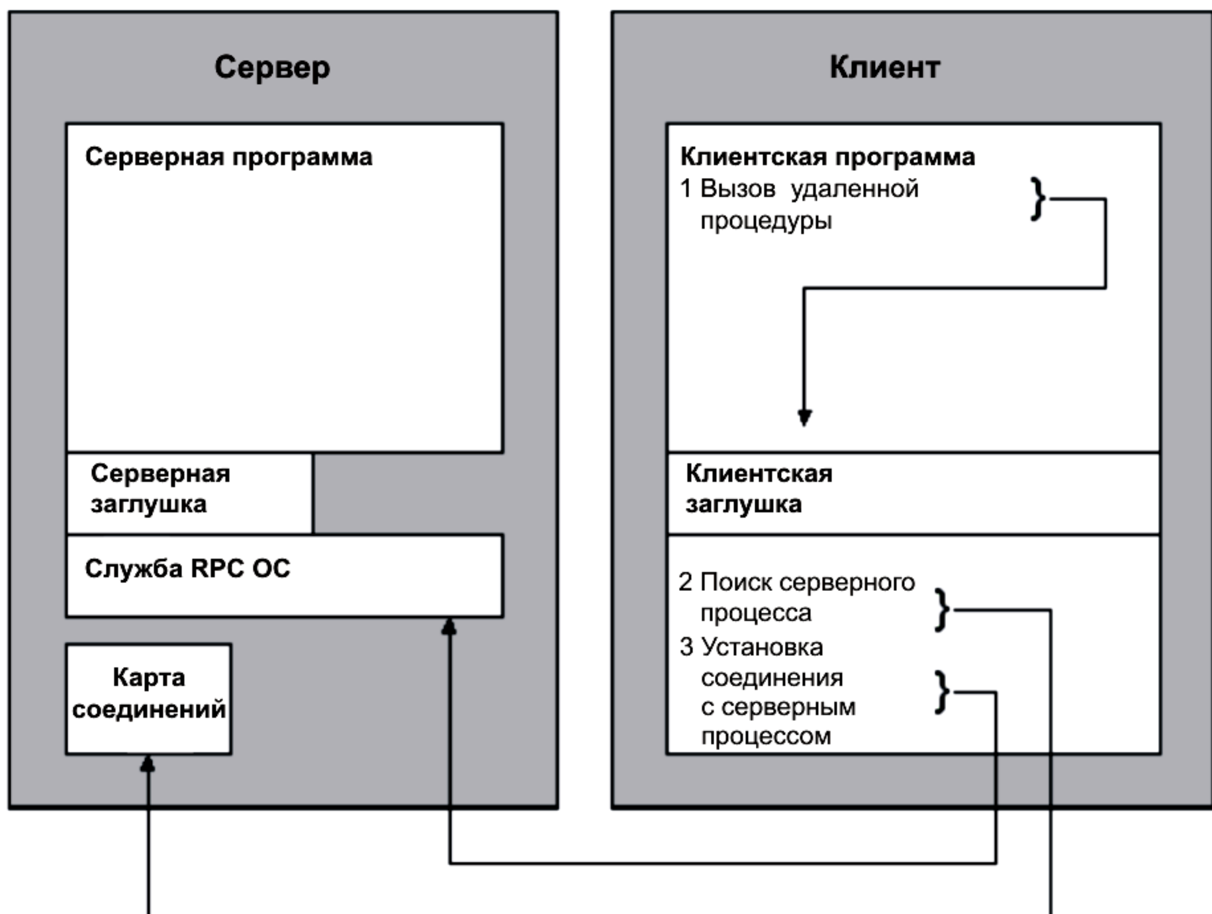

Рис. 3.11 – Подготовка клиентской программы для вызова удаленной процедуры

3.3.2.4 Создание дескриптора соединения

Для создания дескриптора соединения необходимо указать UUID, протокол, адрес сервера.

Например,

```
RPC_STATUS status;
unsigned short *StringBinding;
RPC_BINDING_HANDLE BindingHandle;
```

```
status = RpcStringBindingCompose(NULL,  // Object UUID
             L"ncacn_ip_tcp",       // Protocol sequence to use
             L"MyServer.MyCompany.com",
// Server DNS or Netbios Name
             NULL,
             NULL,
             &StringBinding);
```

Функция RpcStringBindingCompose, подобно printf, создает строку и соединяет всю информацию в заданном формате.

Затем вызывается функция RpcBindingFromStringBinding, с помощью которой создается дескриптор соединения. Например,

```
status = RpcBindingFromStringBinding(StringBinding,
&BindingHandle);
```

Затем строка, созданная функцией RpcStringBindingCompose, освобождается с помощью вызова функции RpcStringFree. Например,

```
RpcStringFree(&StringBinding);
```

На этом подготовительный этап завершается.

3.3.2.5 Вызов удаленной процедуры

Клиентская программа может управлять объемом служебной информации, передаваемой на сервер. Это достигается с помощью указания типа дескриптора соединения (Binding Handle). Различают следующие типы: automatic, implicit и explicit (рис. 3.12).

Тип «automatic» означает, что клиенту и серверу нет никакой нужды обмениваться служебной информацией о соединении. В этом случае дескрипторы соединения не создаются и происходит непосредственный вызов удаленной процедуры.

Тип «implicit» (неявный) означает, что клиент сам создает параметры соединения и дескриптор. Служба RPC клиента только управляет дескриптором. Сервер при этом работает также, как и при автоматическом дескрипторе.

Рис. 3.12 – Типы дескрипторов

Тип «explicit» (явный) означает, что клиент создает параметры соединения и дескриптор. Служба RPC клиента передает созданный дескриптор на сервер, который управляет данным дескриптором.

3.3.2.6 Нахождение серверной программы

После установки параметров и дескриптора соединения, производится вызов удаленной процедуры. В клиентской программе вызывается заглушка, в которой первым шагом производится поиск серверного приложения, что осуществляется на основе информации, связанной с дескриптором соединения. Затем этот запрос попадает в службу RPC серверной программы, которая ищет серверное приложение, просматривая таблицу соединений. Если такое соединение находится в таблице, то создается серверный дескриптор соединения, на основе которого вызывается удаленная процедура найденного серверного приложения.

3.3.2.7 Передача параметров от клиентского приложения серверному

После того как серверное приложение найдено и создан дескриптор серверного соединения, производится преобразование и передача параметров от клиентской заглушки серверной. Серверная заглушка принимает параметры, преобразует их передает в удаленную процедуру серверного приложения. После выполнения процедуры производится конвертация и передача выходных параметров в клиентское приложение. Передача параметров от клиентского приложения серверному и обратно называется маршаллиногом.

Таблица 3.1

Имя протокола	Описание
ncacn_ip_tcp	TCP над IP
ncacn_nb_tcp	TCP над NetBIOS
ncacn_nb_ipx	IPX над NetBEUI
ncacn_nb_nb	NetBIOS над NetBEUI
ncacn_np	Именованные каналы
ncacn_spx	SPX
ncacn_dnet_nsp	DECnet
ncadg_ip_udp	UDP над IP
ncadg_ipx	IPX
ncalrpc	Локальный вызов процедуры

3.3.3 Пример создания сетевого приложения на основе RPC Windows

В этом примере по шагам расписано создание сетевого приложения на основе модели «клиент/сервер». Перечислим эти шаги:

1. Создается файл описания интерфейса (IDL) и файл конфигурации приложения (ACF).
2. По средством использования компилятора MIDL, на основе файлов IDL и AFC, создается заголовочный файл и функции заглушки для клиентского и серверного приложений.
3. Создается проект клиентского приложения.
4. Создается проект серверного приложения.
5. Создаются exe программы для клиента и сервера.

Рассмотрим пример программы, печатающей "Hello, Word". Представим эту программу из main функции и функции HelloProc, которая печатает строку (эти функции разделены на разные файлы).

```
/* file hellop.c */
#include <stdio.h>
void HelloProc(unsigned char * pszString)
{
    printf("%s\n", pszString);
}

/* file: hello.c, a stand-alone application */
#include "hellop.c"
void main(void)
{
    unsigned Char * pszString = "Hello, World";
    HelloProc(pszString);
}
```

Проблем никаких нет, если это все на одном компьютере. Но если main находится на одном компьютере, а HelloProc на другом, возникает

необходимость использования механизма RPC. Далее рассмотрим пример построения механизма RPC для данного случая.

3.3.3.1 Определение интерфейса

Определение интерфейса – это формальное описание того, как клиентские и серверные приложения обмениваются информацией между собой в процессе работы. Интерфейс описывает, как вызвать нужную удаленную процедуру, какие параметры следует принять от клиента, а какие параметры передать клиенту. Кроме того, интерфейс описывает типы передаваемых данных и способы передачи данных от клиента серверу и наоборот.

Описание интерфейса осуществляется с помощью языка MIDL (the Microsoft Interface Definition Language), который содержит Си-подобный синтаксис. Интерфейс должен быть записан в специальном файле с расширением IDL. Кроме того, необходимо записать файл конфигурации с расширением ACF.

3.3.3.2 Генерация UUID

Первым шагом создания интерфейса необходимо сгенерировать уникальный идентификатор интерфейса UUID (universally unique identifier). Генерация UUID осуществляется с помощью специальной программы **uuidgen.**

Следующая команда генерирует UUID и записывает его в файл Hello.idl:

```
C:\>uuidgen /i /ohello.idl
```

Файл hello.idl будет выглядеть приблизительно похожим на следующий:

```
[
   uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
   version(1.0)
]
interface INTERFACENAME
{

}
```

3.3.3.3 IDL файл

IDL файл содержит одно или более описаний интерфейсов. Каждое описание интерфейса содержит заголовок и тело. Заголовок заключенный в квадратные скобки содержит UUID интерфейса и некоторую дополнительную информацию. Тело содержит описание прототипов функций в

стиле языка Си, описание типов аргументов, и процесса передачи их по сети. Рассмотрим описание интерфейса для RPC функции HelloProc.

```
//file hello.idl
[
    uuid(7a98c250-6808-11cf-b73b-00aa00b677a7),
    version(1.0)
]
interface hello
{
    void HelloProc([in, string] unsigned char * pszString);
    void Shutdown(void);
}
```

Заголовок содержит UUID и версию функции. Тело содержит описание прототипа функции HelloProc и ShutDown. Ключевое слово [in] означает то что, значение этого параметра передается от клиента к серверу. Ключевое слово [string] означает что, значением этого параметра является строка символов.

Чтобы клиент мог закрыть серверное приложение, интерфейс содержит прототип функции Shutdown**.**

3.3.3.4 Файл конфигурации

Файл конфигурации (ACF) задает некоторые параметры взаимодействия между клиентом и сервером. Например, если клиентское приложение содержит сложные структуры данных, зависящие от локального компьютера. Тогда необходимо их преобразовать в формат хранения данных независящий от конкретного компьютера и, соответственно, указать их спецификации в ACF файле. Например, необходимо задать тип дескриптора, который имеет три значения: auto_handle, implicit_handle, explicit_handle. В нашем примере используется дескриптор типа implicit_handle.

```
//file: hello.acf
[
    implicit_handle (handle_t hello_IfHandle)
]
interface hello
{
}
```

3.3.3.5 Генерация файла заглушки

Задав определение интерфейса, необходимо разработать тесты программ сервера и клиента. Для этого необходимо записать простой файл (makefile) для генерации заглушки (stub) и заголовочных файлов (header files). Используя MIDL компилятор, можно получить заглушки и заголовочные файлы. Например, записав команду

C:\> midl hello.idl

получим файлы hello.h, Hello_c.c файл, содержащий клиентскую заглушку и Hello_s.c – файл содержащий серверную заглушку.

Файл Hello.h содержит прототипы функций HelloProc и Shutdown, две функции управления памятью, midl_user_allocate и midl_user_free, которые необходимы для передачи строки символов от клиента серверу.

Поскольку в ACF файле задан атрибут [implicit_handle**]**, то будут сгенерированы дескрипторы *hello_IfHandle*, клиентского приложения *hello_v1_0_c_ifspec* и для серверного приложения *hello_v1_0_s_ifspec*. Клиентские и серверное приложения будут использовать эти дескрипторы.

```
/*file: hello.h */
/* this ALWAYS GENERATED file contains the definitions for the
interfaces */
/* File created by MIDL compiler version 3.00.06
/* at Tue Feb 20 11:33:32 1996 */
/* установки ключей для компилятора hello.idl:
    Os, W1, Zp8, env=Win32, ms_ext, c_ext
    error checks: none */
//@@MIDL_FILE_HEADING(  )
#include "rpc.h"
#include "rpcndr.h"

#ifndef __hello_h_
#define __hello_h_

#ifdef __cplusplus
extern "C"{
#endif

/* Forward Declarations */

void __RPC_FAR * __RPC_USER MIDL_user_allocate(size_t);
void __RPC_USER MIDL_user_free( void __RPC_FAR * );

#ifndef __hello_INTERFACE_DEFINED__
#define __hello_INTERFACE_DEFINED__

            /* size is 0 */
void HelloProc(
    /* [string][in] */ unsigned char __RPC_FAR *pszString);
    /* size is 0 */
void Shutdown( void);
extern handle_t hello_IfHandle;

extern RPC_IF_HANDLE hello_v1_0_c_ifspec;
extern RPC_IF_HANDLE hello_v1_0_s_ifspec;
#endif /* __hello_INTERFACE_DEFINED__ */
```

```
/* Additional Prototypes for ALL interfaces */
/* end of Additional Prototypes */
#ifdef __cplusplus
}
#endif
#endif
```

3.3.3.6 Клиентское приложение

Файл клиентского приложения Helloc.c должен включать заголовочный файл Hello.h, в котором содержится описание дескрипторов и интерфейса, сгенерированного MIDL.

В исходном тексте клиентского приложения должны быть вызваны специальные функции службы RPC, которые ищут и устанавливают связь с серверным приложением, вызывают заданную удаленную функцию, принимают значения выходных параметров, завершают сеанс связи.

Функция RpcStringBindingCompose заносит различные параметры связи в одну строку. В этом примере в качестве связи с сервером используется механизм, основанный на каналах (pipes), сеть локальная ("ncacn_np"). Функция RpcBindingFromStringBinding создает дескриптор соединения для передачи серверу, *hello_IfHandle*. Затем производится вызов удаленной процедуры.

Вызывается удаленная процедура HelloProc, затем через механизм заглушек – Shutdown. Если в процессе обработки удаленного вызова обнаруживается ошибка, то будет сгенерировано исключение RpcExcept.

После вызова удаленных процедур клиентское приложение должно освободить память, выделенную для строки соединения, вызвав функцию RpcStringFree. Затем клиент должен завершить работу RPC службы и освободить дескриптор связи, вызвав функцию RpcBindingFree. Полный текст клиентского приложения записан ниже.

```
/* file: helloc.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"

void main()
{
    RPC_STATUS status;
    unsigned char * pszUuid             = NULL;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszNetworkAddress   = NULL;
    unsigned char * pszEndpoint    = "\\pipe\\hello";
    unsigned char * pszOptions          = NULL;
    unsigned char * pszStringBinding    = NULL;
    unsigned char * pszString      = "hello, world";
```

```
    unsigned long ulCode;
 //создаем строку связи
    status = RpcStringBindingCompose(pszUuid,
                                    pszProtocolSequence,
                                    pszNetworkAddress,
                                    pszEndpoint,
                                    pszOptions,
                                    &pszStringBinding);
    if (status)
    {
        exit(status);
    }
//создаем дескриптор
    status = RpcBindingFromStringBinding(pszStringBinding,
                                        &hello_IfHandle);

    if (status)
    {
        exit(status);
    }

 //вызываем удаленные функции
    RpcTryExcept
    {
        HelloProc(pszString);
        Shutdown();
    }
    RpcExcept(1)  //если есть ошибка, обрабатываем ее
    {
        ulCode = RpcExceptionCode();
        printf("Runtime reported exception 0x%lx = %ld\n",
ulCode, ulCode);
    }
    RpcEndExcept

//освобождаем память, занятую строкой связи
    status = RpcStringFree(&pszStringBinding);

    if (status)
    {
        exit(status);
    }
 //освобождаем  дескриптор соединения
    status = RpcBindingFree(&hello_IfHandle);

    if (status)
    {
        exit(status);
    }

    exit(0);
```

```
}  // end main()
```

3.3.3.7 Серверное приложение

На серверной стороне распределенного приложения система информирует о том, что соответствующие сервисы имеются в наличии и находятся в режиме ожидания клиентских запросов. Рассмотрим пример организации серверного приложения

Файл Hellos.c содержит главную процедуру сервера. Файл Hellop.c содержит тексты удаленных процедур (смотри описание файла выше).

