МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Кафедра электронных вычислительных машин

Д.А. Пынькин, И.И. Глецевич

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

по дисциплине
«Системное программное обеспечение
локальных вычислительных сетей»
для студентов специальности 40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

МИНСК 2005

УДК 681.3.068(075.8)
ББК 32.973.26-018.2 я73
К00

Авторы: Д.А. Пынькин, И.И. Глецевич

К00 Лабораторный практикум по дисциплине «Системное программное обеспечение локальных вычислительных сетей» для студентов специальности 40 02 01 «Вычислительные машины, системы и сети» всех форм обучения / Д.А. Пынькин, И.И. Глецевич. – Мн.: БГУИР, 2005. – 00 с.; ил. 0.

ISBN 985-444-206-3

В настоящем лабораторном практикуме рассматриваются теоретические и практические основы архитектуры приложений, взаимодействующих между собой в локальных вычислительных сетях. Особое внимание уделяется построению распределенных систем. Все примеры и задания рассчитаны на выполнение в операционной системе Linux.

УДК 681.3.068(075.8)
ББК 32.973.26-018.2 я73

ISBN 985-444-206-3

Св. план 2005, поз. 37

Учебное издание

Авторы: Пынькин Денис Александрович
Глецевич Иван Иванович

ЛАБОРАТОРНЫЙ ПРАКТИКУМ
по дисциплине
«Системное программное обеспечение
локальных вычислительных сетей»
для студентов специальности 40 02 01
«Вычислительные машины, системы и сети»
всех форм обучения

Редактор Н.А. Бебель

Подписано в печать — Формат 60x84 1/16
Бумага — Печать офсетная — Усл. печ. л.
Уч.-изд. л. 4,8 — Тираж 100 экз. — Заказ

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»
Отпечатано в БГУИР. Лицензия ЛП № 156. 220013, Минск, П. Бровки, 6

СОДЕРЖАНИЕ

Лабораторная работа №1
АРХИТЕКТУРА КЛИЕНТ-СЕРВЕР

Цель: Изучить аспекты проектирования и реализации сетевых приложений с использованием архитектуры клиент-сервер.
Длительность: 4 часа.

1.1Теоретическая часть

1.1.1Классическая клиент-серверная модель

В базовой модели клиент-сервер все процессы делятся на 2 возможно перекрывающиеся группы. Процессы, реализующие некоторую службу, например файловую систему или базу данных называются *серверами*. Процессы, запрашивающие службы у серверов путем пересылки запроса и последующего ожидания ответа от сервера, называются *клиентами*. Взаимодействие клиента и сервера, называемого также режим работы *запрос-ответ*, показан на рисунке 1.1.

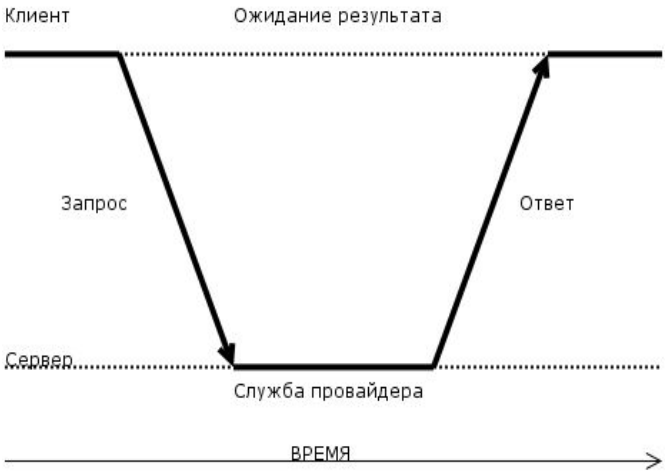


Рис.1.1. Классическая модель клиент-сервер

Использование не требующего соединения протокола дает нам выигрыш по используемым ресурсам, однако создание надежного протокола не требующего соединений – тяжелая задача. Поэтому практически всеми программами, требующими надежности, используется надежный протокол с установлением соединений.

1.1.2Разделение приложений по уровням

Модель клиент-сервер всегда была предметом множества дебатов и споров. Один из главных вопросов состоял в том, как точно разделить клиента и сервер. Четкого разделения здесь не может быть. Однако часто рекомендуют разделять их на 3 уровня:

- уровень пользовательского интерфейса;

- уровень обработки;
- уровень данных.

Уровень пользовательского интерфейса содержит все необходимое для непосредственного общения с пользователем (например управление клавиатурой, дисплеем). Уровень обработки обычно содержит приложения, а уровень данных – собственно данные, с которыми происходит работа.

Уровень пользовательского интерфейса

Уровень пользовательского интерфейса обычно реализуется на клиентах. Этот уровень содержит программы, посредством которых пользователь может взаимодействовать с приложением.

Простейший вариант пользовательского интерфейса не содержит ничего, кроме символьного (или графического) дисплея. Современные пользовательские интерфейсы поддерживают совместную работу приложений через единственное графическое окно и, в ходе действий пользователя, обеспечивают через это окно обмен данными.

Уровень обработки

На этом уровне трудно выделить какие-то закономерности, обычно здесь реализуется логика работы программы.

Чтобы пояснить задачи этого уровня, в качестве примера приведем обычный запрос к поисковой машине в Интернете. Если отбросить различные баннеры, картинки и тому подобные, не относящиеся к запросу украшательства, то в итоге пользовательский интерфейс сократиться до поля ввода запроса, а в ответе будет выводиться список заголовков найденных страниц.

Ядро поисковой машины логически находится на уровне обработки. Это ядро занимается генерированием запросов к базе данных проиндексированных web-страниц и формированием результатов поиска.

Уровень данных

На этом уровне находятся программы, которые поставляют данные обрабатывающим их приложениям. Специфическим требованием этого уровня является требование сохранности – это означает, что когда приложение не работает, данные должны сохраняться в определенном месте в расчете на дальнейшее использование. В простейшем варианте уровень данных реализуется файловой системой, однако часто может использоваться и полнофункциональная база данных. В модели клиент-сервер этот уровень обычно находится на стороне сервера.

1.1.3Варианты архитектуры клиент-сервер

Существует несколько вариантов физического разделения такой трехзвенной архитектуры. Простейшей является взаимодействие только 2-х типов машин:

- клиентские машины, на которых имеются программы, реализующие только пользовательский интерфейс или его часть;

– серверы, реализующие все остальное.

Фактически, при такой архитектуре клиент выступает в роли терминала, а уровень обработки и уровень данных реализуется на сервере.

Один из подходов к организации клиентов и серверов – это распределение программ, находящихся на уровне обработки данных, на различные машины, как показано на рисунке 1.2.

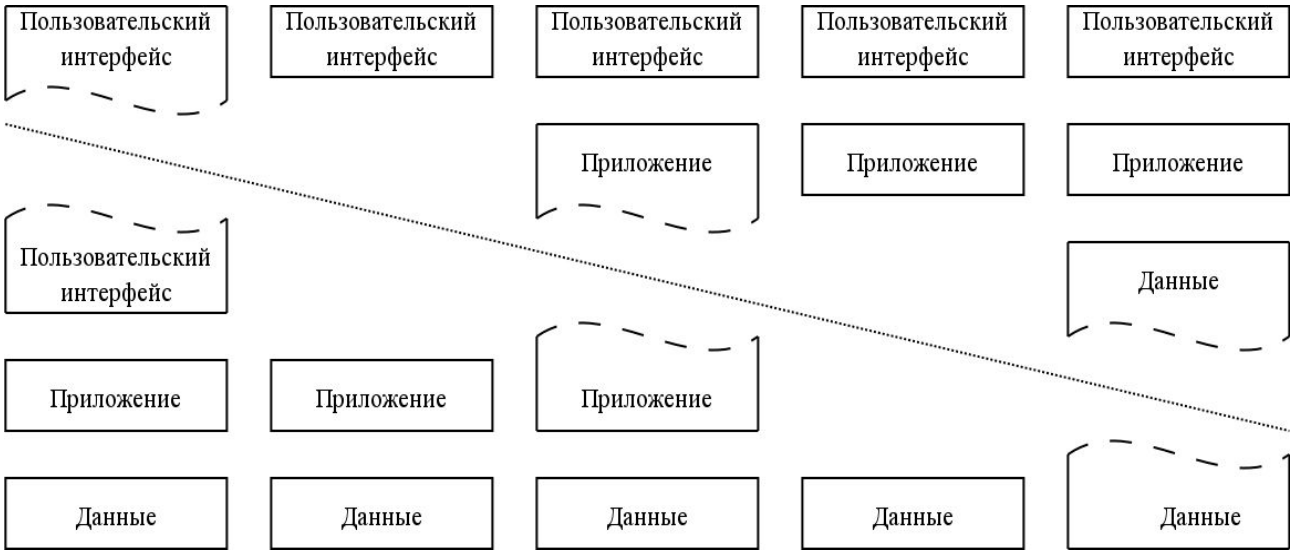


Рис. 1.2. Альтернативные формы организации архитектуры клиент-сервер

Это называется физически двухзвенная архитектура. Однако иногда может понадобиться, чтобы серверы работали и как клиенты, что приводит нас к физически трехзвенной архитектуре изображенной на рисунке 1.3. При этом сервер приложений работает как клиент, с точки зрения сервера базы данных.