Серверное приложение использует три функции службы RPC: RpcServerUseProtseqEp, RpcServerRegisterIf, RpcServerListen. Первые две обеспечивают службу RPC о способах организации взаимодействия клиентов и удаленных процедур. Функция RpcServerUseProtseqEp обеспечивает:

1) описание протокола;
2) описание адреса сервера (endpoint);
3) параметры ожидания и запросов.

Функция RpcServerRegisterIf регистрирует интерфейс в службе RPC. В нашем примере интерфейс записан в файле hello.h в структуре hello_v1_0_s_ifspec.

После регистрации вызывается функция RpcServerListen, которая ожидает запросов клиентских приложений. Кроме того серверное приложение должно иметь две вспомогательные функции для распределения памяти, используемой в серверной заглушке: midl_user_allocate и midl_user_free. Текст сервера и соответствующих функций приведен ниже.

```
/* file: hellos.c */
#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "hello.h"

void main()
{
    RPC_STATUS status;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity      = NULL;
    unsigned char * pszEndpoint     = "\\pipe\\hello";
    unsigned int    cMinCalls       = 1;
    unsigned int    cMaxCalls       = 20;
    unsigned int    fDontWait       = FALSE;

    status = RpcServerUseProtseqEp(pszProtocolSequence,
                                   cMaxCalls,
                                   pszEndpoint,
                                   pszSecurity);
```

```
    if (status)
    {
        exit(status);
    }

    status = RpcServerRegisterIf(hello_v1_0_s_ifspec,
                                 NULL,
                                 NULL);

    if (status)
    {
        exit(status);
    }

    status = RpcServerListen(cMinCalls,
                             cMaxCalls,
                             fDontWait);

    if (status)
    {
        exit(status);
    }

 }  // end main()

/*******************************************************/
/*          MIDL allocate and free                     */
/*******************************************************/

void __RPC_FAR * __RPC_USER midl_user_allocate(size_t len)
{
    return(malloc(len));
}

void __RPC_USER midl_user_free(void __RPC_FAR * ptr)
{
    free(ptr);
}
```

3.3.3.8 Завершение работы сервера

Для того, чтобы сервер завершил свою работу, необходимо вызвать функции RpcMgmtStopServerListening и RpcServerUnregisterIf или просто вызвать exit() в функции сервера main. В первом случае необходимо выполнить следующие шаги: сгенерировать исключение с помощью вызова функции RpcMgmtStopServerListening; затем необходимо отменить регистрацию интерфейса, вызвав функцию RpcServerUnregisterIf.

Для завершения работы сервера со стороны клиента используется функция ShutDown, которая вызывает описанные выше функции.

void Shutdown(void)

```
{
    RPC_STATUS status;

    status = RpcMgmtStopServerListening(NULL);

    if (status)
    {
       exit(status);
    }

    status = RpcServerUnregisterIf(NULL, NULL, FALSE);

    if (status)
    {
       exit(status);
    }
} //end Shutdown
```

4 МНОГОПОТОЧНЫЕ ПРИЛОЖЕНИЯ

4.1 Процессы

Вычислительный процесс – абстракция операционной системы, предназначенная для обозначения выполняемой программы в памяти компьютера. Поэтому детально данное понятие рассматривается в курсах, посвященных операционным системам. Ниже будет кратко изложено понятие «процесс» с точки зрения создания сетевых приложений.

Поскольку *сетевые приложения* – это совокупность программ, взаимодействующих в компьютерной сети на основе клиент/серверной модели, логично рассматривать клиентские и серверные процессы. Однако для операционной системы нет различия между клиентским и серверными процессами, поэтому данные процессы будут рассматриваться совместно.

Понятие «процесс» зависит от операционной системы. Так ОС UNIX и ОС Microsoft Windows трактуют процессы по-разному. Первая трактует процесс как копию программы, исполняемая на компьютере. Вторая – трактует процесс, как некоторый объект со своим адресным пространством. Непосредственно выполнением команд программы осуществляет поток (thread). Поэтому в MS Windows вычислительный процесс объединяет понятие «процесс» и «поток». Процесс может иметь несколько потоков, тогда говорят о параллельном выполнении программы.

Рассмотрим подробнее процессы и потоки в MS Windows. Процесс определяется как копия исполняемой программы, находящейся в оперативной памяти. Процесс инертен и задает адресное пространство исполняемой программы. Кроме того, процесс является объектом ядра ОС и, соответственно, имеет свое описание, в структурах ОС. Идентификации этого описания осуществляется с помощью дескриптора (HANDLE).

Создания механизма собственного адресного пространства для каждого процесса основано на использовании виртуальной памяти и позволяет минимизировать влияние процессов друг на друга при выполнении нескольких программ одновременно.

Для выполнения команд загруженной программы для данного процесса создается один или несколько потоков. Таким образом, *поток* – абстракция ОС Windows, предназначенная для исполнения программы данного процесса. Поскольку процессов может быть несколько, то потоки всех процессов ставятся в одну очередь и каждому потоку дается квант времени процессора, в которое он выполняет фрагмент программы. Этот механизм называется разделением времени процессора. Если процесс отвечает за распределение исполняемой программы в памяти компьютера, то поток отвечает за выполнение фрагмента программы в выделенный квант времени процессора. Таким образом, хотя процесс и потоки являются независимыми объектами ядра ОС, они неразрывно связаны друг с другом.

Процесс в ОС Windows создается в следующих случаях:

1. При запуске некоторой программы с помощью стандартных средств ОС.

2. Запуск процесса с помощью функции CreateProcess().

Рассмотрим подробнее запуск программы с помощью функции CreateProcess().

Ниже записан прототип функции CreateProcess:

```
BOOL CreateProcess(
PCTSTR ApplicationName,
PTSTR CommandLine,
PSFCURITY_ATTRIBUTES Process,
PSECURITY_ATTRIBUTES Thread,
BOOL InheritHandles,
DWORD Create,
PVOID Environment,
PCTSTR CurDir,
PSTARTUPINFO StartInfo,
PPROCESS_INFORMATION ProcInfo
);
```

При вызове этой функции операционная система создает виртуальное пространство нового процесса: записывает туда исполняемый код, заносит статические библиотеки кода (DLL), грузит ресурсы, создает: переменные окружения, структуру описания процесса, дескриптор процесса, первичный поток выполнения.

Параметры этой функции следующие:

ApplicationName – имя exe программы;

CommandLine – командная строка;

Process – атрибуты защиты процесса;

Thread – атрибуты защиты потока;

InheritHandles – флаг наследования;

Create – задает флаги, определяющие создание процесса;

Environment – указывает на блок памяти, хранящий строки переменных окружения;

CurDir – устанавливает текущие диск и каталог для создаваемого процесса;

StartInfo – этот параметр указывает на структуру STARTUPINFO:

```
typedef struct _STARTUPINFO {
DWORD cb;
PSTH lpReserved;
PSTR lpDesktop;
PSTR lpTitle;
DWORD dwX;
DWORD dwY;
DWORD dwXSize;
DWORD dwYSize;
DWORD dwXCountChars;
DWORD dwYCountChars;
```

```
DWORD dwFillAttribute;
DWORD dwFlags;
WORD wShowWindOw;
WORD cbReserved2;
PBYTE lpReserved2;
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

ProcInfo – этот параметр указывает на структуру PROCESS_INFORMATION;

```
typedef struct _PROCESS_INFORMATION {
HANDLE hProcess;
HANDLE hThread;
DWORD dwProcessId;
DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Пример запуска exe-программы NOTEPAD

```
//организуем структуры  STARTUPINFO и PROCESS_INFORMATION
//записываем командную строку
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR CommandLine[] = TEXT("NOTEPAD");
//создаем процесс и первичный поток
CreateProcess(NULL, CommandLine, NULL, NULL, FALSE, 0, NULL,
NULL, &si, &pi);
```

Все дескрипторы процесса и первичного потока записаны в структуре *pi*.

Пример запуска дочернего процесса

```
void main( VOID )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);

    // Start the child process.
    if( !CreateProcess( NULL, // No module name (use command
line).
        "MyChildProcess", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
```

```
        NULL,       // Use parent's environment block.
        NULL,       // Use parent's starting directory.
        &si,        // Pointer to STARTUPINFO structure.
        &pi )       // Pointer to PROCESS_INFORMATION structure.
    )
        ErrorExit( "CreateProcess failed." );

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );

    CloseHandle( pi.hThread );
}
```

4.2 Потоки (Thread)

Поток – это объект ядра ОС MS Windows, предназначенный для выполнения команд некоторой программы и функции. В понимании потоков важным является концепция разделения времени. Разделение времени – это последовательное выполнение нескольких программ на одном процессоре. Разделение времени подразумевает, что на выполнение каждой программы выделяется некоторый квант времени и имеется механизм переключения от выполнения одной программы к другой. Создается иллюзия, что процессор параллельно выполняет несколько программ. Механизм переключения основан на возможности сохранения и восстановления всех значений регистров процессора в некоторой памяти.

Таким образом, создавая многопоточное приложение на одном быстром процессоре, можно обрабатывать запросы многих клиентов. Механизм многопоточного приложения идеально подходит для создания серверного приложения. С другой стороны, многопоточное приложение можно выполнять на нескольких процессорах одновременно. Это позволяет масштабировать серверное приложение. В случае, когда клиентов не так много используется однопроцессорный компьютер, для большого числа клиентов – многопроцессорный. Механизм работы многопоточного серверного приложения следующий:

1. Имеется первичный поток, ожидающий запросов от клиентов.
2. При появлении запроса создается новый поток или используется некоторый свободный поток, который и обрабатывает далее запрос от клиентского приложения.
3. Закончив обработку, первичный поток ждет нового запроса.

Рассмотрим подробнее создание и использование потоков в ОС Windows. Первичный поток приложения всегда создается операционной системой вместе с созданием процесса. Вторичные или дополнительные потоки создаются с помощью функции CreateThread. Прототип функции представлен ниже:

```
HANDLE CreateThread(
PSECURITY_ATTRIBUTES psa,
DWORD Stack,
PTHREAD_START_ROUTINE StartAddr,
PVOID Param,
DWORD Create,
PDWORD ThreadID);
```

Параметры следующие:

- psa – атрибуты защиты потока;
- Stack – размер стека;
- StartAddr – адрес функции потока;
- Param – параметр, передаваемый в функцию потока;
- Create – флаг запуска потока;
- ThreadId – идентификатор потока.

Функция CreateThread возвращает значение дескриптора потока. Если это значение NULL, то поток не был создан. Код ошибки можно узнать, запросив его с помощью функции GetLastError().

Ниже представлен простой пример нового потока. Функция ThreadFunc – функция потока, которая принимает параметр lpParam, содержащий указатель на целую переменную. Далее эта функция преобразует число в строку с помощью функции wsprintf и затем вызывается ящик сообщений для выдачи этого числа на экране. Поток завершится после нажатия клавиши «ОК» на ящике сообщений.

```
DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
    char szMsg[80];

    wsprintf( szMsg, "ThreadFunc: Parameter = %d\n", *lpParam
);
    MessageBox( NULL, szMsg, "Дополнительный поток.", MB_OK );

    return 0;
}
```

При запуске программы создается процесс и первичный поток, полная информация хранится в StartUpinfo.

```
VOID main( VOID )
{
    DWORD dwThreadId, dwThrdParam = 1;
    HANDLE hThread;
    //запускаем еще поток
    hThread = CreateThread(
        NULL,                        //атрибуты защиты отсутствуют
        0,                           // размер стека по умолчанию
        ThreadFunc,                  // функция потока
```

```
        &dwThrdParam,      // значение параметра функции потока
        0,                 // флаг запуска по умолчанию
        &dwThreadId);      // возвращает идентификатор потока

   //Проверка на ошибку  и ее обработка

   if (hThread == NULL)
      ErrorExit( "CreateThread failed." );
//освободить дескриптор потока
MessageBox( NULL, "Первичный поток", "Работа.", MB_OK );
   CloseHandle( hThread );
}
```

Поскольку потоки обрабатываются параллельно, то на экране появится два ящика сообщений – первичного потока и дополнительного потока.

4.3 Синхронизация потоков

Рассмотрим следующую задачу: имеется два потока и некоторая переменная x, доступ к которой имеется у обоих потоков. Первый поток выполняет оператор x=x+10. Второй – y=x+5. Перед выполнением потоков значения x=10 y=5. Ниже в таблице 3.2 представлены команды, которые выполняет процессор для реализации соответствующих операторов потоков.

Таблица 3.2 – Команды

Поток 1 (x=x+10)	Поток 2 (y=x+5)
Прочитать x в регистр RX	Прочитать x в регистр RX
Прибавить 10 в RX	Прибавить 5 в RX
Запомнить результат RX в x	Запомнить результат RX в y

Необходимо напомнить, что при переключении между потоками, контент активного потока запоминается, а контент следующего – восстанавливается.

Тогда рассмотрим два возможных сценария работы процессора.

Сценарий 1.

Поток 1:Прочитать x в регистр RX

Поток 2: Прочитать x в регистр RX

Поток 1: Прибавить 10 в RX

Поток 2: Прибавить 5 в RX

Поток 1: Запомнить результат RX в x

Поток 2: Запомнить результат RX в y

Выполнение этого сценария приведет к следующему результату x=20, y=15.

Сценарий 2.

Поток 1:Прочитать x в регистр RX

Поток 1: Прибавить 10 в RX
Поток 1: Запомнить результат RX в x
Поток 2: Прочитать x в регистр RX
Поток 2: Прибавить 5 в RX
Поток 2: Запомнить результат RX в y

В результате выполнения сценария 2 будет получим x=20, y=25.

Как видно, результаты не совпадают. В первом случае y=15, во втором случае y=25. Это говорит о том, что возник конфликт доступа к переменной *x*. Для того, чтобы не возникало таких конфликтов, необходимо синхронизировать доступ потоков к общим переменным.

Синхронизация потоков в Windows может быть реализована в пользовательском и системном режимах. В пользовательском режиме не происходит переход в режим работы ядра, тем самым быстродействие повышается. Однако этот режим обеспечивает сравнительно простые схемы синхронизации. В режиме работы ядра Windows позволяет создавать разнообразные схемы синхронизации потоков.

4.4 Атомарный доступ

Монопольный режим доступа к переменным можно организовать с помощью специальных функций Windows: InterlockedCompareExchange, InterlockedExchangeAdd, InterlockedDecrement, InterlockedExchange, and InterlockedIncrement

Ниже приведен прототип функции InterlockedExchange, которая в монопольном режиме изменяет значение переменной.

```
LONG InterlockedExchange(
    LPLONG Target,
  // адрес 32 разрядной переменной для изменения значения
    LONG Value      // новое значение
   );
```

Данная функция возвращает предыдущее значение переменной.

Функция InterlockedExchangeAdd() монопольно добавляет значение Increment, по адресу, записанному в первом параметре. Прототип функции следующий:

```
LONG InterlockedExchangeAdd (
    PLONG Addend,   //адрес переменной
    LONG Increment  // значение, которое надо прибавить
   );
```

Функция InterlockedExchangeAdd() возвращает предыдущее значение переменной, адрес которой записан в первом параметре.

Пример. Необходимо организовать нумерацию потоков таким образом, чтобы при входе в поток был получен некоторый номер. Предполо-

жим, что номер хранится в глобальной переменной Count. Тогда фрагмент кода, обеспечивающий правильное выполнение, будет следующим:

```
int Count;
DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
…
 num=InterlockedIncrement(&Count);
…
}
```

4.5 Критические секции

Критическая секция (critical section) – это небольшой участок программного кода, который требует монопольного доступа к общей памяти. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу.

Рассмотрим пример. Пусть имеется простейший линейный односвязный список.

```
typedef struct {
   LIST *next;
int value;
} LIST;

LIST *list;
```

Необходимо создать многопоточное приложение, в котором некоторые потоки могли бы вставлять данные в этот список list.

Текст функции добавления в список следующий:

```
void AddList(int value){
//создаем элемент списка и инициализируем поля
LIST *item=CreateItem(value);
if( list!=NULL) { //если список уже есть
  item->next=list; //вставляем новый элемент
}
 list=item; //указатель списка устанавливаем на вставленный
}
```

Если несколько потоков попытаются одновременно добавить в список элемент, то произойдет коллизия. Кроме того, функции *Intrloked,* рассмотренные выше, здесь также не помогут, поскольку требуется атомарный доступ не к одной переменной, а к нескольким. Поэтому здесь можно воспользоваться критическими секциями. Для этого необходимо создать критическую секцию, в нее войти и затем выйти. Эти действия производятся с помощью функций: *InitializeCriticalSection, EnterCriticalSection, LeaveCriticalSection.*

Тогда для синхронизации потоков, которые используют функцию добавления AddList, необходимо записать следующий код:

```
CRITICAL_SECTION g_cs;

DWORD WINAPI ThreadFunc(PVOID pvParam)
{
…
 EnterCriticalSection(&g_cs);  //войти в критическую секцию
 AddList(value);
LeaveCriticalSection(&g_cs); //выйти из критической секции
...
}
```

В этом случае, критическая секция защищает доступ к добавлению элемента в список, пока не будет выполнена функция *LeaveCriticalSection.* Это означает, что все остальные потоки, которым необходимо вставить элемент в список, будут ждать пока, в данный поток не выполнит функцию *LeaveCriticalSection.*

4.6 Синхронизация потоков в системном режиме

Синхронизация потоков в системном режиме осуществляется с помощью объектов ядра и специальных функций WaitForSingleObject, WaitForMultipleObjects.

Объектами ядра являются:

- процессы (Process);
- потоки (Thread);
- события (Event);
- ожидаемые таймеры (Waitable Timer);
- семафоры (Semaphore);
- мьютексы (Mutex).

Все объекты ядра могут быть в двух состояниях – свободном и занятом. Правила, по которым объект находится в занятом или свободном состоянии, зависят от самого объекта. Например, поток находится в занятом состоянии, пока он не завершится. Потоки можно приостановить, пока некоторые объекты находятся в занятом состоянии. Для этого используются функции WaitForSingleObject, WaitForMultipleObjects.

Функция WaitForSingleObject приостанавливает поток, если некоторый объект занят, или не будет исчерпано время ожидания. Прототип функции следующий:

```
DWORD WaitForSingleObject(
 HANDLE hObject, //дескриптор объекта
 DWORD dwMilliseconds //время ожидания
);
```

Данная функция возвращает целое значение:

WAIT_OBJECT_0 – объект освободился;

WAIT_TIMEOUT – завершилось время ожидания;

WAIT_FAILED – ошибка, например, неверный дескриптор.

Пример, ожидание завершения процесса, где hProcess – дескриптор процесса.

```
DWORD res = WaitForSingleObject(hProcess, 3000);

switch (res)
{
case WAIT_OBJECT_0:
// процесс завершается
break;

case WAIT_TIMEOUT:
// процесс не завершился в течение 3000 мс
break;

case WAIT_FAILED:
// неправильный вызов функции
break;
}
```

Функция WaitForMultipleObjects ожидает освобождения одного из многих объектов или всех сразу, вариант ожидания зависит от установки специального флага. Прототип функции записан ниже:

```
DWORD WaitForMultipleObjects(
 DWORD dwCount,  //размер массива дескрипторов объектов
 CONST HANDLE* phObjects,//адрес массива дескрипторов объектов
 BOOL fWaitAll, //флаг варианта ожидания
 DWORD dwMilliseconds//время ожидания в миллисекундах
);
```

Ниже представлен пример ожидания завершения одного из трех процессов, представленных дескрипторами.

```
//организуем массив дескрипторов

HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
```

Ожидаем завершения одного из трех (третий параметр FALSE).

```
DWORD res = WaitForMultipleObjects(3, h, FALSE, 3000);
```

```
switch (res)
{

case WAIT_FAILED:
// Ошибка
break;

case WAIT_TIMEOUT:
//ни один из процессов не освободился в течение 3000 мс
break;

case WAIT_OBJECT_0 + 0:
// завершился первый процесс (hprocess1)

break;

case WATT_OBJECT_0 + 1:
// завершился процесс второй процесс (hProcess2)
break;

case WAIT_OBJECT_0 + 2:
// завершился процесс третий процесс(hProcess3)
break;
}
```

Если необходимо ждать окончания всех процессов, то параметру *fWaitAll* нужно присвоить значение TRUE.

4.6.1 События (Events)

Событие (Event) – объект ядра, имеющий два состояния «свободно» и «занято». События могут быть с автосбросом и со сбросом вручную. Событие с автосбросом выводит из режима ожидания один из потоков и автоматически переходит а занятое состояние. Событие со сбросом вручную может активизировать все ожидающие его потоки. Поэтому, чтобы перевести его в занятое состояние, необходимо вызвать специальную функцию.

Объект «событие» создается с помощью функции CreateEvent, прототип которой записан ниже:

```
HANDLE CreateEvent(
PSECURITY_ATTRIBUTES psa,  //атрибуты защиты
BOOL fManualReset,                       //тип события
 BOOL fInitialState,                     //начальное значение
 PCTSTR pszName                          //имя события
);
```

Функция CreateEvent возвращает значение дескриптора события. Если NULL, то, вызвав GetLastError, получим код ошибки.

Если значение параметра fManualReset является TRUE, то событие будет с ручным сбросом, FALSE – с автосбросом.

Параметр *fInitialState* определяет начальное состояние события – TRUE (свободное) или FALSE (занятое).

Для явного перевода события в свободное состояние используется функция SetEvent. Прототип представлен ниже:

```
BOOL SetEvent(HANDLE hEvenet);
```

Для явного перевода события в занятое состояние используется функция ResetEvent. Прототип представлен ниже:

```
BOOL ResetEvent(HANDLE hEvenet);
```

Рассмотрим пример. Пусть имеется четыре потока: один пишет в буфер (Writer), а остальные ждут, потом читают (Reader1, Reader2, Reader3).

```
HANDLE hEvent;

int main( )
{

/*создаем событие со сбросом вручную, в занятом состоянии, без
имени*/
hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

// порождаем три новых потока

HANDLE hThread[3];

DWORD ID;

hThread[0] = _beginthreadex(NULL, 0, Reader1, NULL, 0, &ID);
hThread[1] = _beginthreadex(NULL, 0, Reader2, NULL, 0, &ID);
hTbread[2] = _beginthreadex(NULL, 0, Reader3, NULL, 0, &ID);

Writer( );

// разрешаем всем потокам обращаться буферу
SetEvent(hEvent);

//ждем завершения всех  потоков

DWORD res = WaitForMultipleObjects(3,hThread, TRUE, INFINITE);

CloseHandle(hEvent);

for(int i=0; i<3; i++) CloseHandle(hThread[i]);
```

```
return 0;

}

DWORD WINAPI Reader1(PVOID pvParam)
{
// ждем, когда в буфер будет написано
WaitForSingleObject(hEvent, INFINITE);
// обращаемся буферу
return(0);
}

DWORD WINAPI Reader2(PVOID pvParam)
{
// ждем, когда в буфер будет написано
WaitForSingleObject(hEvent, INFINITE);
// обращаемся буферу
return(0);
}

DWORD WINAPI Reader3(PVOID pvParam)
{
// ждем, когда в буфер будет написано
WaitForSingleObject(hEvent, INFINITE);
// обращаемся буферу
return(0);
}
```

Эту задачу можно усложнить, поставив условие: писатель многократно пишет в буфер, а читатели многократно читают.

В общем случае, эта задача имеет вид: имеется m писателей, которые пишут в некоторый буфер, затем n читателей должны прочитать. И вся схема работает многократно.

4.6.2 Ожидаемые таймеры

Ожидаемый таймер – это объект ядра, который самостоятельно переходит в свободное состояние через определенное время или регулярные промежутки времени. Для создания ожидаемого таймера необходимо вызвать функцию *CreateWaitableTimer*. Прототип этой функции записан ниже:

```
HANDLE CreateWaitableTimer(
 PSECURITY_ATTRIBUTES psa, //атрибуты защиты
 BOOL fManualReset,        //тип таймера
  PCTSTR pszName);         //имя таймера
```

Параметр *fManualReset* определяет тип ожидаемого таймера: со сбросом вручную (TRUE) или с автосбросом (FALSE). Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков,

ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом – лишь одного из потоков.

Функция CreateWaitableTimer возвращает дескриптор таймера. Если функция возвращает NULL, это сигнализирует об ошибке. Для определения кода ошибки используйте функцию GetLastError.

Таймер всегда создается в занятом состоянии. Чтобы таймер заработал, необходимо вызвать функцию *SetWaitableTimer*.

```
BOOL SetWaitableTimer(
 HANDLE hTimer, //дескриптор таймера
//время срабатывания таймера первый раз
const LARGE_INTEGER *pDueTime,
//период времени, через который будет срабатывать таймер
LONG lPeriod,
//адрес функции
PTIMERAPCROUTINE pfnCompletionRoutine,
//значение параметра для функции
PVOID pvArgToCompletionRoutine,
 BOOL fResume);
```

Интервал задания параметра pDueTime 100 наносекунд в формате FILETIME.

FILETIME структура 64-битное значение с интервалом в 100 наносекунд, начиная с 1 января 1601г. Объявление этой структуры следующее:

```
typedef struct _FILETIME {
    DWORD dwLowDateTime;
    DWORD dwHighDateTime;
} FILETIME;
```

Для преобразования системного времени в формат FILETIME имеется функция:

```
BOOL SystemTimeToFileTime(
// адрес структуры, содержащей системное время
   CONST SYSTEMTIME *lpSystemTime,
//адрес структуры  FILETIME
   LPFILETIME lpFileTime
   );
```

Период задается в миллисекундах.

4.6.3 Семафоры

Семафор объект синхронизации потоков, который позволяет иметь счетчик в заданных границах от 0 до некоторого максимального значения. Счетчик уменьшает значение на единицу всякий раз, когда поток перестает ждать семафор в некоторой функции ожидания.

Семафор находится в занятом состоянии, если его счетчик равен нулю и в свободном состоянии если его значение больше нуля.

Семафор – удобный инструмент учета разделяемых между потоками ресурсов. Предположим, что у нас имеется 5 потоков, которые могут использовать 3 буфера. Семафор позволяет организовать надежный доступ 5 потоков к 3 буферам. Каждый поток ожидает семафор, если счетчик семафора более нуля, то один из ожидающих потоков становится активным, при этом счетчик уменьшается на единицу. Как только счетчик равное нулю, все буферы разобраны. То остальные потоки будут ждать. Увеличение счетчика семафора осуществляется специальной функцией ReleaseSemaphore. В нашем случае это должен делать поток при освобождении одного из буферов.

Для создания семафора используется функция CreateSemaphore, прототип который записан ниже:

```
HANDLE CreateSemaphore(

// атрибуты защиты
   LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
    LONG lInitialCount,  // начальное значение счетчика
    LONG lMaximumCount,  // максимальное значение счетчика
    LPCTSTR lpName  // имя семафора
   );
```

Функция CreateSemaphore возвращает дескриптор семафора. Если функция возвращает NULL, это сигнализирует об ошибке. Для определения кода ошибки используйте функцию GetLastError().

Функция ReleaseSemaphore увеличивает счетчик на указанную величину и записывает предыдущее значение по заданному адресу. Прототип функции записан ниже.

BOOL ReleaseSemaphore(

```
  HANDLE hSemaphore,     // дескриптор семафора
// число, которое надо прибавить к счетчику семафора
  LONG lReleaseCount,
// адрес, куда будет записано предыдущее значение счетчика
  LPLONG lpPreviousCount
   );
```

4.6.4 Мьютексы

Мьютекс – объект синхронизации Windows, который позволяет потоку владеть разделяемым ресурсом монопольно. Мьютекс подобен критической секции, однако работает в режиме ядра. Это позволяет синхронизировать потоки для разных процессов, задавать максимальное время ожидания потока. Мьютекс создается функцией CreateMutex, прототип этой функции записан ниже:

```
HANDLE CreateMutex(
```

```
    LPSECURITY_ATTRIBUTES lpMutexAttributes, // атрибуты защиты
    BOOL bInitialOwner,  // начальный флаг (свободен, занят)
    LPCTSTR lpName  // имя мьютекса
   );
```

Функция CreateMutex возвращает дескриптор мьютекса. Если функция возвращает значение NULL, это сигнализирует об ошибке. Для определения кода ошибки используют функцию GetLastError().

Для освобождения мьютекса владеющий поток должен вызвать функцию ReleaseMutex. Прототип записан ниже:

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

Примеры использования мьютекса:

1. Мьютекс надо использовать, если в один буфер или список хотят записать информацию несколько потоков.
2. С помощью мьютекса можно запретить запускать новые копии программы, которая уже есть в памяти компьютера.

4.7 Пулы потоков

Решение проблемы создания, управления и уничтожения некоторого множества потоков зависит от конкретной задачи и носит довольно сложный характер. Однако для определенных сценариев уже построены специальные функции, которые обеспечивают управление пулом потоков. Можно перечислить следующие сценарии:

1) очередь асинхронных вызовов функций
2) использование порта завершения ввода/вывода.

4.7.1 Очередь асинхронных вызов функций

Типичный сценарий серверного приложения заключается в следующем: первичный поток ждет запроса клиентских приложений. При получении запроса первичный поток записывает его в очередь запросов на обработку, и, затем, снова переходит в режим ожидания. Как только в очереди появились запросы, начинают работать потоки, обеспечивающие обработку запросов клиентов. При этом запрос из очереди удаляется.

Такой сценарий можно организовать с помощью функции QueueUserWorkItem, прототип которой записан ниже:

```
BOOL QueueUserWorkItem(
 PTHREAD_START_ROUTINE pfnCallback, //функция потока
 PVOID pvContext,       //параметр
ULONG dwFlags   //флаги, задающие варианты сценария
);
```

Эта функция помещает «рабочий элемент" (work item) в очередь пула потоков и тут же возвращает управление. Рабочий элемент – это просто функция (на которую ссылается параметр *pfnCallback),* принимающая единственный параметр *pvContext.* У рабочего элемента должен быть следующий прототип:

```
DWORD WINAPI WorkItemFunc(PVOID pvContext);
```

4.7.2 Использование порта завершения ввода/вывода

Для создания пула потоков на основе порта завершения ввода/вывода по сценарию обработки запросов клиентов используются функции: CreateIoCompletionPort, PostQueuedCompletionStatus и GetQueued-CompletionStatus.

Первая функция используется для создания порта завершения ввода/вывода. Прототип этой функции следующий:

```
HANDLE CreateIoCompletionPort (
// файл, ассоциированный с портом завершение ввода/вывода
    HANDLE FileHandle,
    HANDLE ExistingCompletionPort, // существующий порт
    DWORD CompletionKey, // ключи, задающие опции

// число потоков в ассоциированные с портом
    DWORD NumberOfConcurrentThreads
   );
```

С помощью функции PostQueuedCompletionStatus посылается запрос в очередь на обработку ввода/ вывода. Протопип функции следующий:

```
BOOL PostQueuedCompletionStatus(
//дескриптор порта завершения ввода/вывода
    HANDLE CompletionPort,
// число переданных данных
    DWORD dwNumberOfBytesTransferred,
    DWORD dwCompletionKey,    // ключ завершения
    LPOVERLAPPED lpOverlapped // адрес буфера перекрытия
   );
```

С помощью функции GetQueuedCompletionStatus осуществляется выборка запроса из очереди запросов порта завершения ввода/вывода. Протопип функции следующий:

```
BOOL GetQueuedCompletionStatus(

    HANDLE CompletionPort,    // порт завершения ввода вывода
//адрес переменной, куда будет записано число переданных байт
    LPDWORD lpNumberOfBytesTransferred,
```

```
// адрес переменной, куда будет записан ключ завершения
    LPDWORD lpCompletionKey,
// адрес структуры перекрытия
    LPOVERLAPPED *lpOverlapped,
    DWORD dwMilliseconds // время ожидания в миллисекундах
   );
```

4.7.3 Пример организации пула потоков

В этом примере создается простейший порт завершения ввода/вывода. Общее число потоков 5. В функцию потока передается дескриптор порта и номер потока. В главной программе первоначально создается порт завершения ввода/вывода. Затем создается 5 потоков для обработки очереди запросов. Затем в очередь посылается 15 запросов. Затем в очередь посылается 5 запросов для завершения потоков. Далее, ожидается завершение всех потоков.