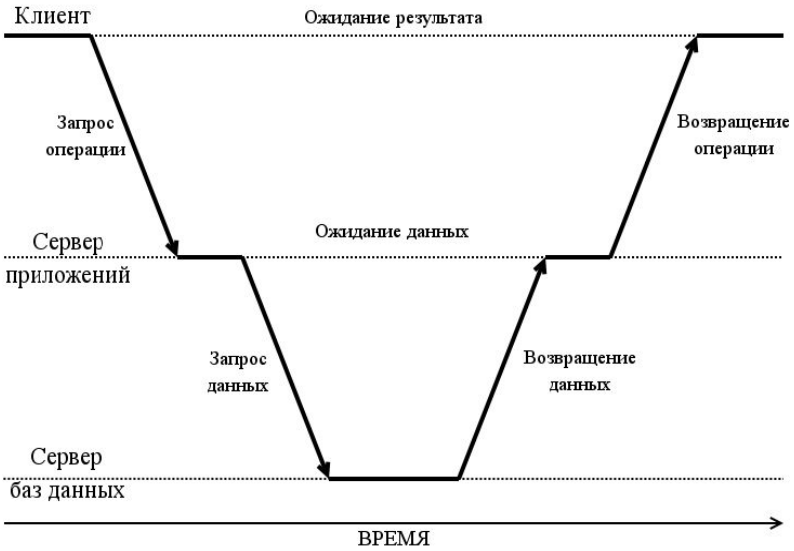


Рис. 1.3. Трехзвенная архитектура клиент-сервер

Современные варианты архитектуры

Во множестве приложений обработка данных организована как многозвенная архитектура приложений клиент-сервер. Такой тип

распределения называется *вертикальным распределением*. Особенностью такого типа является то, что оно достигается размещением логически разных компонентов на различных физических машинах.

Существует также *горизонтальное распределение*. При таком типе распределения клиент или сервер может содержать физически разделенные части логически однородного модуля, причем работа с каждой из частей может протекать независимо. Это делается для выравнивания нагрузки.

Для некоторых несложных приложений сервера может не быть вообще. Такую организацию обычно называют *одноранговым распределением* (peer-to-peer).

1.2Архитектура клиента

Приложения, действующие в качестве клиентов, концептуально проще приложений, выполняющих функции серверов, по нескольким причинам. Во-первых, в основной части клиентского программного обеспечения явно не предусмотрено обеспечение параллельной работы с несколькими серверами. Во-вторых, основная часть клиентского программного обеспечения применяется в виде обычных прикладных программ. В отличие от серверного, для клиентского программного обеспечения обычно не требуются специальные привелегии, поскольку оно, как правило, не обращается к привилегированным портам протокола. В-третьих, большая часть клиентского обеспечения не должна заботиться о правилах защиты, поскольку в этом вопросе оно полагается на операционную систему.

1.2.1Определение местонахождения сервера

В клиентском программном обеспечении может использоваться один из методов для определения местонахождения сервера:

- доменное имя или IP-адрес сервера могут быть заданы в виде константы во время трансляции клиентской программы;
- клиентская программа может требовать у пользователя указывать имя сервера при ее вызове;
- информация о местонахождении сервера предоставляется из постоянного хранилища данных (файл, база данных);
- для поиска сервера используется отдельный протокол.

Если адрес сервера задан в виде константы, клиентское программное обеспечение работает быстрее и в меньшей степени зависит от конкретной локальной вычислительной сети. Из минусов следует указать, что клиентская программа должна быть перекомпилирована при изменении местонахождения сервера, а также то, что клиент не может быть переподключен к другому серверу. В качестве компромисса можно использовать символьное имя вместо IP-адреса.

При использовании файла конфигурации клиент становится более гибким, но зависит от наличия этого файла в системе.

Широковещательный запрос хорошо действует в небольшой сети, однако в больших сетях возникают проблемы при работе. Кроме того динамический поиск создает сложности при проектировке алгоритмов клиента и сервера.

Если пользователю предоставляется возможность указать адрес сервера при вызове клиентского ПО, то клиентская программа становится более универсальной и появляется возможность менять местонахождение сервера. Также появляется возможность использовать и другие методы поиска сервера.

Еще один аспект поиска сервера следует из того, какой тип службы предоставляется сервером. От этого зависит нужно ли использовать какой-либо определенный сервер или же можно использовать первый ответивший. В качестве альтернативы, в ответе сервера может также содержаться список других серверов.

1.2.2Алгоритм клиентов с установлением логического соединения

Задача построения клиента с использованием протокола TCP является самой простой из всех задач сетевого программирования.

Алгоритм клиента TCP:

1. Найти IP-адрес и номер порта протокола сервера, с которым необходимо установить связь.
2. Распределить сокет.
3. Указать, что для соединения нужен произвольный, неиспользуемый порт протокола на локальном компьютере и позволить ПО TCP выбрать такой порт.
4. Подключить сокет к серверу.
5. Выполнить обмен данными с сервером по протоколу прикладного уровня.
6. Закрыть соединение.

Анализ параметра адреса

Обычно в качестве имени сервера клиент позволяет задавать или адрес или доменное имя. Для определения того, что было задано, программа проверяет, содержит ли параметр алфавитные символы. Если да, то считается, что задано имя, если же только цифры и точки – то IP-адрес. Иногда может понадобится задавать не только адрес сервера, но и указывать порт протокола.

Хотя каждая клиентская программа может указывать адрес и порт по-своему, существует общепринятый синтаксис этих параметров:

- сначала записывается имя, а вторым параметром – порт;
- имя и порт рассматриваются как один параметр, разделенный двоеточием.

Чтобы преобразовать символьное имя в IP-адрес можно воспользоваться функцией `gethostbyname`, а чтобы получить номер порта по имени – `getservbyname`. Наряду с этим есть функция для преобразования символьного имени протокола в константу, соответствующую этому протоколу: `getprotobyname`.

Адрес сервера в клиентской программе указывается в структуре `sockaddr_in`, однако пользователь задает его либо как IP-адрес, заданный цифрами через точку, либо как символьное доменное имя. Соответственно в клиентской программе необходимо преобразовать адрес заданный

пользователем в двоичный адрес, используемый компьютером.

Вызов `socket` создает оконечную точку для коммуникации и возвращает её дескриптор:

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

Ниже показан пример реализации подпрограммы, которая создает сокет, используя адрес и порт, заданные пользователем в текстовой форме.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>

char * host = "localhost";
char * service = "imap";
char * proto = "tcp";
struct sockaddr_in sin;

int mksock( char *host, char * service, char * proto, struct
sockaddr_in *sin)
{
struct hostent *hptr;
struct servent *sptr;
struct protoent *pptr;
int sd=0, type;

    memset( sin, 0, sizeof( *sin));
    sin->sin_family = AF_INET;

    if( hptr = gethostbyname( host))
     memcpy( & sin->sin_addr, hptr->h_addr, hptr->h_length);
    else
     return -1;

    if ( ! ( pptr = getprotobyname( proto)) )
     return -1;

    if( sptr = getservbyname( service, proto))
     sin->sin_port = sptr->s_port;
    else if( (sin->sin_port =
               htons(( unsigned short)  atoi (service))) == 0)
         return -1;

    if ( strcmp( proto, "udp") == 0)
     type = SOCK_DGRAM;
    else
     type = SOCK_STREAM;

    if ( (sd = socket( PF_INET, type, pptr->p_proto)) < 0)
```

```
    {
     perror( "Ошибка при распределении сокета");
     return -1;
    }

    return sd;
}
```

Для большей части клиентов нет нужды привязывать свою программу к определенному сетевому интерфейсу, это должно происходить автоматически, однако, если возникает необходимость самостоятельно указывать адрес привязки, то нужно учитывать маршрутизацию на локальной машине.

При выборе номера порта, клиент, в отличие от сервера, не должен использовать общепринятые адреса различных служб. Кроме того, должен использоваться еще незанятый порт. При вызове функции `connect` такой порт может выбираться автоматически.

Подключение сокета к TCP серверу

Производится с помощью вызова `connect` . Этот вызов выполняет 4 задачи:

- проверяет, является ли сокет действительным и не был ли он подключен;
- заполняет поле адреса конечной точки в дескрипторе сокета (из 2-го параметра);
- выбирает локальный адрес в дескрипторе сокета, если он еще не задан;
- инициирует соединение TCP и возвращает результат в вызывающую программу.

Вызов `connect` инициирует соединение локального сокета с удаленным :

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

Взаимодействие с сервером при использовании TCP

Если соединение установлено, то обычно прикладной протокол определяет взаимодействие по принципу запрос-ответ. Для этого используется вызовы `send` и `recv` (либо стандартные функции для работы с дескрипторами файлов: `write` и `read`):

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

int recv(int s, void *buf, size_t len, int flags).
```

Поскольку протокол TCP является потоковым, то он не гарантирует доставку в виде тех же фрагментов, которые были отправлены. Программное обеспечение протокола TCP может разбить любой блок данных на части и передавать каждую часть в отдельном сегменте, либо собрать несколько блоков в один сегмент. Поэтому в основе разработки программ, использующих протокол TCP лежит следующий принцип: поскольку программное обеспечение протокола TCP не учитывает границ между записями, то в любой

программе, принимающей данные из соединения TCP, необходимо предусмотреть возможность получения данных в виде фрагментов, составляющих лишь несколько байтов.