В процессе демонстрации работы пула потоков на экран выдаются ящики сообщений в которых отображается номер потока и номер обрабатываемого запроса. По завершению программы, все дескрипторы закрываются и выдается ящик сообщения «Конец всеее!!!!».

```
#include "stdafx.h"
#define MAXQUERIES   15
#define CONCURENTS    5
#define POOLSIZE                  5
//задаем тип структуры
typedef struct {int nThread; HANDLE  hcport; } KruStruct;

unsigned __stdcall PoolProc( void *arg );

//главная программа
int _tmain(int argc, _TCHAR* argv[])
{
  int     i;
  HANDLE  hcport, hthread[ POOLSIZE ];
  DWORD   temp;
  KruStruct hsl[POOLSIZE];
  /* создаем порт завершения ввода-вывода */
  hcport = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL,
NULL, CONCURENTS);

/* создаем пул потоков */
for ( i = 0; i < POOLSIZE; i++ ) {
       hsl[i].nThread=i+1;
       hsl[i].hcport=hcport;

         hthread[i]=(HANDLE)
         _beginthreadex(
         NULL,
         0,
```

```
         PoolProc,
         (void*)&hsl[i],
         0,
         (unsigned *)&temp);
}

/* посылаем несколько запросов в порт */
for ( i = 0; i < MAXQUERIES; i++ ) {
  PostQueuedCompletionStatus( hcport, 1, i, NULL );
  Sleep( 60 );
}

/* для завершения работы посылаем специальные запросы */
for ( i = 0; i < POOLSIZE; i++ ) {
   PostQueuedCompletionStatus(hcport,0, (ULONG_PTR)-1,NULL);
}
/* дожидаемся завершения всех потоков пула
     и закрываем описатели */
MessageBox(NULL,"Ждем","End",MB_OK);
WaitForMultipleObjects( POOLSIZE, hthread, TRUE, INFINITE );
for ( i = 0; i < POOLSIZE; i++ ) CloseHandle( hthread[i] );
CloseHandle( hcport );
MessageBox(NULL,"Конец всеее!!!!","End",MB_OK);
return 0;
}

//рабочий элемент (функция потока)

unsigned __stdcall PoolProc( void *arg )
{
  DWORD        size;
  ULONG_PTR    key;
  LPOVERLAPPED      lpov;
  KruStruct *phsl=(KruStruct*)arg;
/*производим выборку из очереди, если очередь пуста, ждем вре-
мя INFINITE*/
  while (
    GetQueuedCompletionStatus(
       phsl->hcport,&size, &key, &lpov, INFINITE
  )) {
         char str[20];
    /* проверяем условия завершения цикла */
    if ( !size && key == (ULONG_PTR)-1 ) break;
       wsprintf(str,"Thread %d Query %d",phsl->nThread,key);
       MessageBox(NULL,str,"hhh",MB_OK);
    Sleep( 300 );
  }
  return 0L;
}
```

5 ПРОСТЕЙШЕЕ СЕТЕВОЕ ПРИЛОЖЕНИЕ, ОСНОВАННОЕ НА СОКЕТАХ

Большинство сетевых приложений, используемых в настоящее время реализовано на основе семейства протоколов TCP/IP. Рассмотрим пример простейшего сетевого приложения, основанного на модели «клиент/сервер».

5.1 Сервер

Сервер является многопоточным. В первичном потоке создается сокет, присваивается локальный адрес, вызывается функция прослушивания запросов.

Затем организуется цикл приема и обработки запроса клиентского приложения. В этом цикле работает функция accept() и функция запуска потока обработки запроса.

Функция потока Request() посылает клиенту запрос «кто ты?», затем принимает строку символов от клиента, создает файл протокола, имя которого зависит от счетчика и записывает туда строку. Ниже записан исходный текст сервера с комментариями.

```
#include <windows.h>
#include <winsock.h>
#include <process.h>
#include <io.h>
#include <stdio.h>
#pragma hdrstop
//номер порта сервера
#define SRV_PORT 1234
#define BUF_SIZE 64
#define TXT_QUEST "Who are you?\n"

int code;                //код завершения
volatile long count; //счетчик запросов
char NameProtocol[20]; //имя протокола
unsigned int ret;
DWORD len;
//потоковая функция обработки запроса
unsigned _stdcall Request(void* ptr_s) {
//принимаем сокет клиента
  int sclient = (reinterpret_cast<int>(ptr_s));
  char buf[BUF_SIZE];
  int from_len;
  //посылаем запрос «кто ты?»
   send(sclient, TXT_QUEST, sizeof(TXT_QUEST),0);
//принимаем ответ от клиента
      from_len = recv(sclient, buf, BUF_SIZE,0);
     if(from_len>0){
```

```
//берем номер, чтобы два потока не схватили один и тот же
//номер, используем функцию InterlockedIncrement().
          int num=InterlockedIncrement(&count);
//формируем имя файла
          wsprintf(NameProtocol,"InnaPro%04d.pro",num);
//создаем файл
HANDLE  hf=CreateFile(
NameProtocol,
GENERIC_WRITE,
0,NULL,CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL,NULL);
//записываем в файл принятую строку
WriteFile(hf,buf,from_len,&len,NULL);
//закрываем файл
 CloseHandle(hf);
  }
  shutdown (sclient, 0);
  if(WSAGetLastError()!=0) MessageBox(NULL, "ErrShut-
Down","Serv", MB_OK);
  closesocket(sclient);
  if(WSAGetLastError()!=0) MessageBox(NULL, "ErrCloseSock-
et","Serv", MB_OK);
  return 0;
}

//главная программа
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{
   int s, s_new;
   WSADATA info;
   int from_len;
   char buf[BUF_SIZE];
   struct sockaddr_in sin, from_sin;

       //MessageBox(NULL, "Begin", "Serv", MB_OK);
//загружаем библиотеку winsock.dll
       if (WSAStartup(MAKELONG(1, 1), &info) == SOCKET_ERROR)
{
    MessageBox(NULL, "Could not initialize socket library.",
               "Startup", MB_OK);
    return 1;
       }
//создаем сокет для протокола TCP
    s = socket (AF_INET, SOCK_STREAM, 0);
//организуем структуру sockaddr
    memset ((char *)&sin, '\0', sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = SRV_PORT;
  int tmp;
```

```
  if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR,(char*) &tmp, 
sizeof(tmp)) < 0) {
          code=WSAGetLastError();
          MessageBox(NULL, "ErrBind", "Bind", MB_OK);
          return 0;
       }
//связывает сокет с локальным адресом
    if(bind (s, (struct sockaddr *)&sin, sizeof(sin))!=0)
       {
          code=WSAGetLastError();
          MessageBox(NULL, "ErrBind", "Bind", MB_OK);
          return 0;
       }
//прослушиваем канал, очередь длиной 3.
    listen (s, 3);
//организуем цикл приема и обработки запросов
from_len = sizeof(from_sin);
while (1) {
//ждем и принимаем запрос от клиентского приложения
  s_new = accept(s, (struct sockaddr *)&from_sin, &from_len);
//запускаем поток на обработку запроса
    HANDLE hHandle = reinter-
pret_cast<HANDLE>(_beginthreadex(0,0,Request,(void*)
s_new,0,&ret));
//закрываем дескриптор потока
CloseHandle(hHandle);
}
//выгружаем библиотеку winsock.dll
  WSACleanup();
   return 0;

}
```

5.2 Клиентское приложение

Клиентское приложение вначале использует функцию Send Server, которая организует связь с сервером и посылает туда сообщение.

```
#include <windows.h>
#include <winsock.h>
#include <io.h>
#define SRV_HOST "Ws317-2"
#define CLNT_PORT 1235
#define SRV_PORT 1234
#define BUF_SIZE 64
#define TXT_QUEST "Who are you?\n"
#define TXT_ANSW "I am your client\n"

//Функция «послать на сервер».
int SendServer(
   char *svr_host, //имя сервера
```

```
   int svr_port,      //порт сервера
   char *buf,         //что послать
   int lenbuf          //длина
)
{
     SOCKET s;
        char bufrecv[20];
        WSADATA info;
     int from_len;
     struct hostent *hp;
     struct sockaddr_in clnt_sin, srv_sin;
        int err;
    //создаем сокет
     s = socket (AF_INET,  SOCK_STREAM, 0);
     if((err=WSAGetLastError())!=0) return err;

     //заполняем структуру для  адреса сервера

     memset ((char *)&srv_sin, '\0', sizeof(srv_sin));
     hp = gethostbyname (svr_host);

     srv_sin.sin_family = AF_INET;
     memcpy ((char *)&srv_sin.sin_addr,hp->h_addr,hp-
>h_length);
     srv_sin.sin_port = svr_port;
//делаем запрос на соединение
        connect (s, (struct sockaddr *)&srv_sin,
sizeof(srv_sin));
     if((err=WSAGetLastError())!=0) return -err;

//принимает «кто ты?»
     from_len = recv (s, bufrecv,20, 0);
     if((err=WSAGetLastError())!=0) return err;
//посылаем сообщение
    send (s, buf, lstrlen(buf), 0);
        if((err=WSAGetLastError())!=0) return err;
     closesocket(s);
        if((err=WSAGetLastError())!=0) return err;
        return 0;
}

//главная функция клиентского приложения
WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
LPSTR lpCmdLine, int nCmdShow)
{

        WSADATA info;
          int err;
        char errs[20];

    MessageBox(NULL, "Begin", "Client", MB_OK);
```

```
//грузим библиотеку winsock.dll
        if (WSAStartup(MAKELONG(1, 1), &info) == SOCKET_ERROR)
{
    MessageBox(NULL, "Could not initialize socket library.",
               "Startup", MB_OK);
    return 1;
        }
//посылаем первый запрос
        if((err=SendServer("ws317-2",1234,"Привет всем всем
всем!!!",24))!=0) {
           wsprintf(errs,"Ошибка %d",err);
           MessageBox(NULL,errs,"Client",MB_OK);
        }
//посылаем второй запрос
        if(SendServer("ws317-2",1234,"Привет
12344444444444uuuu!!!",27)!=0){
        wsprintf(errs,"Ошибка %d",err);
        MessageBox(NULL,errs,"Client",MB_OK);
        }
        WSACleanup ();
         exit (0);
 }
```

6 РАЗРАБОТКА СЕТЕВЫХ ПРИЛОЖЕНИЙ НА ОСНОВЕ WWW-СЕРВЕРА

6.1 Обзор технологий

Обычная схема взаимодействия WWW-сервера и браузеров основана на статичном представлении html-страниц (рис. 6.1). Однако, развитие всемирной паутины привело к необходимости генерировать веб-страницы динамически. Реализация этой идеи позволило создавать разнообразные функциональные WWW-сервера. Такие как виртуальные магазины, биржи, банки, университеты и пр.

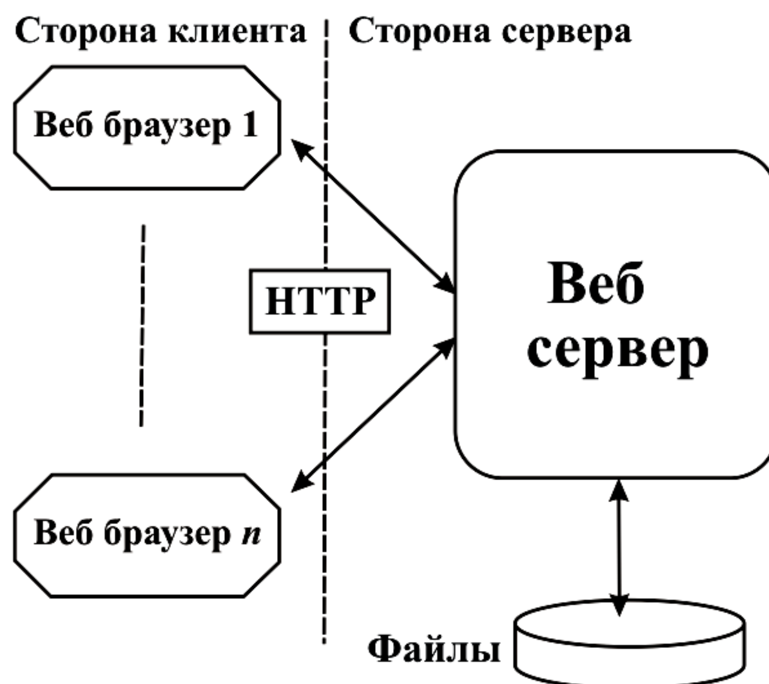

Рис. 6.1 – Взаимодействие веб-браузеров и сервера

В основе этой новой технологии лежит возможность расширения функциональности WWW-сервера за счет присоединения дополнительных программ.

WWW сервер, проанализировав запрос клиента, может создать дочерний процесс, запустив в нем некоторую exe-программу. Эта программа обработает запрос и сгенерирует ответ в виде html-страницы, которую передаст серверу, а сервер– клиенту (рис. 6.2).

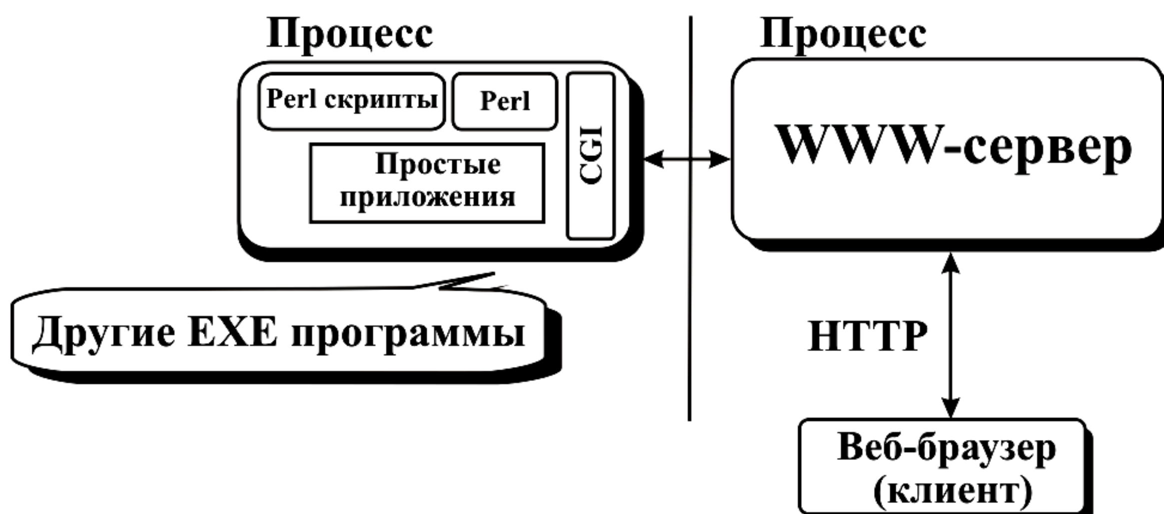

Рис. 6.2 – Механизм расширения функциональности сервера

В настоящее время имеется несколько различных технологий, позволяющих строить такие расширения.

1. CGI (Common Gateway Interface) является стандартом интерфейса, служащего для связи внешней программы с WWW-сервером. Программы, работающие по такому интерфейсу совместно с WWW-сервером, называют скриптами или CGI-программами.

2. ISAPI (Internet Server Application Programming Interface) разработан фирмой Microsoft для расширения функциональных возможностей своего сервера IIS (Internet Information Services).

Данный интерфейс предполагает, что функциональное расширение представлено в виде DLL.

3. SSI (Server Side Includes) – технология, позволяющая использовать возможности препроцессора Си, для сбора html-страницы.

4. PHP: Hypertext Preprocessor – это широко распространенный, поддерживающий концепцию открытого кода, скриптовый язык, предназначенный для расширения функциональных возможностей WWW-сервера.

Рассмотрим простой пример:

```
<html>
    <head>
        <title>Example</title>
    </head>
    <body>

        <?php
        echo "Hi, I'm a PHP script!";
        ?>

    </body>
</html>
```

Из приведенного примера видно, что утверждения PHP вставляются в HTML-страницу и при просмотре страницы сервером, вызывается интерпретатор PHP, который генерирует HTML-текст. Особенно хорошо зарекомендовал себя PHP для работы с базами данных.

ASP (Active Server Pages) – это технология фирмы Microsoft, которая генерирует html-страницы на стороне сервера. Эта технология разработана для сервера. Тем самым программирование сайтов на ASP стало простым и быстрым, за счет большого числа встроенных объектов. Несколько скриптовых языков может поддерживать ASP технология, однако основным языком является VBScript. Рассмотрим несколько примеров:

1. Вставка строки "Hello World!" в тело HTML страницы.

```
1.  <html>
2.  <body>
3.  <% Response.Write("Hello World!") %>
4.  </body>
5.  </html>
```

Или

```
1.  <html>
2.  <body>
3.  <%= "Hello World!" %>
4.  </body>
5.  </html>
```

2. Ниже приведен пример доступа к базе данных MS Access Database.

```
1.  <%
2.      Set oConn = Server.CreateObject("ADODB.Connection")
3.      oConn.Open "DRIVER={Microsoft Access Driver (*.mdb)};
    DBQ=" & Server.MapPath("DB.mdb")
4.      Set rsUsers = Server.CreateObject("ADODB.Recordset")
5.      rsUsers.Open "SELECT * FROM Users", oConn
6.  %>
```

JSP (Java Server Pages) – это Java технология, которая позволяет создавать программы генерации HTML, XML и других типов документов на стороне www сервера. В основе этой технологии лежит применение языка Java.

Широкие возможности для программирования WWW-расширений предлагают системы Delphi, CBuilder, VC.

6.2 Программирование CGI-скриптов

6.2.1 Описание интерфейса

Рассмотрим подробнее программирование CGI-скриптов. *CGI-скрипт* – это exe-программа, которая будет вызываться www-сервером. Передача данных CGI-программе осуществляется в следующем формате:

имя=значение&имя1=значение1&...

Здесь "имя" это название параметра, а "значение"– его содержимое. Методов передачи данных в таком формате существует два – **GET** и **POST**. При использовании метода **GET** данные передаются серверу вместе с **URL**:

http://.../cgi-bin/test.cgi?имя=значение&имя1=значение1&...

При использовании метода **POST** данные посылаются внутри самого **HTTP** запроса.

Так как длина **URL** ограничена, то методом **GET** нельзя передать большой объем данных, а метод **POST** обеспечивает передачу данных, не ограниченных по длине.

Получение данных самим скриптом также различается. При использовании метода **GET** данные следующие за "**?**" помещаются в переменную среды **QUERY_STRING**. При использовании **POST** содержимое запроса перенаправляется в стандартный поток ввода, т.е. в **stdin**.

Для того, чтобы CGI – программа могла узнать какой метод используется для передачи данных, сервер создает переменную среды **REQUEST_METHOD**, в которую записывает **GET** или **POST**.

Весь процесс получения данных от WWW-сервера можно представить следующим алгоритмом:

1. Получить все необходимые переменные окружения, используя функцию GetEnvironmentString(), которая возвращает указатель на строку, содержащую список пар.

```
<имя_1>=<значение_1>
<имя_2>=<значение_2>
…
<имя_i>=<значение_i>
```

Пример кода для печати всех переменных окружения

```
LPTSTR lpszVariable;
LPVOID lpvEnv;

// получаем указатель на строку, содержащую все пары
 lpvEnv = GetEnvironmentStrings();

// Все пары отделены между собой  NULL байтом, и весь блок
 // также имеет  NULL байт в конце

//организуем цикл для печати пар «имя/значение»
for (lpszVariable = (LPTSTR) lpvEnv; *lpszVariable; lpszVaria-
ble++)
{
```

```
   while (*lpszVariable) //печатаем найденную пару
      putchar(*lpszVariable++);
   putchar('\n'); //если ноль, печатаем перевод строки
}
```

Для получения значения конкретной переменной используется функция GetEnvironmentVariable. Прототип этой функции следующий:

```
DWORD GetEnvironmentVariable(
LPCTSTR lpName,// имя переменной окружения
//адрес буфера для получения значения переменной
LPTSTR lpBuffer,
DWORD nSize    // длина буфера
   );
```

При успешном завершении, функция запишет значение переменной в буфер и вернет количество символов в значении переменной. Если такой переменной не будет найдено, то функция вернет ноль, если размер буфера меньше значения переменной, то функция вернет размер значения переменой, а в буфер писать не будет.

Получение данных от сервера в зависимости от метода передачи:

Если переменная REQUEST_METHOD имеет значение "GET", то взять данные из переменной окружения QUERY_STRING. Если переменная REQUEST_METHOD имеет значение "POST", то необходимо получить длину данных из переменной окружения CONTENT_LENGTH. Затем считать данные из стандартного ввода (stdin). Далее декодировать и разбить на пары "имя=значение".

Считывание данных через поток **sdtin** должно осуществляться в динамическую память, или же во временный файл, в случае если размер памяти ограничен или данные слишком велики для полного размещения в ОЗУ.

Пример,

```
  char *cgi_data;
  ...
  long content_length=atol(getenv("CONTENT_LENGTH"));
  cgi_data=(char *)malloc(content_length);
  if (cgi_data!=NULL)
     fread(cgi_data,content_length,1,stdin);
  ...
```

6.2.2 Взаимодействие WWW-сервера и CGI-программы

При обслуживании запроса клиента, требующего вызов CGI-скрипта, сервер создает поток и дочерний процесс, в котором будет исполняться CGI-программа. Общение потока сервера и дочернего процесса осуществляется через неименованный канал (anonymous pipe). Для CGI-программы

соответствующим образом настраивается stdin и stdout. Данные для скрипта записываются в stdin, а данные клиенту скриптом записываются в stdout, например, функцией printf(). После того, как скрипт завершил работу данные отправляются коиенту.

Если процесс взаимодействия между клиентом и скриптом носит диалоговый характер, т.е. текущий сеанс зависит от предыдущих, то для каждого клиента необходимо создавать специальный файл или базу данных, где будет установлено состояние клиента на текущий момент. Анализ этого состояния позволит соответственно реагировать на запрос клиента в текущем диалоге.

Поскольку одновременных запросов клиентов может быть некоторое множество, то соответственно в памяти сервера будет запущено некоторое множество дочерних процессов. Это означает, что для доступа к ресурсам необходимо использовать синхронизацию на основе объектов ядра Windows: событий, семафоров, мьютексов и пр.

Самым простым выходом из данной ситуации является создание специального инициализирующего процесса, который устанавливает значения разделяемых ресурсов и создает объекты синхронизации. Такой процесс должен запускаться и останавливаться вместе с запуском и остановкой сервера. Серверный и инициализирующий процессы могут быть логически не связаны.

Далее запускаются дочерние CGI-процессы, которые открывают соответствующие объекты и синхронизируются с помощью Wait-функций.

6.2.3 Переменные среды о сервере

Эти переменные сервер устанавливает для того, чтобы скрипт мог узнать, с каким сервером он работает. Сюда входят данные о портах сервера, его версии, типе интерфейса CGI и т.д. В каждой версии сервера часто прибавляются новые переменные, но следующие переменные должен устанавливать сервер любой версии.

GATEWAY_INTERFACE

Указывает версию интерфейса CGI, который поддерживает сервер. Например:
CGI/1.1

SERVER_NAME

Содержит IP адрес сервера или его доменное имя. Например:
www.ie.tusur.ru

SERVER_PORT

Номер порта, по которому сервер получает http запросы. Стандартный порт для этого 80.

SERVER_PROTOCOL

Версия протокола http, который использует сервер для обработки запросов. Например:

HTTP/1.1

SERVER_SOFTWARE

Название и версия программы сервера. Например:
Apache/1.3.3 (Unix) (Red Hat/Linux)

Эти переменные обеспечивают все необходимые данные о сервере, на котором запускается скрипт. Если сервер сконфигурирован для работы с одним хостом, то вероятнее, что информация эта не понадобится. Сейчас большинство серверов позволяют создавать так называемые "виртуальные" хосты. Это один компьютер, который поддерживает много **IP** адресов и различает запросы от клиентов по требуемому хосту, на которые соответственно выдает странички с сайтов. Тут уже могут понадобиться данные о портах сервера (т.к. многие хосты просто "сидят" на других портах, например, 8080, 8081 и т.д.) и его **IP** адрес с именем.

При помощи переменных данного типа скрипт узнает полную информацию о запросе к нему. Т.е. каким методом будут передаваться данные, их тип, длину и т.д.

AUTH_TYPE

Тип авторизации используемой сервером. Например:
Basic.

CONTENT_FILE

Путь к файлу с полученными данными. Используется только в серверах под Windows. Например:
c:\website\cgi-temp\103421.dat

CONTENT_LENGTH

Длина переданной информации в байтах. Т.е. сколько надо считать байтов из **stdin**. Например:
10353

CONTENT_TYPE

Тип содержимого посланного серверу клиентом. Например:
text/html

OUTPUT_FILE

Файл для вывода данных, используется только серверами под Windows. Аналогично CONTENT_FILE.

PATH_INFO и **PATH_TRANSLATED**

В современных веб-серверах стало возможным после имени скрипта указывать еще какой-то определенный путь. Эти переменные работают следующим образом. Предположим, существует скрипт с именем **1.cgi** в каталоге сервера **/cgi-bin**, тогда при вызове скрипта в виде:

http://.../cgi-bin/1.cgi/dir1/dir2

данные переменные установятся следующим образом:

PATH_INFO=/dir1/dir2
PATH_TRANSLATED=/home/httpd/html/dir1/dir2

Очевидно, что эти переменные указывают на папку относительно корневой директории сервера. При этом **PATH_TRANSLATED** будет содержать абсолютный путь до этого каталога на диске сервера. В данном случае корневым каталогом сервера считается **/home/httpd/html/**, это путь в **Unix**–системах.

Под **dos/win** системами переменная **PATH_INFO** не изменится, а **PATH_TRANSLATED** будет содержать **d:\apache\htdocs\dir1\dir2** (в данном случае корнем сервера является директория **d:\apache\htdocs**).

QUERY_STRING

Содержит данные переданные через **URL**. Такие данные указываются после имени скрипта и знака «**?**». Пример:

http://.../cgi-bin/1.cgi?d=123&name=kostia
тогда переменная **QUERY_STRING** будет содержать

d=123&name=kostia

Данные, передаваемые таким образом кодируются методом **URL**.

REMOTE_ADDR

Содержит **IP**–адрес пользователя пославшего запрос скрипту. Если обращаетесь к любому скрипту в интернете, то данная переменная будет содержать ваш **IP**–адрес. Пример:
192.148.1.26

REMOTE_HOST

Содержит доменное имя, при условии, что прописан на каком-либо **DNS** сервере.

REQUEST_METHOD

Содержит метод передачи данных шлюзу: **GET** или **POST**.

REQUEST_LINE

Содержит строку из запроса протокола HTTP. Например:
GET /cgi-bin/1.cgi HTTP/1.0

SCRIPT_NAME

Содержит имя вызванного скрипта. Например: **1.cgi**.

HTTP_ACCEPT

Эта переменная перечисляет все типы данных, которые может получать и обрабатывать клиент. Часто она содержит просто ***/***, т.е. клиент может получать все подряд. Пример:
/,image/gif,image/x-xbitmap

HTTP_REFERER

Содержит **URL** страницы, с которой был произведен запрос, т.е. которая содержит ссылку на шлюз. Пример:

http://www.firststeps.ru/index.html

HTTP_USER_AGENT

Содержит в себе название и версию браузера, которым пользуется клиент для запроса.

HTTP_ACCEPT_ENCODING

Указывает набор кодировок, которые может получать клиент. Например:
koi8-r, gzip, deflate

HTTP_ACCEPT_LANGUAGE

Содержит в себе список языков в кодах **ISO**, которые может принимать клиент. Например:
ru, en, fr

HTTP_IF_MODIFIED_SINCE

Содержит в себе дату, которая указывает на то, что получаемые данные должны не ранее данной.

HTTP_FROM

Содержит список почтовых адресов клиента.

Таким образом, программа может сразу вызвать браузер с уже подготовленной страничкой и пользователю не придется даже знать адрес этого поисковика и как он работает, все знает программа.

```
#include <stdio.h>
#include <stdlib.h>

//Здесь надо вставить процедуру получения
//параметра по его имени... Она была описана
//раньше.

char *getparam(...)
{
};

int main()
{
  char *user=NULL;
  char *content=NULL;
  char *request_method=getenv("REQUEST_METHOD");
  if (strcmp(request_method,"GET")!=0)
       {
          printf("Content-type: text/html\n\n");
          printf("Unknown REQUEST_METHOD. Use only GET !\n");
          return -1;
       };

  content=getenv("QUERY_STRING");
  user=getparam(content,"user=");
  printf("Content-type: text/html\n\n");
```

```
  printf("User name=\"%s\"\n",user);
};
```

После того как программа будет собрана и скомпилирована, расположите ее в директории **cgi-bin** веб-сервера. Теперь можно записать:

http://localhost/cgi-bin/primer.cgi?user=hello

Результат:

User name="hello"

6.3 Программный интерфейс ISAPI

Технология CGI имеем один важный недостаток, особенно это касается операционной системы Windows – на каждый запрос клиента WWW-сервер создает процесс. Это приводит к снижению эффективности работы системы. Для устранения этого недостатка фирма Microsoft разработала специальный интерфейс, который позволяет подключать DLL библиотеки. Это позволяет внедрять функциональные расширения в основной процесс сервера, что существенно повышает эффективность выполнения скрипта, по сравнению с CGI.

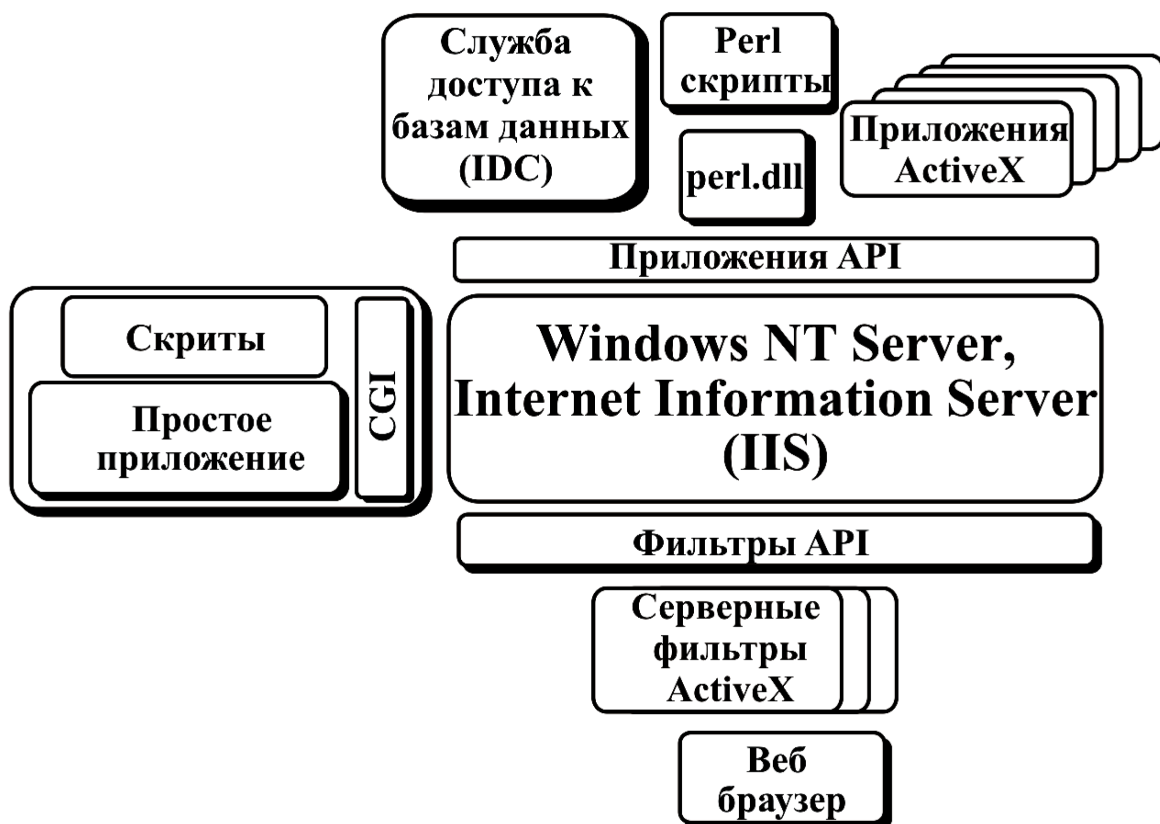

Рис. 6.3 – Структура IIS с интерфейсами CGI и API

Процесс работы скрипта, разработанного на интерфейсе API, следующий (рис. 6.4): веб-браузер создает запрос и передает на сервер, сервер принимает URL, анализирует, грузит соответствующую DLL, если это надо, (в примере my_app.dll) и вызывает соответствующую функцию DLL, которая обрабатывает запрос клиента.