Закрытие соединения TCP

Для корректного завершения соединения и освобождения сокета вызывается функция закрытия дескриптора `close`. Однако часто возникает ситуация, когда сервер не знает будут ли еще приходить запросы от клиента и наоборот, когда клиент не знает все ли данные выдал сервер.

Для обхода этой проблемы во многих реализациях интерфейса сокетов реализована функция `shutdown`:

```
int shutdown ( int sd, int how)
```

где `how` принимает значения `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR` (`0`, `1`, `2` соответственно, однако рекомендовано использовать символьные константы), для закрытия сокета в одном из направлений. Для слушающего сокет это выглядит как конец файла.

Механизм частичного закрытия позволяет устранить неопределенность в работе прикладных протоколов, которые передают произвольный объем информации в ответ на запрос. В таких случаях клиент выполняет операцию частичного закрытия после передачи последнего запроса; затем сервер закрывает соединение после передачи последнего ответа.

1.2.3Алгоритм клиентов без установления логического соединения

Алгоритм во многом напоминает алгоритм клиента TCP. В этом алгоритме проблема надежности игнорируется.

Алгоритм клиента UDP:

1. Найти IP-адрес и номер порта протокола сервера, с которым необходимо установить связь.
2. Распределить сокет.
3. Указать, что для соединения нужен произвольный, неиспользуемый порт протокола на локальном компьютере и позволить программному обеспечению UDP выбрать такой порт.
4. Указать сервер, на который должны передаваться сообщения.
5. Выполнить обмен данными с сервером по протоколу прикладного уровня.
6. Закрыть сокет.

В клиентском приложении сокет UDP может использоваться в одном из двух основных режимов: подключенном и неподключенном. Подключенные сокеты удобно применять для взаимодействия с конкретным сервером, а неподключенные – если адресат может меняться, в этом случае нужно для каждого сообщения указывать адрес сервера.

Если используется подключение, то вызов функции connect только записывает информацию об удаленной точке в дескриптор сокета. Даже если

вызов успешен, это еще не означает, что адрес удаленной точки действителен или что сервер является достижимым.

Закрытие соединения UDP

Применение функции `close` приводит к тому, что программномое обеспечение UDP будет отбрасывать все дальнейшие пакеты. При этом удаленная сторона не информируется о закрытии соединения, поэтому приложения нужно проектировать таким образом, чтобы удаленный участник знал, как долго сокет должен оставаться доступным до его закрытия.

Вызов функции `shutdown` не отправляет каких-либо сообщений удаленной стороне и сокет просто помечается, как неприменимый для передачи данных в указанном направлении.

Ненадежность UDP

Клиентское ПО, в котором используется протокол UDP, должно обеспечивать надежность с помощью различных методов, таких как контроль за последовательностью поступления пакетов, подтверждения, тайм-ауты и повторная передача. Проектирование правильных, надежных и эффективных протоколов для больших сетей требуют опыта.

1.3Архитектура сервера

Концептуальный алгоритм сервера

По сути каждый сервер функционирует по следующему алгоритму: в нем создается сокет и выполняется его привязка сокета. Затем сервер входит в бесконечный цикл, в котором он принимает очередной запрос, поступающий от клиента, обрабатывает этот запрос, формирует ответ и отправляет его клиенту.

Однако на практике такой алгоритм применяется редко, только для создания самых простейших служб.

1.3.1Типы серверов

По типу внутренней архитектуры сервера можно разделить на 2 больших класса: последовательные и параллельные. Сервер называется *последовательным*, если он выполняет обработку запросов друг за другом, *параллельным* – если обеспечивает одновременное выполнение нескольких запросов (не обязательно используется несколько потоков выполнения).

Последовательные реализации сервера, которые являются более простыми для разработки, могут привести к снижению производительности, поскольку клиентам приходится ожидать доступа к службе. Параллельные же обеспечивают лучшую производительность, хотя и являются более сложными для проектирования и разработки.

По типу используемого протокола разделяются на сервера с установлением логического соединения (TCP в стеке TCP/IP) и сервера без установления логического соединения (UDP в стеке TCP/IP).

Поскольку протоколы TCP и UDP по своим хар-кам резко отличаются друг от друга, то выбор нужного протокола зависит от требований прикладного протокола.

Серверы с установлением логического соединения

Основным преимуществом использования протокола с уст. лог. соед. является простота программирования. Поскольку такой транспортный протокол самостоятельно решает проблемы потери пакетов и их доставки с нарушением исходного порядка, в серверной программе можно не заниматься решением таких проблем. Поэтому в сервере достаточно реализовать только функции управления и использования соединений.

Из недостатков следует отметить, что для каждого логического соединения требуется создавать отдельный сокет, соответственно появляются издержки на распределение сокета и управление соединением. Второй недостаток следует из архитектуры протокола TCP – требуется трехэтапное квитирование при установке и разрыве соединения, что невыгодно для использования передачи небольших объемов данных в небольшой сети. В качестве альтернативного выхода – можно использовать протокол T/TCP (Transaction TCP).

Наиболее важный недостаток заключается в том, что простаивающие соединения, по которым не проходят пакеты, напрасно используют ресурсы.

Серверы без установления логического соединения

При использовании протокола UDP, сервер не может полагаться на транспортный протокол для получения надежной связи. Поэтому один или оба участника соединения должны взять на себя эту заботу. Обычно функции по обеспечению повторной передачи запросов, при неполучении ответов, возлагаются на клиентов. Если же в сервере необходимо разбивать ответы на несколько пакетов, то в нем может также понадобиться реализовать механизм повторной передачи (квитирование, тайм-ауты). Часто этот механизм тестируется в локальной сети и при переносе в глобальную перестает работать.

Еще одним плюсом протокола без установления соединения является то, что с его помощью можно реализовывать широковещательные или групповые рассылки.

Таким образом типы серверов можно представить в виде таблицы 1.1.

Краткая характеристика типов серверов:

- Последовательный сервер без установления логического соединения Используется в службах, требующих незначительного времени для обработки каждого запроса.
- Последовательный сервер с установлением логического соединения Используется в службах, требующих незначительного времени для обработки каждого запроса, однако требующих надежного протокола доставки сообщений. За счет больших издержек на установку и завершение соединения среднее время отклика часто значительно выше, чем у предыдущего сервера.
- Параллельный сервер без установления логического соединения

Довольно редко применяемый тип сервера. Во многих случаях затраты на создание потоков или процессов не оправдывают повышения эффективности, достигнутого за счет параллелизма.

- Параллельный сервер с установлением логического соединения
  Наиболее распространенный тип сервера, поскольку сочетает надежный протокол (подходит и для глобальных сетей) с возможностью одновременной работы с несколькими клиентами. Существует 3 типа реализации параллелизма работы: с помощью потоков, процессов и асинхронного ввода/вывода.
  При использовании потоков или процессов один из них является ведущим, остальные – ведомыми.

Таблица 1.1. Типы серверов

Последовательный без установления логического соединения	Последовательный с установлением логического соединения
Параллельный без установления логического соединения	Параллельный с установлением логического соединения

1.3.2Алгоритмы серверов

Последовательный сервер с установлением логического соединения:

1. Создать сокет и привязать его к общепринятому адресу службы.
2. Перевести сокет в пассивный режим.
3. Принять из сокета запрос на установление соединения и получить новый сокет для установления соединения.
4. Считывать в цикле запросы от клиента, формировать ответы и отправлять их клиенту, в соответствии с прикладным протоколом.
5. После завершения обмена данными с клиентом закрыть соединение и перейти к 3 пункту.

Последовательный сервер без установления логического соединения:

1. Создать сокет и привязать его к общепринятому адресу службы.
2. Считывать в цикле запросы от клиента, формировать ответы и отправлять их клиенту, в соответствии с прикладным протоколом.

Параллельный сервер без установления логического соединения:

1. Ведущий поток. Создать сокет и привязать его к общепринятому адресу службы.
2. Ведущий поток. В цикле считывать запросы с помощью `recvfrom` и создавать новые ведомые потоки (процессы) для формирования ответа.

1. Ведомый поток. Работа потока начинается с получения конкретного запроса от ведущего потока, а также доступа к сокету.
2. Ведомый поток. Сформировать ответ согласно прикладному протоколу и отправить его клиенту с использованием функции `sendto`.
3. Ведомый поток. Завершить работу потока.

Параллельный сервер с установлением логического соединения:

1. Ведущий поток. Создать сокет и привязать его к общепринятому адресу службы.
2. Ведущий поток. Перевести сокет в пассивный режим.
3. Ведущий поток. Вызвать в цикле функцию `accept` для получения очередного запроса от клиента и создавть новый ведомый поток или процесс для формирования ответа.
1. Ведомый поток. Работа начинается с получения доступа к соединению, полученному от ведущего потока.
2. Ведомый поток. Выполнять в цикле работу с клиентом через соединение.
3. Ведомый поток. Закрыть соединение и завершить работу.

Псевдопараллельный сервер с установлением логического соединения:

1. Создать сокет и привязать его к общепринятому адресу службы. Добавить сокет к списку сокетов, через которые может осуществляться ввод/вывод.
2. Использовать функцию `select` для получения информации о готовности существующих сокетов к вводу/выводу
```
int select(int n, fd_set *readfds, fd_set *writefds,
          fd_set *exceptfds, struct timeval *timeout)
```
3. Если первоначальный сокет готов, то использовать функцию `accept` для получения очередного запроса на установление соединения и добавить новый сокет к списку сокетов, через которые может осуществляться ввод/вывод.
4. Если готов сокет, отличный от первоначального, использовать функцию `recv` для получения очередного запроса, сформировать ответ и передать ответ клиенту с использованием функции `send`.
5. Перейти к пункту 2.

Создание сокета для сервера не отличается от создания сокета для клиента, поэтому может использоваться та же функция. Для привязки сокета к порту (именованию сокета) используется функция `bind`:

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

В качестве примера, ниже приведена подпрограмма для привязки и перевода сокета в пассивный режим:

```
int servsock(  char *host, char * service, char * proto, struct
sockaddr_in *sin)
{
int sd;
    if ( (sd = mksock( host, service, proto,
                    (struct sockaddr_in *) sin)) == -1)
     return -1;
    if( bind( sd, (struct sockaddr *) sin, sizeof( *sin)) < 0)
    {
     perror( "Ошибка при привязке сокета");
     return -1;
    }
    if( strcmp( proto, "tcp") == 0)
    {
     if ( listen( sd, qlen) == -1)
     {
         perror( "Ошибка при переводе сокета в пассивный режим");
         return -1;
     }
    }
    return sd;
}
```

Однако следует учесть, что адрес должен принадлежать одному из интерфейсов компьютера, либо быть равным 0.0.0.0, что соответствует `INADDR_ANY`, если сервер должен привязываться ко всем интерфейсам.

Если сервер использует протокол TCP, то для перевода в пассивный режим применяется функция `listen`:

```
int listen(int s, int backlog)
```

где также указывается длина очереди запросов на установление соединения, а для получения следующего входящего соединения – `accept`:

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

1.3.3 Методы улучшения функционирования серверов

- функционирование в фоновом режиме: с помощью `fork` запускаем копию программы, а родительский процесс принудительно завершается. При этом должны закрываться все унаследованные дескрипторы файлов;
- сервер должен отключать управляющий терминал, чтобы не получать от него сигналы:
  ```
  fd=open("/dev/tty", O_RDWR);
  ioctl( fd, TIOCNOTTY, 0);
  close( fd);
  ```
- сервер должен переходить в безопасный каталог используя `chdir`;
- необходимо изменить маску по умолчанию для создания файлов используя `umask( 027);`
- желательно изменить группу процесса `setpgrp( 0, getpid())`, чтобы не

получать сигналы, предназначенные для родительской группы;

- необходимо открыть стандартные дескрипторы для корректной работы библиотечных процедур:

```
fd=open(“/dev/null”, O_RDWR); //ввод
dup( fd); //вывод
dup( fd); //ошибки;
```

- сервер должен предотвращать запуск нескольких копий (lock-файлы или другие системные функции);
- желательно игнорировать сигналы, не относящиеся к работе сервера;
- если есть возможность, то желательно использовать системные методы ведения журнала событий.

1.3.4Примеры реализации серверов

Пример реализации последовательного сервера без установления логического соединения:

```
main( void)
{
char * host = "amok.home";
char * service = "2525";
char * proto = "udp";
struct sockaddr_in sin, remote;
struct timeval timev;
int sd, rlen, readed;
char buf[513], *t_now;

    if ( (sd = servsock( host, service, proto,  &sin, 10)) == -1)
    {
     printf( "Ошибка при создании сокета\n");
     return 1;
    }

    while(1)
    {
     rlen = sizeof( remote);
     if( (readed = recvfrom( sd, buf, 512, 0,
                    (struct sockaddr *)&remote, &rlen)) != -1)
     {
         gettimeofday( &timev, NULL);
         t_now = ctime( &(timev.tv_sec));
         sendto(sd, t_now, strlen(t_now), 0,
                    (struct sockaddr *)&remote, sizeof (remote));
     }
    }
    return 0;
}
```

Пример реализации последовательного сервера с установлением логического соединения:

```
main( void)
{
...
char * proto = "tcp";
int sd, rsd, rlen, readed;
...

    while(1)
    {
     rlen = sizeof( remote);
     rsd = accept( sd, (struct sockaddr *)&remote, &rlen);
     if( (readed = recv(rsd, buf, 512,0)) != -1)
     {
         gettimeofday( &timev, NULL);
         t_now = ctime( &(timev.tv_sec));
         send(rsd, t_now, strlen(t_now), 0);
     }
     close( rsd);
    }
    return 0;
}
```

Пример реализации параллельного сервера с установлением логического соединения с помощью процессов:

```
    while(1)
    {
     rlen = sizeof( remote);
     rsd = accept( sd, (struct sockaddr *)&remote, &rlen);
     switch( fork())
     {
     case -1:
          exit(ERR);
     case 0:
          close( sd);
          exit ( TCP_Proc ( rsd) );
     default:
          close( rsd);
     }
    }
```

Пример реализации параллельного сервера с установлением логического соединения с помощью потоков:

```
pthread_t th;
pthread_attr_t ta;

    while(1)
    {
     rlen = sizeof( remote);
     rsd = accept( sd, (struct sockaddr *)&remote, &rlen);
     pthread_create (&th,&ta, TCP_Proc, rsd);
    }
```

Пример реализации параллельного сервера с установлением логического соединения с помощью асинхронного ввода/вывода:

```
fd_set rfds, afds;
int nfds;
...
    nfds = getdtablesize();
    FD_ZERO( &afds);
    FD_SET( sd, &afds);

    while(1)
    {
     memcpy( &rfds, &afds, sizeof(rfds));

     if ( select( nfds, &rfds, (fd_set *) 0, (fd_set *) 0,
                    (struct timeval *) 0) <0)
     {
         perror("Select");
         return 2;
     }
     if( FD_ISSET( sd, &rfds))
     {
         rlen = sizeof( remote);
         rsd = accept( sd, (struct sockaddr *)&remote, &rlen);
         FD_SET( rsd, &afds);
     }
     for( rsd=0; rsd < nfds; ++rsd)
         if( (rsd != sd) && FD_ISSET( rsd, &rfds))
         {
          TCP_Proc( rsd);
          close( rsd);
          FD_CLR( rsd, &afds);
         }
    }
```

1.4 Задание

Создать сервер и клиент, взаимодействующие между собой согласно заданному варианту. Для связи использовать сокеты.

Все клиенты должны иметь возможность задания адреса и номера порта с командной строки. Адрес сервера и порта может задаваться как в символьном, так и в числовом виде.

У клиентов должен присутствовать либо текстовый, либо графический интерфейс с отображением текущего состояния и результатов.

Таблица 1.2.

	Про-токол	**Тип сервера**	**Функция сервера**	**Примечания**
1	UDP	последовательный	Передача файла	Файл передается блоками размером в 1 КБ. В запросе указывается полный путь к файлу и номер блока. Для поиска доступных серверов используется широковещательная рассылка.
2	UDP	последовательный	Выполнение команды ОС	Результат выполнения команды передается клиенту. Для поиска доступных серверов используется широковещательная рассылка.
3	UDP	параллельный (процессы)	Выполнение команды ОС	Результат выполнения команды передается клиенту. Поддерживается работа с несколькими клиентами одновременно. Для поиска доступных серверов используется широковещательная рассылка.
4	UDP	параллельный (потоки)	Передача файла	Поддерживается работа с несколькими клиентами одновременно. Текущее положение указателя в файле в запросе не указывается ни прямо, ни косвенно. Для поиска доступных серверов используется широковещательная рассылка.
5	TCP	последовательный	Выполнение команды ОС	Клиент должен иметь возможность работы с несколькими серверами одновременно.
6	TCP	Псевдо-параллельный	Передача файла	Управление передачей с помощью срочных данных.
7	TCP	параллельный (процессы)	Выполнение команды ОС	Клиент должен иметь возможность работы с несколькими серверами одновременно.
8	TCP	параллельный (потоки)	Передача файла	Управление передачей с помощью срочных данных.