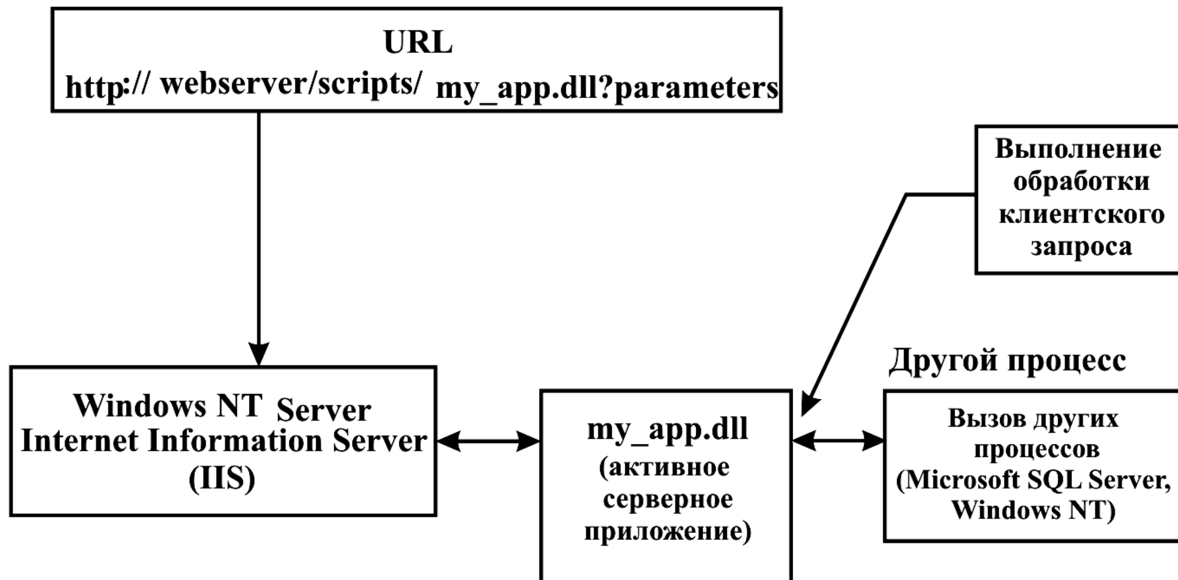

Рис. 6.4 – Обработка запроса клиента

Интерфейс ISAPI имеет две функции: GetExtensionVersion() HttpExtensionProc() и одну структуру данных EXTENSION_CONTROL_BLOCK.

Функция GetExtensionVersion() предназначена для определения версии ISAPI и может быть использована для вспомогательных целей. Прототип функции следующий:

```
BOOL WINAPI
GetExte nsionVersion( HSE_VERSION_INFO  *  pVer );
```

Пример:

```
BOOL WINAPI GetExtensionVersion( HSE_VERSION_INFO  *  pVer ) {
  pVer->dwExtensionVersion =
       MAKELONG( HSE_VERSION_MINOR, HSE_VERSION_MAJOR );
  lstrcpyn( pVer->lpszExtensionDesc,
                "Sample Web Server Application",
                HSE_MAX_EXT_DLL_NAME_LEN );
  return TRUE;
 } // GetExtensionVersion()
```

Функция HttpExtensionProc() является эквивалентом main в CGI-интерфейсе, вызывается всякий раз, как только запросит клиент. Посколь-

ку клиенты могут сделать запросы одновременно, она может быть вызвана в разных потоках. Прототип следующий:

```
DWORD WINAPI
HttpExtensionProc(EXTENSION_CONTROL_BLOCK  *  pECB);
```

где структура EXTENSION_CONTROL_BLOCK имеет следующий вид:

```
typedef struct _EXTENSION_CONTROL_BLOCK {
    DWORD     cbSize;              // размер структуры
    DWORD     dwVersion;           // версия
    HCONN     ConnID;              // дескриптор соединения
    DWORD    dwHttpStatusCode;    // HTTP код статуса
    CHAR        lpszLogData[HSE_LOG_BUFFER_LEN];
    LPSTR       lpszMethod;              // метод (GET, POST)
    LPSTR       lpszQueryString;       // Строка запроса
    LPSTR       lpszPathInfo;               // путь
    LPSTR       lpszPathTranslated;         //путь
    DWORD    cbTotalBytes;      // общее число байт от клиента
    DWORD    cbAvailable;       // число байт в наличии
    LPBYTE    lpbData;          // указатель на данные
    LPSTR       lpszContentType;   // тип данных от клиента
BOOL (WINAPI * GetServerVariable)
        (ConnID, lpszVariableName, lpvBuffer, lpdwSize );

BOOL (WINAPI * WriteClient)
        (ConnID, Buffer, lpdwBytes, dwReserved );

BOOL (WINAPI * ReadClient)
        (ConnID, lpvBuffer, lpdwSize );

BOOL (WINAPI * ServerSupportFunction)
        (ConnID, dwHSERRequest, lpvBuffer, lpdwSize, lpdw-
DataType );

} EXTENSION_CONTROL_BLOCK,
   LPEXTENSION_CONTROL_BLOCK;
```

Функция GetServerVariable() предназначена для чтения значения серверной переменной по имени.

```
BOOL
(WINAPI * GetServerVariable) (HCONN ConnID,
                    LPSTR lpszVariableName,
                    LPVOID lpvBuffer,
                    LPDWORD lpdwSize);
```

Эта функция является аналогом функции getenv() в CGI.

Функция ReadClient() предназначена для чтения данных от клиента. Прототип функции следующий:

```
BOOL
(WINAPI * ReadClient) (HCONN     ConnID,
                LPVOID     lpvBuffer,
                LPDWORD lpdwSize );
```

Функция WriteClient() предназначена для записи данных клиенту. Прототип функции следующий:

```
BOOL
(WINAPI * WriteClient) (HCONN ConnID,
               LPVOID   Buffer,
               LPDWORD  lpdwBytes,
               DWORD         dwReserved );
```

Эта функция эквивалентна записи в stdout для CGI

Функция ServerSupportFunction() выполняет множество разнообразных сервисных функций: переадресацию запроса, формирование заголовка, управление сессией и др.

```
BOOL (WINAPI * ServerSupportFunction) (
          HCONN     ConnID,
          DWORD     dwHSERRequest,
          LPVOID    lpvBuffer,
          LPDWORD   lpdwSize,
          LPDWORD   lpdwDataType );
```

Пример использования функции HttpExtensionProc, в котором на каждый запрос клиента выдается счетчик числа запросов.

```
//описываем счетчик
static int hits = 0;

DWORD WINAPI
HttpExtensionProc( LPEXTENSION_CONTROL_BLOCK ecb )
{

//описываем заголовок
   char *header   =  "Content-Type: text/plain";
   int headerlen  = strlen( header );
   char msg[256];
   int msglen;

   /* используем серверную функцию для записи заголовка */
   ecb->ServerSupportFunction( ecb->ConnID,
HSE_REQ_SEND_RESPONSE_HEADER, 0,
                               &headerlen, (DWORD *)header );
```

```
   /* организуем строку для вывода числа запросов */
   sprintf( msg, "Стрница была показана %d раз", Interlocke-
dIncrement(&hits));
   msglen = strlen( msg );
//отослать клиенту
   ecb->WriteClient( ecb->ConnID, msg, &msglen, 0 );

   /*вернуть успешное завершение */
   return HSE_STATUS_SUCCESS;
}
```

6.4 Фильтры IIS

Фильтр – это тип ISAPI модуля, предназначенный для предварительной обработки запросов клиентов. Данный тип модуля позволяет настроить сервер на обработку клиентских запросов, не предусмотренный стандартными средствами сервера.

Фильтры используются в специализированных приложениях, связанных с IIS, и обычно выполняют следующие задачи:

- шифрование;
- ведение журналов;
- аутентификация;
- сжатие данных.

Расширения ISAPI – наиболее частый способ применения ISAPI. Фильтры ISAPI довольно сложны в создании, и сфера их использования ограничена. Для создания фильтра необходимо создать DLL с функциями: GetFilterVersion(), HttpFilterProc()

Кроме того, нужно знать две структуры: флаги уведомления фильтра и структура контекста фильтра. Фильтр может быть настроен:

- на фильтр может быть вызван при чтении данных от клиента, при этом можно изменить данные или перенаправить в другой каталог, расшифровать и прочее;
- фильтр может быть вызван при передачи данных клиенту, в этом случае данные могут быть изменены, закодированы и т.д.
- фильтр может быть вызван при обработке заголовков и занесении их в таблицу;
- фильтр может быть вызван для аутентификации, при необходимости создания не стандартной схемы.

Вот не полный перечень того, что могут делать фильтры.

Пример 1. Поддержка HTTP Cookies.

Cookies – специальный механизм настройки HTML-документа на локальном компьютере клиента. Используя фильтр, можно построить такой

механизм настройки для конкретных клиентов, которые обращаются на данный сервер.

Инициализация фильтра осуществляется вызовом функции *GetFilterVersion()*. Ниже приведена конкретная функция, в которой устанавливаются значения флагов уведомления и имя фильтра.

```
#include "httpfilt.h"
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

BOOL WINAPI
GetFilterVersion( HTTP_FILTER_VERSION *pVer )
{
   pVer->dwFilterVersion = HTTP_FILTER_REVISION;
   strncpy( pVer->lpszFilterDesc, "A Cookie Filter",
SF_MAX_FILTER_DESC_LEN );
   /* установка флагов уведомлений */
   pVer->dwFlags = SF_NOTIFY_SECURE_PORT |
                   SF_NOTIFY_NONSECURE_PORT |
                   SF_NOTIFY_PREPROC_HEADERS;
   return TRUE;
}
```

Всю остальную обработку производит функция *HttpFilterProc*, которую будет вызывать сервер. В этом примере при получении cookie и URL при обработке заголовка запроса клиента. Если в заголовке присутствует cookie, то соответствующий URL и cookie записываются в специальный файл. Если в заголовке не присутствует cookie, то случайно его генерируем и передаем его клиенту.

```
static void
RandomBytes( char *buffer, int count )
{
   /* заполняем буфер случайными цифрами*/
   int i;
   for( i=0; i<count; i++ ) buffer[i] = '0' + (rand() % 10);
   buffer[i] = '\0';
}

DWORD WINAPI
HttpFilterProc(
                PHTTP_FILTER_CONTEXT pfc,
                DWORD notificationType,
                VOID *pvNotification
 )
{
   /* преобразуем к SF_NOTIFY_PREPROC_HEADERS */
```

```
   HTTP_FILTER_PREPROC_HEADERS *headers
        = (HTTP_FILTER_PREPROC_HEADERS *) pvNotification;
   char cookie[256],     url[256];
   int  cookielen = 256, urllen=256;

   /* берем информацию из заголовка*/
   if( headers->GetHeader( pfc, "Cookie:", cookie, &cookielen
) &&
       headers->GetHeader( pfc, "url", url, &urllen ) &&
       cookielen > 1 && urllen > 1 )
      {
      /* если есть cookie */
      int fd = open( "/tmp/cookie.log",
O_WRONLY|O_CREAT|O_APPEND, 0755 );
      if( fd != -1 )
      {
         char outbuff[514];
         sprintf( outbuff, "%s %s\n", cookie, url );
         write( fd, outbuff, strlen( outbuff ) );
         close( fd );
      }
   }
else {
      /* устанвливаем в заголовок */
      char msg[256];
      RandomBytes( cookie, 16 );
      sprintf( msg, "Set-Cookie: %s\r\n", cookie );
      pfc->AddResponseHeaders( pfc, msg, 0 );
   }
   return SF_STATUS_REQ_NEXT_NOTIFICATION;
}
```

Пример 2. Аутентификация

В этом примере показано, как обеспечить контроль доступа на сайт с использованием паспорта. Для простоты, пользователь имеет имя "fred" и паспорт "bloggs". Его легко расширить на использование удаленной базы данных.

Первым делом необходимо записать функцию *GetFilterVersion* для запроса на установку флагов уведомления.

```
#include <string.h>
#include "httpfilt.h"

BOOL WINAPI
GetFilterVersion( HTTP_FILTER_VERSION *pVer )
{
   //устанавливаем имя и версию фильтра
   pVer->dwFilterVersion = HTTP_FILTER_REVISION;
   strncpy( pVer->lpszFilterDesc, "Basic auth filter",
SF_MAX_FILTER_DESC_LEN );
//устанавливаем флаги
```

```
   pVer->dwFlags = SF_NOTIFY_SECURE_PORT |
                   SF_NOTIFY_NONSECURE_PORT |
                   SF_NOTIFY_AUTHENTICATION;
   return TRUE;
}
```

Затем записываем функцию HttpFilterProc(), которая выполняет аутентификацию. Вспомогательная функция Denied() сообщает клиенту, что запрос на аутентификацию не удовлетворен.

```
static void
Denied( PHTTP_FILTER_CONTEXT pfc, char *msg )
{
   int l = strlen( msg );
   pfc->ServerSupportFunction( pfc,
SF_REQ_SEND_RESPONSE_HEADER,
                     (PVOID)   "401 Permission Denied",
                     (LPDWORD) "WWW-Authenticate: Basic
realm=\"foo\"\r\n",
                               0 );
   pfc->WriteClient( pfc, msg, &l, 0 );
}

//обработка запроса на аутентификацию
DWORD WINAPI
HttpFilterProc( PHTTP_FILTER_CONTEXT pfc,
                DWORD notificationType,
                VOID *pvNotification )
{
   /* преобразование указателей*/
   HTTP_FILTER_AUTHENT *auth = (HTTP_FILTER_AUTHENT
*)pvNotification;

   if( auth->pszUser[0] == 0)
    {
      Denied( pfc, "No user/password given" );
      return SF_STATUS_REQ_FINISHED;
   }
   if( strcmp( auth->pszUser, "fred" ) )
    {
      Denied( pfc, "Unknown user" );
      return SF_STATUS_REQ_FINISHED;
   }
   if( strcmp( auth->pszPassword,"bloggs") )
    {
      Denied( pfc, "Wrong Password" );
      return SF_STATUS_REQ_FINISHED;
   }
   return SF_STATUS_REQ_NEXT_NOTIFICATION;
}
```

7 ПРОГРАММИРОВАНИЕ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ PYTHON

Язык Питон является высокоуровневый язык программирования общего назначения, основанный на современных парадигмах программирования: структурного, объектно-ориентированного, функционального, событийно-ориентированного, многопоточного. Он был разработан в 80 годах XX века сотрудником голландского института CWI Гвидо ван Россумом. Язык Питон представлен на всех без исключения операционных системах [18].

Особенность языка Питон является использование интерпретатора на этапе выполнения кода. Поэтому этот язык хорошо подходит для создания прототипа программной системы. Рассмотрим использование средств Питона для создания клиент-серверных приложений. Для этого рассмотрим его возможности для работы с сокетами и создание многопоточных приложения [19].

Сокеты

Для работы с сокетами в языке Python имеется пакет socket, который обеспечивает основные функции для работы с сокетами. Рассмотрим некоторые из них.

Константы

socket.AF_UNIX
socket.AF_INET
socket.AF_INET6
socket.SOCK_STREAM
socket.SOCK_DGRAM
socket.SOCK_RAW
socket.SOCK_RDM
socket.SOCK_SEQPACKET

Методы

- socket.socket(family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None) – функция создания сокета;
- socket.create_connection(address[, timeout[, source_address]]) – создает соединение и взвращает (пару (host, port));
- socket.getaddrinfo(host, port, family=0, type=0, proto=0, flags=0) – преобразует информацию о порте и хосте в 5-элементный список: (family, type, proto, canonname, sockaddr).