ЛАБОРАТОРНАЯ РАБОТА №2
РЕАЛИЗАЦИЯ ПРОТОКОЛА ПРИКЛАДНОГО УРОВНЯ НА ПРИМЕРЕ FTP

Длительность: 4 часа

Цель: Расширить полученные ранее знания в области сетевого программирования, более подробно изучить особенности протоколов модели OSI/ISO и практически реализовать компоненты клент-серверной модели протокола прикладного уровня FTP.

2.1. Введение

Как и следует из его названия, протокол FTP (File Transfer Protocol) предназначен для пересылки файлов между двумя удаленными станциями. Протокол формировался многие годы. Изначально задуманный для обеспечения обычной передачи информации между двумя ЭВМ, он претерпел множество изменений, усовершенствований и свой окончательный вид обрел лишь в 1985 году. Успех протокола неразрывно связан со стремительным развитием и расширением глобальной сети Internet.

Протокол FTP, как и большинство протоколов Internet, в чистом виде соответствует классической клиент-серверной модели. Для его использования необходимы два компонента:

1. FTP-клиент – обычно обслуживает запросы пользователя и работает на локальной по отношению к нему станции
2. FTP-сервер – обычно обслуживает запросы клиента и работает на удаленной станции.

В качестве транспорта, протокол FTP использует стек TCP/IP.

Ключевым документом, стандартизирующим внутреннее содержимое и все аспекты применения протокола FTP, является RFC 959. Данный стандарт входит в комплект документов Request for Comments рабочей группы Network Working Group, основу деятельности которой составляет формирование нормативных документов, относящихся ко многим вопросам из области функционирования Internet. RFC 959 заменил более раннюю версию RFC 765, а также все версии вплоть до самых первых – RFC 114 и 141. Согласно этим документам протокол рассматривается как ориентированный на пользователя.

Несмотря на то, что протокол FTP является одним из старейших, и на то, что появилось множество других более совершенных протоколов (в первую очередь следует отметить HTTP, который сейчас зачастую и ассоциируется с Internet), он не теряет своей актуальности и продолжает широко использоваться. Наличие поддержки FTP – неотъемлемая составляющая практически любой серверной операционной системы. Современные реализации протокола позволяют в некоторой степени избавиться от его недостатков.

Таким образом, поскольку протокол FTP, с одной стороны, является классическим протоколом Internet, и, с другой стороны, при выполнении задания потребует полноценного знания основных аспектов сетевого программирования, именно он и был выбран в качестве объекта изучения на данной лабораторной работе. Кроме того, реализация FTP на базе

предоставляемого ядром операционной системы транспорта TCP/IP не настолько сложна, как реализация полноценного клиента или сервера HTTP.

2.2. Обзор

Как уже было сказано выше, протокол FTP относится к протоколам, ориентированным на пользователя. Это означает, что реализация, по крайней мере клиента, обязана предоставлять пользователю более или менее функционально полный интерфейс.

Классический интерфейс FTP-клиента, широко применяющийся в оболочках UNIX и командных строках Windows, представляет собой интерпретатор командной строки, активизируемый вводом команды (UNIX)

```
#ftp
```

с последующим вызовом соответствующей программы. В качестве параметров можно задать имя или адрес FTP-сервера, а также номер порта, если он отличен от стандартного. Если команда выполнилась успешно, появляется приглашение интерпретатора:

```
ftp>
```

Существуют также множество прикладных пакетов, использующих графический интерфейс.

FTP-сервер представляет собой непрерывно выполняющуюся программу, ожидающую запросы от FTP-клиентов, выраженную в виде демона UNIX либо сервиса Windows. В операционных системах UNIX работа демонов обычно контролируется конфигурационными файлами, а в Windows – соответствующими оконными средствами.

Таким образом, обобщенную структурную схему сетевой надсистемы, использующей протокол FTP, можно представить как показано на рис. 2.1 (RFC 959).

Показана взаимосвязь между одним FTP-клиентом и одним FTP-сервером, но возможна также схема взаимодействия, когда по инициативе FTP-клиента осуществляется файловый обмен между двумя FTP-серверами. Как в составе FTP-клиента, так и в составе FTP-сервера выделяются соответствующие протокольные интерпретаторы (protocol interpreters) и процессы пересылки данных (data transfer processes).

Термин «удаленный» здесь и далее по тексту используется исключительно в пространственном смысле.

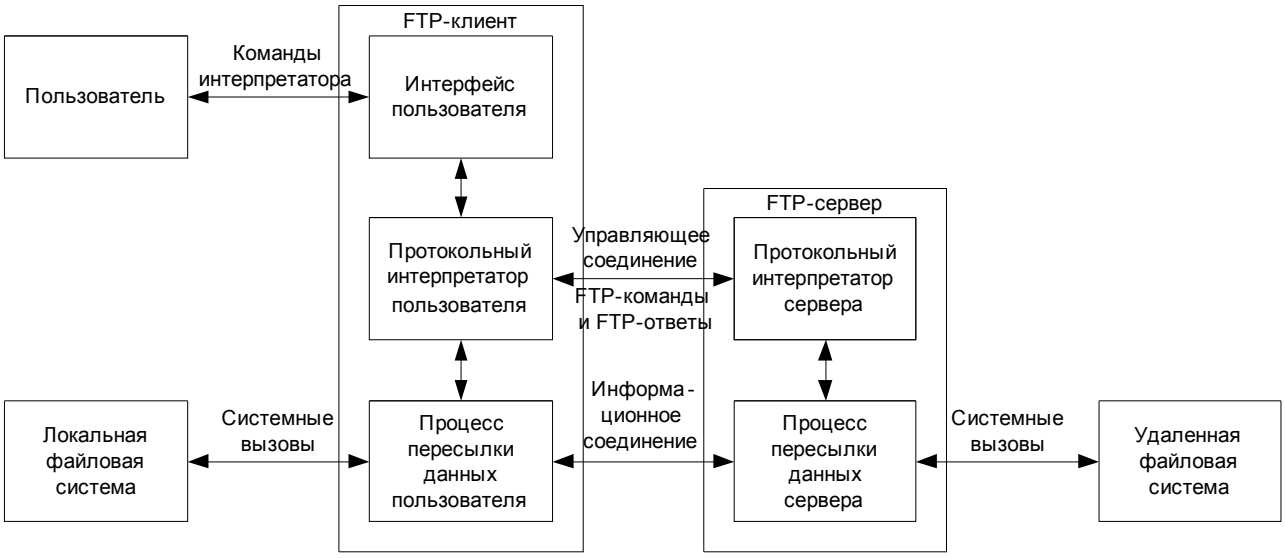


Рис. 2.1. Структура сетевой надсистемы, работающей по протоколу FTP

Как видно, в отличие от многих других, протокол FTP связан не с одним, а с двумя соединениями и, соответственно, стандартом (RFC 1060) для него зарезервированы два номера портов (на стороне FTP-сервера), указанные в табл. 2.1.

Таблица 2.1

Порт	Название	Описание
20	`FTP Data`	Управляющее соединение
21	`FTP Control`	Информационное соединение

Сначала FTP-клиентом формируется управляющее соединение (control connection), которое в дальнейшем используется только для передачи FTP-команд от FTP-клиента и FTP-ответов от FTP-сервера. FTP-сервер принимает, интерпретирует и выполняет FTP-команды, а также передает FTP-ответы. Одно или несколько информационных соединений (data connections), предназначенных исключительно для пересылки данных, т.е. файлов и каталогов, формируются FTP-сервером либо FTP-клиентом. Они не существуют на протяжении всего сеанса взаимодействия и могут создаваться и ликвидироваться по мере необходимости. Управляющее же соединение может быть завершено только после осуществления полезного информационного обмена, если таковой был.

В некоторых особых ситуациях может делаться отказ от использования стандартных портов.

Также видно, что можно выделить три уровня, связанных с применением протокола FTP:

1. Настройка, запуск и использование пользователем FTP-клиента, а администратором – FTP-сервера.
2. Работа пользователя с протокольным интерпретатором.
3. Скрытое от пользователя взаимодействие непосредственно по протоколу FTP.

На каждом из этих уровней существует свое понятие термина «команда».

На самом высоком уровне это то, что, например, в UNIX традиционно называется командой операционной системы – в дальнейшем, просто «команда». На промежуточном уровне это уже команда, вводимая при работе с программой FTP-клиента – в дальнейшем, «команда интерпретатора». И, наконец, на низком уровне это собственно команда протокола, передаваемая через управляющее соединение, – в дальнейшем, «FTP-команда». Некоторые аббревиатуры команд интерпретатора и FTP-команд совпадают. Но, особенно при программировании, необходимо понимать, что аббревиатуры все-таки отличаются, и учитывать тот факт, что одна команда интерпретатора может реализовываться последовательностью из нескольких FTP-команд.

В настоящее время широко распространены т.н. «анонимные» (anonymous) FTP-сервера (RFC 1635), предоставляющие всем желающим определенные файловые ресурсы (обычно расположенные в каталоге `/pub`). Как правило, для аутентификации в такой системе достаточно ввести имя пользователя `anonymous` и произвольный пароль, например адрес e-mail.

2.3. FTP-команды

На каждом шаге работы по протоколу FTP активно генерируются FTP-команды согласно BNF (Backus-Naur Form). FTP-команда передается только по управляющему соединению, только FTP-клиентом и представляет собой последовательность длиной до четырех байт, за которой могут следовать параметры.

Разбиение FTP-команд на группы, их параметры и описание (RFC 959) приведены в табл. 2.2.

Таблица 2.2

FTP-команда	Название	Описание
1	2	3
	Команды контроля доступа (Access Control Commands)	
`USER <SP> <имя_пользователя> <CRLF>`	USER NAME	Идентификация пользователя, необходимая для получения доступа к удаленной файловой системе
`PASS <SP> <пароль> <CRLF>`	PASSWORD	Аутентификация пользователя (должна следовать непосредственно за `USER`)
`ACCT <SP> <аккаунт> <CRLF>`	ACCOUNT	Необходимый для входа в удаленную систему пользовательский аккаунт (альтернатива паре `USER` и `PASS` во время доступа к специфическим ресурсам)
`CWD <SP> <путь> <CRLF>`	CHANGE WORKING DIRECTORY	Смена каталога удаленной файловой системы
`CDUP <CRLF>`	CHANGE TO PARENT DIRECTORY	Переход к родительскому каталогу удаленной файловой системы
`SMNT <SP> <путь> <CRLF>`	STRUCTURE MOUNT	Монтирование пользователем другой файловой системы без выхода
`REIN <CRLF>`	REINITIALIZE	Исключение пользователя из удаленной системы с завершением текущего действия и сохранением настроек
`QUIT <CRLF>`	LOGOUT	Выход пользователя из удаленной системы с завершением текущего действия

Продолжение табл. 2.2

1	2	3

	Команды управления пересылкой (Transfer Parameter Commands)	
PORT <SP> <адрес_и_порт> <CRLF>	DATA PORT	Совокупность IP-адреса и номера порта, необходимая для формирования информационного соединения (передается в виде: h1,h2,h3,h4,p1,p2; где h1,h2,h3,h4 – разбитый на четыре разделенных запятой байта IP-адрес в десятичном представлении, причем байты следуют в правильном порядке; и p1,p2 – аналогичным образом разбитый 16-тибитный номер порта, причем байты также следуют начиная со старшего)
PASV <CRLF>	PASSIVE	Задание пассивного режима обмена
TYPE <SP> <код_файлового_ представления> <CRLF>	REPRESENTATION TYPE	Задание типа файлового представления (A – ASCII, N – Non-print, T – Telnet format effectors, E –EBCDIC, C = Carriage Control, I – Image, L – Local byte size, причем, A и N задаются по умолчанию)
STRU <SP> <код_структуры_файла> <CRLF>	FILE STRUCTURE	Задание структуры файла (F – File Structure, R – Record Structure, P – Page Structure, причем, F задается по умолчанию)
MODE <SP> <код_способа_пересылки> <CRLF>	TRANSFER MODE	Задание способа пересылки (S – Stream, B – Block, C – Compressed, причем, S задается по умолчанию)
	Сервисные команды (Service Commands)	
RETR <SP> <путь> <CRLF>	RETRIEVE	Пересылка полной копии указанного удаленного файла с FTP-сервера (основа для скачивания (download) файлов)
STOR <SP> <путь> <CRLF>	STORE	Сохранение FTP-сервером получаемых данных в виде указанного файла, причем, если файл уже существовал, то он обновляется (основа для закачивания (upload) файлов на FTP-сервер)
STOU <CRLF>	STORE UNIQUE	Сохранение FTP-сервером получаемых данных в виде уникального файла
APPE <SP> <путь> <CRLF>	APPEND (with create)	Сохранение FTP-сервером получаемых данных в виде указанного файла, причем, если файл уже существовал, то данные дописываются в его конец
ALLO <SP> <десятичное_целое_ число> [<SP> R <SP> <десятичное_целое_ число>] <CRLF>	ALLOCATE	Резервирование сервером файлового пространства в байтах, причем, если планируемый для пересылки файл имеет структуру R либо P, то после символа R указывается максимальный размер записи либо страницы
REST <SP> <смещение> <CRLF>	RESTART	Начать пересылку файла с указанного смещения в нем (используется для организации докачки)
RNFR <SP> <путь> <CRLF>	RENAME FROM	Указание удаленного файла перед переименованием
RNTO <SP> <путь> <CRLF>	RENAME TO	Переименование удаленного файла (должна следовать непосредственно за RNFR)
ABOR <CRLF>	ABORT	Принудительное завершение предыдущей FTP-команды и связанной с ней пересылки

Окончание табл. 2.2

1	2	3
DELE <SP> <путь> <CRLF>	DELETE	Удаление удаленного файла или каталога
RMD <SP> <путь> <CRLF>	REMOVE DIRECTORY	Удаление удаленного каталога

MKD <SP> <путь> <CRLF>	MAKE DIRECTORY	Создание удаленного каталога
PWD <CRLF>	PRINT WORKING DIRECTORY	Вывести текущий удаленный каталог
LIST [<SP> <путь>] <CRLF>	LIST	Вывести подробный список файлов из удаленного каталога (если путь не указан, то подразумевается текущий удаленный каталог)
NLST [<SP> <путь>] <CRLF>	NAME LIST	Вывести простой список файлов из удаленного каталога (если путь не указан, то подразумевается текущий удаленный каталог)
SITE <SP> <строка> <CRLF>	SITE PARAMETERS	Предоставление сервером специфической информации
SYST <CRLF>	SYSTEM	Запрос информации об удаленной операционной системе
STAT [<SP> <путь>] <CRLF>	STATUS	Запрос информации у FTP-сервера о текущих параметрах (FTP-ответ может пересылаться как по управляющему, так и по информационному соединению)
HELP [<SP> <строка>] <CRLF>	HELP	Целевой запрос справочной информации у FTP-сервера (обычно информации о FTP-команде, причем, если строка не указана, то выдается обобщенная информация)
NOOP <CRLF>	NOOP	Холостая FTP-команда (обычно используется для поддержания связи)

Параметры отделяются пробелами <SP>. Все команды завершаются парой символов возврата каретки и перевода строки <CRLF>. В квадратные скобки заключены опциональные аргументы.

На рис. 2.2 показан список FTP-команд, поддерживаемых FTP-сервером SunOS 5.9.

2.4. FTP-ответы

Каждая команда, переданная FTP-клиентом, должна сопровождаться по крайней мере одним FTP-ответом со стороны FTP-сервера, сообщающим об успешности ее выполнения. В нормальной ситуации, FTP-клиент ожидает FTP-ответ на текущую FTP-команду перед тем, как передать следующую. При этом широко применяется механизм тайм-аута. В зависимости от реализации, на часть FTP-команд могут возвращаться различные комбинации FTP-ответов, однако существуют и жесткие ограничения. Существуют также рекомендации по наполнению FTP-ответов текстовыми комментариями (RFC 959).

FTP-ответ, состоящий из одной строки, формально определяется следующим образом:

xyz <SP> <текст> <CRLF>

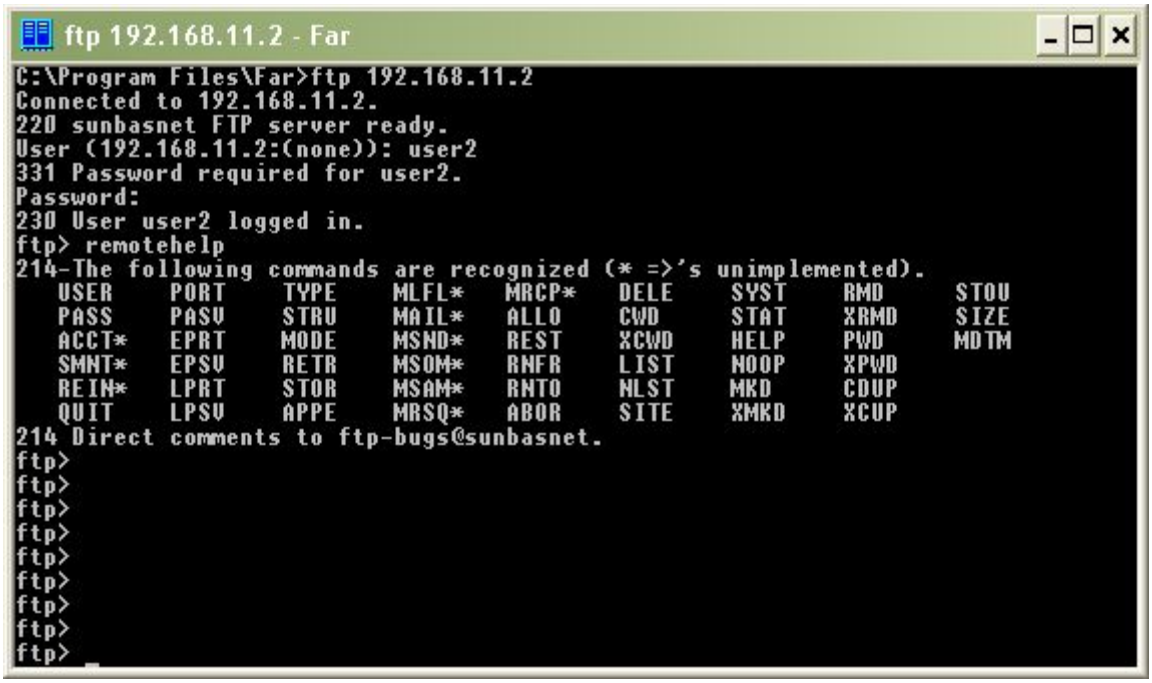


Рис. 2.2. FTP-команды, поддерживаемые FTP-сервером SunOS 5.9

Если же FTP-ответ состоит из нескольких строк, что также допускается, он выглядит:

```
xyz <-> <текст> <CRLF>
<текст> <CRLF>
...
xyz <текст> <CRLF>
```

Целочисленный трехбайтовый код `xyz` декодируется согласно табл. 2.3.