Например

```
socket.getaddrinfo("example.org", 80,
proto=socket.IPPROTO_TCP) [(<AddressFamily.AF_INET6: 10>,
<SocketType.SOCK_STREAM: 1>, 6, '',
```

```
('2606:2800:220:1:248:1893:25c8:1946', 80, 0, 0)),
(<AddressFamily.AF_INET: 2>, <SocketType.SOCK_STREAM: 1>,
6, '', ('93.184.216.34', 80))]
```

- socket.gethostbyname(hostname) – возвращает имя хоста в формате адреса IPv4;
- socket.gethostname() – возвращает имя хоста на котором работает интерпретатор Питона;
- socket.gethostbyaddr(ip_address) – возвращает список содержащий следующие элементы (hostname, aliaslist, ipaddrlist);
- socket.getnameinfo(sockaddr, flags) преобразование структуры sockaddr в список (host, port);
- socket.accept() – принять соединение, возвращает пару (conn, address);
- socket.bind(address) – связывает сокет с хостом;
- socket.close() – закрывает сокет (аналогично файловой операции close);
- socket.connect(address) – принимает запрос на соединение с удаленным хостом;
- socket.getsockname() – возвращает собственный адрес сокета;
- socket.listen([backlog]) – ожидает соединения в режиме сервера;
- socket.recv(bufsize[, flags]) – принимает данные от сокета;
- socket.recvfrom(bufsize[, flags]) – принимает данные от сокета, возвращает пару (bytes, address);
- socket.send(bytes[, flags]) – посылает данные сокету;
- socket.sendto(bytes, address) – посылает данные сокету;
- socket.sendto(bytes, flags, address) посылает данные сокету;
- socket.sendfile(file, offset=0, count=None) – посылает файл сокету.

Создание сокетов

Рассмотрим следующий пример:

```
#распределяем память под структуры сокета и заполняем некото-
рые поля
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# производим запрос на соединение с серверомоn по порту 80
s.connect(("www.python.org", 80))
```

Когда запрос на соединение выполнен можно посылать запросы на чтение и передачу данных. После выполнения операций чтения и записи данных необходимо закрыть сокет.

Схема сервера

В тех случаях, когда алгоритм web-сервера более сложен, тогда создается серверный сокет (сокет на стороне сервера). Например,

```
# создаем INET, STREAMing сокет
```

```
serversocket = socket.socket(socket.AF_INET, sock-
et.SOCK_STREAM)
# связываем сокет с хостом, по известному порту 80
serversocket.bind((socket.gethostname(), 80))
# ждем соединения с клиентом
serversocket.listen(5)
```

Необходимо отметить следующее:

1. Использование вызова метода socket.gethostname() позволяет сделать сокет видимым в сети. Если мы используем вызовы s.bind(('localhost', 80)) или s.bind(('127.0.0.1', 80)), то получим серверный сокет, видимый только на заданном компьютере.
2. Нижние номера портов зарегистрированы для “хорошо известных» сервисов (HTTP, SNMP и т.д.).
3. Вызов listen указывает библиотеки сокетов необходимость для сождание очереди запросов (как правило не более 5) до отказа на запрос соединения.

Основной цикл работы сервера по обработке запроса

```
while True:
    # принять запрос и  получить сокет клиента
    (clientsocket, address) = serversocket.accept()
    # выполнить предварительные действия для сокета клиента
    # запустить поток (процесс) обработки запроса клиента
    ct = client_thread(clientsocket)
    ct.run()
```

Обмен данными между с помощью сокетов

В модуле socket имеется набор различных методов для обмена данными. Для чтения recv(), для записи данных send().

Многопоточное приложение на языке Python

Для организации многопоточного приложения в языке Python имеется модуль threading.

Создание многопоточного приложения. Функции модуля threading

- threading.active_count() – возвращает число потоков находящихся в системе. Это число равно списку потоковых объектов возвращаемых функцией enumerate().
- threading.current_thread() – возвращает текущий потоковый объект.
- threading.get_ident() – возвращает идентификатор (handler) потока.
- threading.enumerate() – возвращает список потоковых объектов имеющихся в системе.
- threading.main_thread() – возвращает потоковый объект главного потока, в котором стартовад интерпретатор Python.
- threading.stack_size([size]) – возвращает размер стека потока, который был создат при инициализации потока.

В модуле определены следующие элементы:

1. Константа threading.TIMEOUT_MAX – максимальное время для блокировки функций (Lock.acquire(), RLock.acquire(), Condition.wait(), etc.).

2. Локальная память потока, которая является уникальной для каждого потока, например,

mydata = threading.local()

mydata.x = 1

переменная mydata.x будет иметь разную память для каждого потока. Для использования локальной памяти потока имеется класс threading.local

Потоковые объекты

Класс Thread представляет исполнение кода, которое будет производится в отдельном потоке управления. Имеется две возможности для инициализации исполнения кода: через вызов конструктора объекта или использовать метод run(). Иными словами использовать только __init__() или run() рассматриваемого класса.

После создания объекта thread необходимо вызвать метод start(), это означает, что поток будет выполняться параллельно с другими. Кроме того поток приобретет статус ‘alive’. Статус ‘alive’ будет отменен если поток завершится или произойдет необрабатываемое исключение. Метод класса is_alive() проверяет имеет ли поток статус ‘alive’.

Другие потоки могут быть вызваны используя метод класса join(). При этом блокирует вызывающий поток до тех пор, пока поток, метод join() которого вызван, не будет завершен.

Поток имеет имя name, которое может быть установлено в конструкторе. Это имя может быть прочитано и изменено с помощью методов getName() и setName()¶

Объекты блокирования

Объекты примитивного блокирования предназначены для синхронизации потоков и могут иметь два состояния «закрытый» или «открытый». Имеется два метода для работы с такими объектами: acquire() и release(). Когда объект открыт, то метод acquire() меняет состояние на закрыто и немедленно возвращает управление. Когда объект закрыт, то метод acquire() блокирует объект пока не будет выполнен метод release() в другом потоке, который разблокирует поток, тогда метод acquire() разблокирует поток и вернет управление. Метод release() только разблокирует ожидающий поток и возвращает управление. В тех случаях, когда заблокировано в acquire() более одного потока, то метод release() разблокирует только один поток, поэтому можно создать множество различных ситуаций для организации синхронизации потоков.

Рассмотрим класс примитивной синхронизации:

class threading.Lock

метод acquire(blocking=True, timeout=-1)
метод release()

Объекты состояния

Переменные состояния всегда связаны с некоторым видом блокировки. Выше были рассмотрены примитивные виды блокировки основанные на методах acquire() и release().

Рассмотрим другие блокировки. Метод wait() блокирует поток пока другой поток не разбудит (активизирует) его вызвав методы notify() или notify_all(). Причем модно установить время ожидания.

Метод notify() активизирует поток, который ожидает некоторый объект состояния, Метод notify_all() активизирует все потоки, которые ожидает данный объект состояния.

Этот механизм является типовым для задачи разделения некоторой общей памяти между совокупностью. Например,

```
# запрос на один элемент
with cv:
  while not an_item_is_available():
# в контексте условия cv
        cv.wait() #ждать
    get_an_available_item() #получить элемент

# создание одного элемента
with cv:
    make_an_item_available()
    cv.notify()
```

В этом примере ожидание первого потока на получение элемента не ограничено по времени поскольку метод wait() вызывается в цикле. Теперь рассмотрим использование метода wait_for() который автоматически проверяет состояние при этом может быть задано время ожидания

```
# запрос на элемент
with cv:
    cv.wait_for(an_item_is_available)
    get_an_available_item()
```

Выбор между методами notify() и notify_all() зависит от числа ожидающих потоков Для работы с объектами состояния используется класс модуля threading class threading.Condition(lock=None)

Объекты семафоры

Объект семафор один из старейших объектов реализации механизмов синхронизации. Объект семафор имеет внутренний счетчик, который метод acquire() уменьшает значение на единицу, пока не достигнет 0, если значения счетчика равно 0 то метод acquire() переводит поток в ждущий

режим. Метод release() увеличивает значение счетчика на единицу, пока не будет достигнуто максимальное значение счетчика, установленное при создании семафора. Для семафора нужно использовать класс

```
class threading.Semaphore(value=1)
```

Пример использования семафора.

```
maxconnections = 5
# ...
pool_sema = BoundedSemaphore(value=maxconnections)
with pool_sema:
    conn = connectdb()
    try:
        # ... use connection ...
    finally:
        conn.close()
```

Объекты события

Объекты события предназначены для синхронизации потоков. В объекте события имеется внутренний флаг, значение которого трактуется как наступление или отсутствие некоторого события, зависящего от контекста выполнения. С эти флагом манипулируют методы set() для установки флага, метод clear() очистка флага, метод wait() ожидание наступления события. Для этого в модуле treading существует класс

```
class threading.Event
```

Объекты таймеры

Объекты таймеры предназначены для управления потоками на основе времени. Таймеры запускаются с помощью метода start() , останавливаются с помощью метода cancel().

Например:

```
def hello():
    print("hello, world")
t = Timer(30.0, hello)
```

t.start() # послег 30 секунд, "hello, world" будет напечатан

Класс таймеров в модуле treading имеет представление

```
class threading.Timer(interval,function,args=None,kwargs=None)
```

Коммуникации между процессами в сети

Для эффективного обмена данными между процессами на одном компьютере необходимо воспользоваться каналами или разделенной памятью, в остальных случаях необходимо использовать механизм сокетов, поскольку библиотеки сокетов реализованы практически на всех платформах. Ниже приведен пример класса для простейшего обмена между процессами, находящимися на разных компьютерах в сети:

```
class MySocket:
    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
```

```
                            socket.AF_INET, sock-
et.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd,
2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)
```

8 ПРОГРАММИРОВАНИЕ КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ JAVA

Крупнейший веб-сервис для хостинга IT-проектов и их совместной разработки GitHub, который еще называют «социальной сетью для разработчиков», использует методику определения популярного языка программирования. Их система под названием PYPL (PopularitY of Programming Languages) основана на количестве поисковых запросов руководств по конкретному языку программирования. Согласно рейтингу языков программирования по версии GitHub по состоянию на 2017 года на первом месте расположился язык Java как основной язык, используемый для разработки родных Android-приложений для смартфонов и планшетов (рис. 4.1).

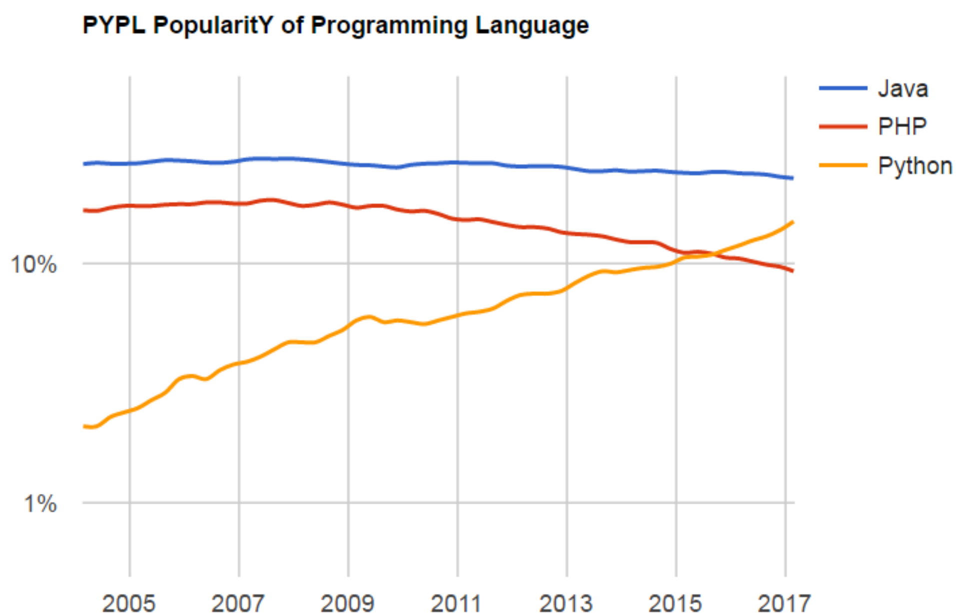

Рис. 4.1 – Рейтинг языков программирования по версии GitHub на 2017

Популярность Java у разработчиков связана с простотой и надежностью языка, который обеспечивает долгосрочную совместимость написанных на нём продуктов. Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java. Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина.

Существуют несколько основных семейств технологий Java, представленных в таблице 4.1.

Таблица 4.1 – Технологии Java

Технология	Описание
Java SE – Java Standard Edition	Основная технология Java, включающая компиляторы, API, Java Runtime Environment; используется для создания пользовательских настольных приложений (desktop).
Java EE – Java Enterprise Edition	Технология создания программного обеспечения уровня предприятия. Используется для разработки WEB-приложений.
Java ME – Java Micro Edition	Технология создания программ для устройств, ограниченных по вычислительной мощности, например, мобильных телефонов.
JavaFX	Технология создания графических интерфейсов корпоративных приложений и бизнеса.
Java Card	Технология создания программ для приложений, работающих на смарт-картах и других устройствах с очень ограниченным объёмом.

Java EE построена поверх платформы Java SE. Java Платформа EE обеспечивает API и среду выполнения для разработки и работающий крупномасштабный, многоуровневый, масштабируемый, надежный и безопасный сетевых приложений. В 2018 Eclipse Foundation переименовала Java EE в Jakarta EE – набор спецификаций и соответствующей документации для языка Java, описывающей архитектуру серверной платформы для задач средних и крупных предприятий.

Java EE API включает в себя несколько технологий, которые расширяют функциональность базовых Java SE API-интерфейсов, среди которых:

- javax.servlet – спецификация сервлетов определяет набор программных интерфейсов для обслуживания HTTP-запросов. Она включает в себя спецификации JavaServer Pages.
- javax.websocket – спецификация Java API для WebSocket определяет набор программных интерфейсов для обслуживания WebSocket-соединений.

Сервлеты

Сервлеты – это компоненты приложений Java 2 Platform Enterprise Edition (J2EE), выполняющиеся на стороне сервера, способные обрабатывать клиентские запросы и динамически генерировать ответы на них. Наибольшее распространение получили сервлеты, обрабатывающие клиентские запросы по протоколу HTTP [18].

Все сервлеты реализуют общий интерфейс Servlet. Для обработки HTTP-запросов можно воспользоваться в качестве базового класса абстрактным классом HttpServlet. Базовая часть классов JSDK помещена в пакет javax.servlet. Однако класс HttpServlet и все, что с ним связано, располагаются на один уровень ниже в пакете javax.servlet.http.

Пакет javax.servlet содержит ряд интерфейсов и классов, устанавливающих обрамление, в котором работают сервлеты (табл. 4.2).

Таблица 4.2 – Пакет javax.servlet

Интерфейс	Описание
Servlet	Объявляет методы цикла жизни для сервлета.
ServletConfig	Позволяет сервлетам получать параметры инициализации.
ServletContext	Активизирует возможности сервлетов для регистрации событий и доступа к информации об их среде.
ServletRequest	Используется для чтения данных из запроса клиента.
ServletResponse	Используется для записи данных в ответ клиенту.
SingleThreadModel	Указывает, что сервлет защищен от многопоточности.

Жизненный цикл сервлета

Жизненный цикл сервлета определяют следующие основные методы: init( ), service() и destroy(). Они реализуются каждым сервлетом и в нужный момент вызываются сервером. Существует 5 шагов:

1. Скачать класс Servlet в память.
2. Создать объект Servlet.
3. Вызвать метод init() в Servlet.
4. Вызвать метод service() вServlet.
5. Вызвать метод destroy() в Servlet.

Первым вызывается метод init(). Он дает сервлету возможность инициализировать данные и подготовиться для обработки запросов. Чаще всего в этом методе программисты помещают код, кэширующий данные фазы инициализации. После этого сервлет можно считать запущенным, он находится в ожидании запросов от клиентов.

При разработке сервлетов в качестве базового класса в большинстве случаев используют не интерфейс Servlet, а класс HttpServlet, отвечающий за обработку запросов HTTP.