Таблица 2.3

Код	Название	Описание
1	2	3
1yz	Positive Preliminary	Предварительное успешное завершение
2yz	Positive Completion	Окончательное успешное завершение
3yz	Positive Intermediate Reply	Промежуточное успешное завершение
4yz	Transient Negative Completion	Ненормальное завершение в текущем случае
5yz	Permanent Negative Completion	Однозначно ненормальное завершение
x0z	Syntax	Синтаксис
x1z	Information	Информация
x2z	Connections	Соединения
x3z	Authentication and Accounting	Аутентификация и аккаунты
x4z	Unspecified As Yet	Стандартом не определено
x5z	File System	Файловая система
110	Restart Marker Reply	Подтверждение изменения файлового смещения (вне зависимости от реализации должно быть: MARK <SP> yyyy <SP> = <SP> mmmm, где yyyy и mmmm – смещения в удаленной и локальной файловых системах соответственно)

Продолжение табл. 2.3

1	2	3
120	Service Ready in nnn Minutes	Запрос планируется обслужить за nnn минут
125	Data Connection Already Open. Transfer Starting	Информационное соединение установлено и пересылка начата
150	File Status Okay. About to Open Data Connection	Статус файла корректен, подготавливается информационное соединение
200	Command Okay	FTP-команда выполнена успешно
202	Command Not Implemented, Superfluous At This Site	В выполнении команды нет необходимости
211	System Status, or System Help Reply	Статус системы или справочная информация
212	Directory Status	Статус каталога
213	File Status	Статус файла
214	Help Message	Справочное сообщение
215	NAME System Type	Официальный тип системы NAME
220	Service Ready For New User	Готовность обслуживать нового пользователя (обычно содержит текстовое приветствие)
221	Service closing control connection	Управляющее соединение завершено
225	Data connection open	Информационное соединение установлено
226	Closing data connection	Информационное соединение завершается
227	Entering Passive Mode	Пассивный режим обмена установлен
230	User Logged In, Proceed	Пользователь вошел в систему, можно продолжать
250	Requested File Action Okay, Completed	Запрошенное действие с файлом выполнено
257	"PATHNAME" Created	Полный путь в файловой системе создан
331	User Name Okay, Need Password	Имя пользователя воспринято, требуется пароль
332	Need Account for Login	Требуется аккаунт для входа в систему
350	Requested File Action Pending Further Information	Запрошенное действие с файлом отложено до поступления дополнительной информации
421	Service not Available, Closing Control Connection	Сервис не доступен, управляющее соединение завершено
425	Can't Open Data Connection	Невозможно установить информационное соединение
426	Connection Closed; Transfer Aborted	Соединение завершено, пересылка прервана
450	Requested File Action not Taken	Запрошенное действие с файлом недопустимо
451	Requested Action Aborted. Local Error In Processing	Запрошенное действие прервано по причине внутренней ошибки
452	Requested Action not Taken. Insufficient Storage Space in System	Запрошенное действие недопустимо по причине недостатка свободного файлового пространства в системе
500	Syntax Error, Command Unrecognized	Синтаксическая ошибка, FTP-команда не распознана
501	Syntax Error in Parameters or Arguments	Синтаксическая ошибка, связанная тс параметрами или аргументами
502	Command not Implemented	FTP-команда не выполнена
503	Bad Sequence of Commands	Недопустимая последовательность FTP-команд
504	Command not Implemented for That Parameter	FTP-команда с этим параметром не выполнена
530	Not Logged In	Отсутствует вход в систему
532	Need Account for Storing Files	Требуется аккаунт для сохранения файла
550	Requested Action not Taken. File Unavailable	Запрошенное действие недопустимо по причине недоступности файла
551	Requested Action Aborted: Page Type Unknown	Запрошенное действие недопустимо, так как неизвестен тип страницы

Продолжение табл. 2.3

1	2	3

552	Requested File Action Aborted. Exceeded Storage Allocation	Запрошенное действие с файлом завершено ненормально по причине превышения размера доступного файлового пространства
553	Requested Action Not Taken. File Name not Allowed	Запрошенное действие недопустимо по причине имени файла

Возможные варианты комбинаций FTP-ответов на FTP-команды приведены в табл.2.4.

Таблица 2.4

FTP-команда	FTP-ответы
установка соединения	120; 220; 220; 421
USER	230; 530; 500, 501, 421; 331, 332
PASS	230; 202; 530; 500, 501, 503, 421; 332
ACCT	230; 202; 530; 500, 501, 503, 421
CWD	250; 500, 501, 502, 421, 530, 550
CDUP	200; 500, 501, 502, 421, 530, 550
SMNT	202, 250; 500, 501, 502, 421, 530, 550
REIN	120; 220; 220; 421; 500, 502
QUIT	221; 500
PORT	200; 500, 501, 421, 530
PASV	227; 500, 501, 502, 421, 530
MODE	200; 500, 501, 504, 421, 530
TYPE	200; 500, 501, 504, 421, 530
STRU	200; 500, 501, 504, 421, 530
ALLO	200; 202; 500, 501, 504, 421, 530
REST	500, 501, 502, 421, 530; 350
STOR	125, 150; (110); 226, 250; 425, 426, 451, 551, 552; 532, 450, 452, 553; 500, 501, 421, 530
STOU	125, 150; (110); 226, 250; 425, 426, 451, 551, 552; 532, 450, 452, 553; 500, 501, 421, 530
RETR	125, 150; (110); 226, 250; 425, 426, 451; 450, 550; 500, 501, 421, 530
LIST	125, 150; 226, 250; 425, 426, 451; 450; 500, 501, 502, 421, 530
NLST	125, 150; 226, 250; 425, 426, 451; 450; 500, 501, 502, 421, 530
APPE	125, 150; (110); 226, 250; 425, 426, 451, 551, 552; 532, 450, 550, 452, 553; 500, 501, 502, 421, 530
RNFR	450, 550; 500, 501, 502, 421, 530; 350
RNTO	250; 532, 553; 500, 501, 502, 503, 421, 530
DELE	250; 450, 550; 500, 501, 502, 421, 530
RMD	250; 500, 501, 502, 421, 530, 550
MKD	257; 500, 501, 502, 421, 530, 550
PWD	257; 500, 501, 502, 421, 550
ABOR	225, 226; 500, 501, 502, 421
SYST	215; 500, 501, 502, 421
STAT	211, 212, 213; 450; 500, 501, 502, 421, 530
HELP	211, 214; 500, 501, 502, 421
SITE	200; 202; 500, 501, 530
NOOP	200; 500, 421

2.5. Команды интерпретатора

В отличие от FTP-команд, команды интерпретатора не стандартизированы, однако в реализациях широко применяются традиционно сложившиеся аббревиатуры. На рис. 2.5 и 2.6 показаны команды, поддерживаемые интерпретаторами FTP-клиентов Windows Server 2003 и ALT Linux соответственно.

В табл. 2.5 приведено поверхностное описание команд интерпретатора FTP-клиента ALT Linux.

2.6. Параметры обмена

Протокол FTP разрабатывался как универсальный – в том числе, и для пересылки файлов между станциями, работающими под управлением различных операционных систем, возможно использующих различные файловые системы. Для того, чтобы обмен по протоколу прошел успешно, необходимо правильно задать или изменить используемые по умолчанию значения следующих параметров.

2.6.1. Режим обмена между FTP-клиентом и FTP-сервером

В зависимости от того, какая из взаимодействующих сторон является инициатором установления информационного соединения различают активный и пассивный режимы обмена. При этом направление пересылки файлов, т.е. какая из сторон является передатчиком, а какая – приемником, не имеет значения.