Для разработки сервлетов потребуется доступ к контейнеру сервлетов или серверу приложений. Наиболее популярными из них являются сервер приложений Glassfish и контейнер сервлетов Tomcat. Сервер приложений Glassfish предоставляется компанией Oracle в комплектеJava EE SDK. Он поддерживается в IDE NetBeans. А контейнер сервлетов Tomcat – это программный продукт с открытым исходным кодом, поддерживаемый организацией Apache Software Foundation [19]. Он также поддерживается в NetBeans и Eclipse.

Пример сервлета

Прежде всего создайте файл HelloServlet.java, который будет содержать исходный код следующей программы:

```
package org.servlet;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class NewServlet extends HttpServlet {

          public void doGet(HttpServletRequest request,
          HttpServletResponse response)
          throws ServletException, IOException {
          response.setContentType("text/html");   //   Установка
типа документа
          PrintWriter out = response.getWriter();
          out.println("<html>    <body>    Hello,    world    !!!
</body></html>");
          out.close();
          }
     }
```

Чтобы начать работать с Servlet, вам необходимо скачать Tomcat Web Server и объявить его в IDE, например в Eclipse.

В **Eclipse**, выберите **Window/References** и добавить **Server/Runtime Environments** (рис. 4.2).

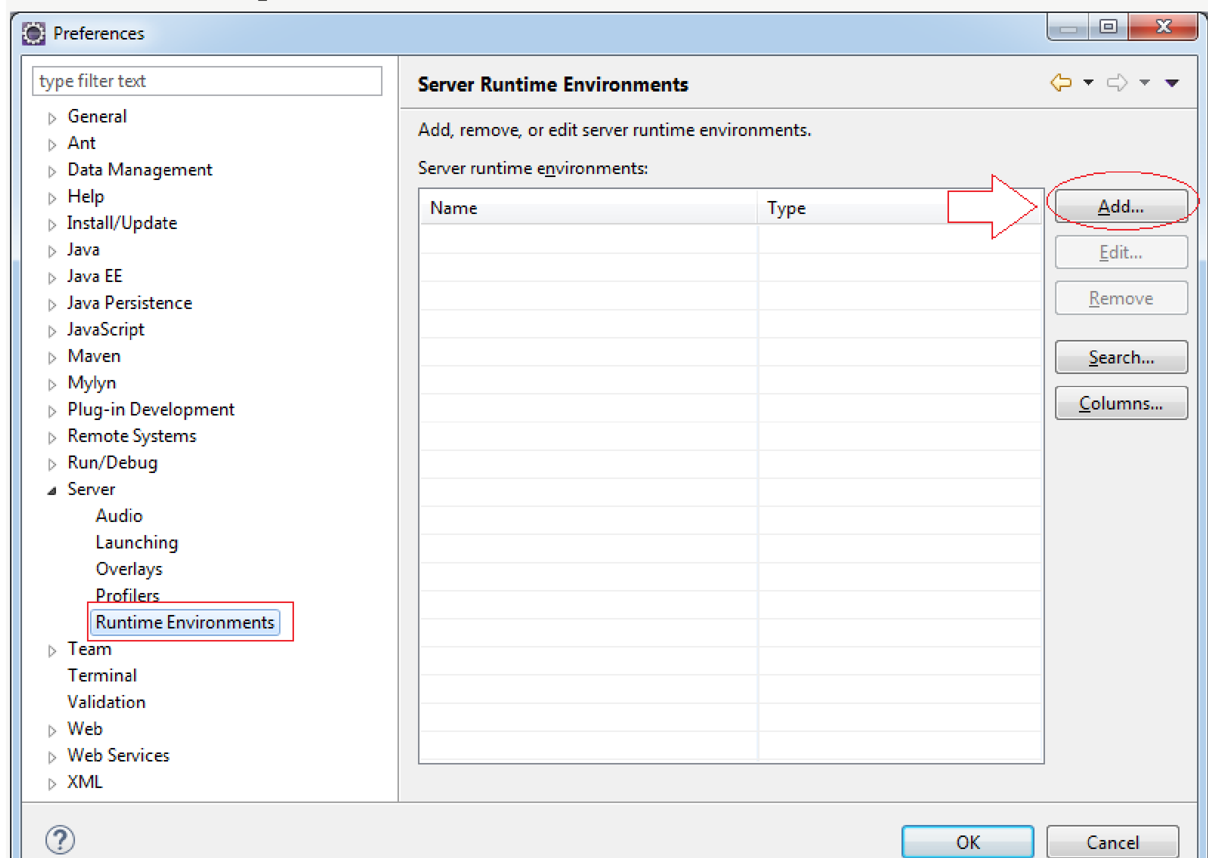

Рис. 4.2 – Объявление Tomcat в Eclipse

Далее необходимо выбрать тип **WebServer Tomcat** (рис. 4.3).

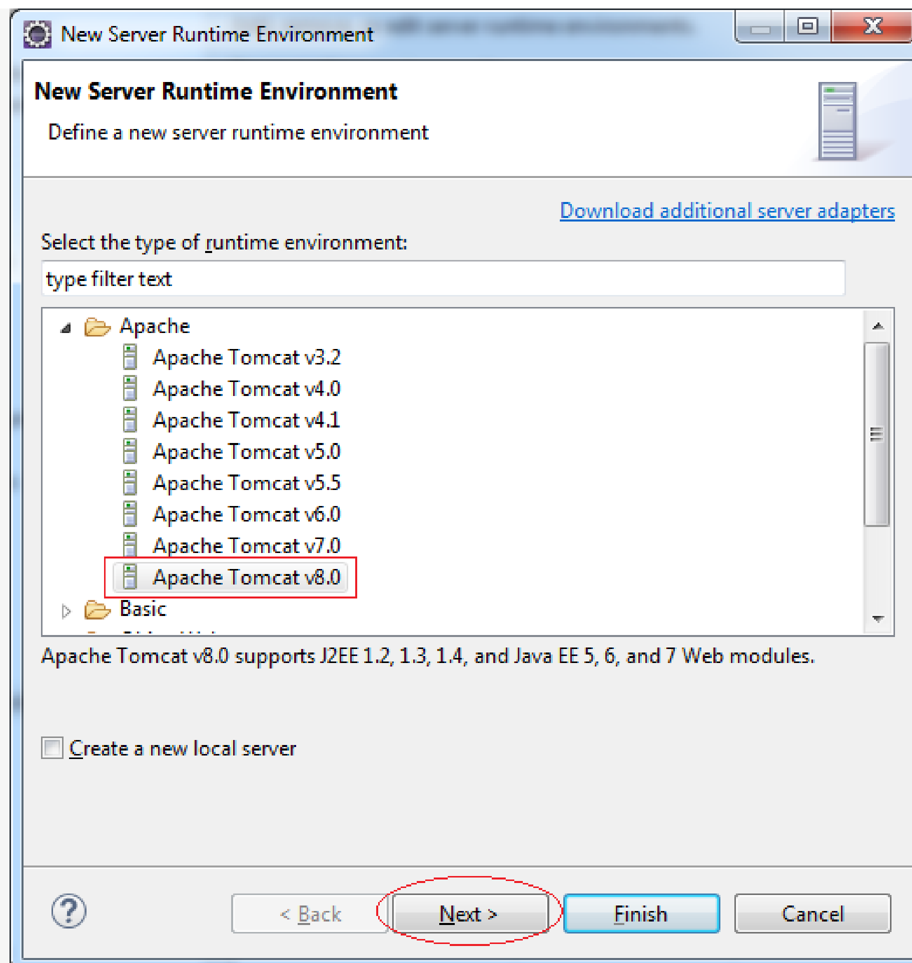

Рис. 4.3 – Указать тип **WebServer Tomcat**

Указать место **Tomcat**, куда вы извлекли на жестком диске и нажмите кнопку **OK**, чтобы закончить.

Щелкните правой кнопкой мыши на проект **ServletTutorial**, выберите " **Run As / Run on Server"**. Website работает в браузере на **Eclipse:** http://localhost:8080/Servlet/hello

В окне браузера должны появиться данные, выводимые сервлетом. Они будут состоять из символьной строки "Hello, world ! ".

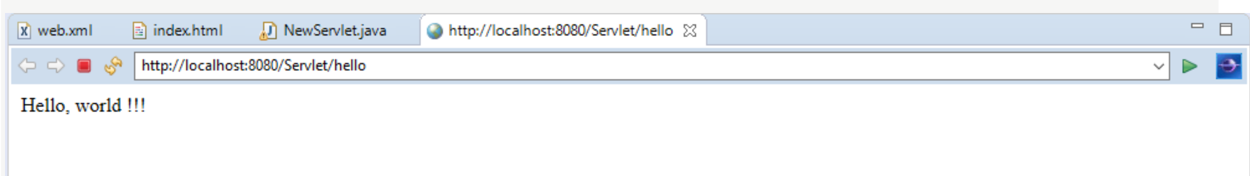

Рис. 4.4 – Результат работы сервлета

Обработка запросов

В классе HttpServlet определены методы doGet() и doPost() для реакции на запросы типа GET и POST клиента. Эти методы вызываются методом service() класса HttpServlet, который, в свою очередь, вызывается при

поступлении запроса на сервер. Метод service сначала определяет тип запроса, а затем вызывает соответствующий метод.

Методы doGet() и doPost() принимают в качестве параметров объекты HttpServletRequest и HttpServletResponse, которые дают возможность осуществлять взаимодействие между клиентом и сервером. Методы интерфейса HttpServletRequest облегчают доступ к данным запроса. Методы интерфейса HttpServletResponse облегчают возврат результатов Web-клиенту в виде HTML.

Рассмотрим пример:

```
package org.servlet;
import java.io.*;

import javax.servlet.ServletException;
import javax.servlet.ServletOutputStream;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class NewServlet extends HttpServlet {

     /** Метод doGet служит для обработки GET-запросов */
      public   void   doGet(HttpServletRequest   request,   HttpS-
ervletResponse
     response) throws ServletException, IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
     // Генерируем форму
      out.println("<FORM method=\"POST\">");
      out.println("<INPUT type=\"text\" name=\"welcome\">");
      out.println("<input type=\"submit\"></FORM>");
      }
      public void doPost(HttpServletRequest request,
               HttpServletResponse response) throws
               ServletException, IOException {
                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
               // Получаем параметр запроса
               String          welcome          =          re-
quest.getParameter("welcome");
                out.println("Hello, "+ welcome);
               }
}
```

Скомпилируем данный сервлет (рис. 4.5).

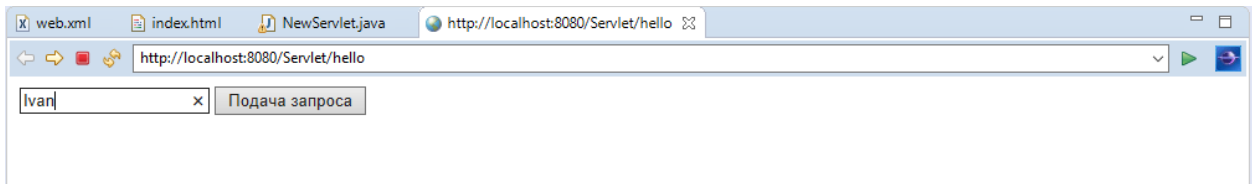

Рис. 4.5 – Результат работы сервлета

Передадим форму, заполненную на веб-странице. После этого браузер выведет результат, динамически сформированный сервлетом (рис. 4.6).

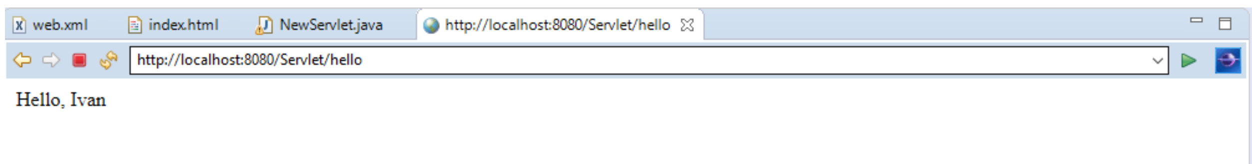

Рис. 4.6 – Результат запроса, посылаемый серверу из браузера

ЗАКЛЮЧЕНИЕ

Учебное пособие «Разработка сетевых приложений» позволяет изучить основы разработки программ, базирующихся на клиент/серверной модели.

Последовательно изложены следующие темы:

1. Логические основы построения сетей.
2. Инструменты и механизмы создания сетевых программ, основанных на использовании сокетов, RPC, именованных каналов (pipes).
3. Организация многопоточных приложений и разнообразные объекты и алгоритмы синхронизации потоков.
4. Механизмы обмена информацией между процессами.
5. Принципы, организация и использование программных интерфейсов CGI и ISAPI для расширения функциональных возможностей WWW-сервера.
6. Рассмотрено большое число примеров.

Дальнейшее изучение данной дисциплины связано с технологиями создания сетевых приложений, таких как CORBA, DCOM, NET, SOAP, DELPHI, разработанных конкретными фирмами.

ЛИТЕРАТУРА

1. Эммерих В. Конструирование распределенных объектов. – М.: Мир, 2002. – 510 с.

2. Рихтер Дж. Windows для профессионалов: создание эффективных приложений с учетом специфики 64-разрядной версии Windows. – СПб.: Питер, 2001. – 752 с.

3. Вильямс А. Системное программирование в Windows 2000 для профессионалов – СПб.: Питер. – 624 с.

4. Семенов Ю.А. Сети Интернет. Архитектура и протоколы. – М.: Блик плюс, 1998. – 424 с.

5. Сван, Том. Программирование для Windows в Borland C++. – М.: БИНОМ, 1995. – 480 с.: ил.

6. Петзолд Ч. Программирование для Windows 95: Все секреты программирования для Windows 95. Т.2. // Мастер: Руководство для профессионалов. – СПб.: Изд-во BHV, 1997. – 368 с.: ил.

7. Фролов А.В., Фролов Г.В. Microsoft Visual C++ и MFC: Программирование для Windows 95 и Windows NT. Ч.1 // Б-ка системного программиста. – М.: ДИАЛОГ-МИФИ, 1995. – 288 с.

8. Фролов А.В., Фролов Г.В. Microsoft Visual C++ и MFC: Программирование для Windows 95 и Windows NT. Ч.2 // Б-ка системного программиста. – М. : ДИАЛОГ-МИФИ, 1995. – 272 с.

9. Фролов А.В., Фролов Г.В. Графический интерфейс GDI в MS Windows // Библиотека системного программиста. М.: ДИАЛОГ-МИФИ, 1994. – 288 с.: ил.

10. Фролов А.В, Фролов Г.В. Мультимедиа для Windows // Библиотека системного программиста. – М.: ДИАЛОГ-МИФИ, 1994. – 284 с.: ил.

11. Фролов А.В., Фролов Г.В. Операционная система Windows 95 // Б-ка системного программиста. – М.: ДИАЛОГ-МИФИ, 1996. – 288 с.: ил.

12. Петзолд Ч. Программирование для Windows 95: Все секреты программирования для Windows 95. Т.1 // Мастер: Руководство для профессионалов. – СПб.: Изд-во BHV, 1997. – 752 с.: ил.

13. Сидни Фейт TCP/IP: Архитектура, протоколы, реализации. – М.: Изд-во «Лори», 2000. – 424 с.

14. Википедия. – URL : www.wikipedia.org (дата обращения: 13.06.2018).

15. Фертиков В.В. Распределенные вычисления: технология Microsoft RPC: Учебное пособие. – Воронеж: Воронежский государственный университет, 2005. – 31 с.

16. Язык программирования Python : учеб. пособие / Р. А. Сузи . – 2 изд., испр . – М. : Интернет-Университет информационных технологий : Бином. Лаборатория знаний, 2007. – 326 с. : ил. – (Основы информационных технологий).

17. Python Tutorial [Электронный ресурс]. – Режим доступа: https://docs.python.org/3/tutorial/index.html] (дата обращения: 4.06.2018).

18. Блинов, И. Н. Java 2: практ. рук. / И.Н. Блинов, В.С. Романчик. – Мн.: УниверсалПресс, 2005. – 403 с.

19. Tomcat Server. – URL : http://tomcat.apache.org/download-80.cgi (дата обращения: 13.06.2018).