Активный режим является рекомендуемым и наиболее используемым. В активном режиме управляющее соединение создается следующим образом. FTP-клиент, используя динамически выделенный порт (с номером больше `1024`), формирует сокет (виртуальный канал) с портом `21` FTP-сервера. Затем, для формирования информационного соединения, FTP-клиент динамически выделяет еще один порт и посылает его номер FTP-серверу (с помощью FTP-команды `PORT`). FTP-сервер формирует сокет с указанным портом, со своей стороны используя порт `20`. Если FTP-клиент не передает команду `PORT`, что в крайней степени не рекомендуется, то FTP-сервер формирует сокет с тем же самым портом FTP-клиента, который используется управляющим соединением. В таких случаях требуется указание особых аргументов системных вызовов в программе процесса пересылки данных пользователя. Аналогичным способом можно решить проблему конфликта с портом `20` и на сервере при необходимости реализации сетевой многопоточности, хотя, опять же, в большинстве современных реализаций предпочтение отдается динамическому выделению портов. Диаграмма взаимодействия по правилам активного режима показана на рис. 2.7.

Рис. 2.5. Команды интерпретатора FTP-клиента Windows Server 2003

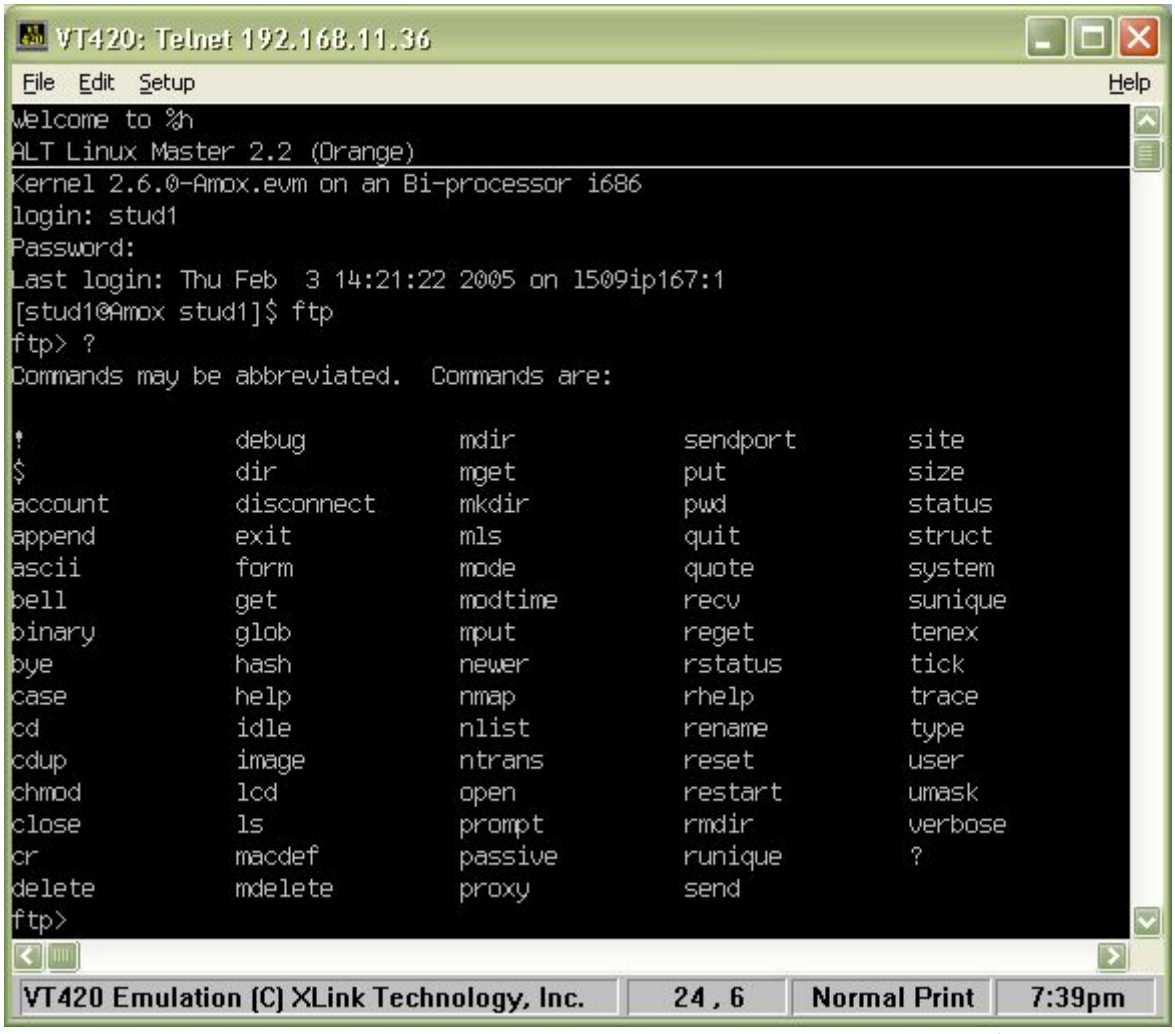


Рис. 2.6. Команды интерпретатора FTP-клиента ALT Linux

Таблица 2.5

Команда и	Описание	Команда и	Описание
!	Выполнить локальную команду SHELL. Если параметр не задан, выйти в интерактивный режим	newer	Скачать удаленный файл, если он новее локального
$	Выполнить локальное макро	nmap	Установить шаблоны отображения имен файлов
account	Войти в удаленную систему	nlist	ls
append	Дописать локальный файл в конец удаленного файла	ntrans	Установить таблицу преобразования имен файлов
ascii	Вкл/выкл ASCII	open	Войти в удаленную систему
bell	Вкл/выкл подачу звукового сигнала после успешного завершения команды интерпретатора	promt	Вкл/выкл интерактивный режим
binary	Вкл/выкл Image	passive	Задать пассивный режим обмена
bye	Выйти из интерпретатора	proxy	Передать FTP-команду прокси
case	Вкл/выкл приведение к строчным буквам	sendport	Вкл/выкл передачу порта для каждого информационного соединения
cd	UNIX cd. Удаленно	put	Закачать файл из текущего локального каталога
cdup	UNIX cd .. Удаленно	pwd	UNIX pwd. Удаленно
chmod	UNIX chmod. Удаленно	quit	exit
close	Выити из удаленной системы	quote	Передать произвольную FTP-команду
cr	Вкл/выкл подавление <CR> ASCII	recv	get
delete	UNIX rm. Удаленно	reget	Скачать удаленный файл и дописать его в конец локального файла
debug	Вкл/выкл отладку	rstatus	Показать статус удаленной станции
dir	UNIX ls -l. Удаленно	rhelp	Выдать справочную информацию об FTP-команде (-ax), поддерживаемых удаленным FTP-сервером
disconnect	Выйти из удаленной системы	rename	Переименовать удаленный файл
exit	Выйти из интерпретатора	reset	Очистить очередь из FTP-ответов
form	Задать файловое представление	restart	Перезапустить пересылку начиная со смещения
get	Скачать удаленный файл в текущий локальный каталог	rmdir	UNIX rmdir. Удаленно
glob	Вкл/выкл поддержку шаблонов для имен локальных файлов	runique	Вкл/выкл поддержку формирования уникальных имен локальных файлов
hash	Вкл/выкл вывод на экран # для каждого переданного буфера	send	put
help	Выдать справочную информацию о команде (-ax) интерпретатора	site	Вывести специфическую информацию
idle	Управление соответствующим удаленным таймером	size	Показать размер удаленного файла
image	Вкл/выкл Image	status	Показать текущий статус
lcd	UNIX cd. Локально	struct	Задать структуру файла
ls	UNIX ls. Удаленно	system	Показать тип удаленной станции
macdef	Определить макро	sunique	Вкл/выкл поддержку формирования уникальных имен удаленных файлов
mdelete	UNIX rm. Несколько удаленных файлов	tenex	Задать страничное представление tenex
mdir	UNIX ls -l. Несколько удаленных файлов	tick	Вкл/выкл подсчет байтов при пересылке
mget	Скачать несколько файлов	trace	Вкл/выкл отслеживание пакетов при пересылке
mkdir	UNIX mkdir. Удаленно	type	Задать файловое представление
mls	UNIX ls. Несколько удаленных каталогов	user	Ввести имя пользователя
mode	Задание способа пересылки	umask	Задать маску создания удаленных файлов
modtime	Показать время последней модификации удаленного файла	verbose	Вкл/выкл режим подробного вывода на экран
mput	Закачать несколько файлов	?	help

Пассивный режим обычно устанавливается принудительно (с помощью FTP-команды PASV). В пассивном режиме FTP-клиент создает как управляющее, так и информационное соединения с FTP-сервером. После того,

как аналогичным образом создано управляющее соединение, FTP-клиент передает FTP-команду `PASV` FTP-серверу. Получив ее, FTP-сервер динамически выделяет порт для информационного соединения и передает его номер FTP-клиенту (с помощью FTP-ответа `227`). Далее, FTP-клиент динамически выделяет еще один порт и формирует сокет с портом, номер которого получил от FTP-сервера. Диаграмма взаимодействия по правилам пассивного режима показана на рис. 2.7.

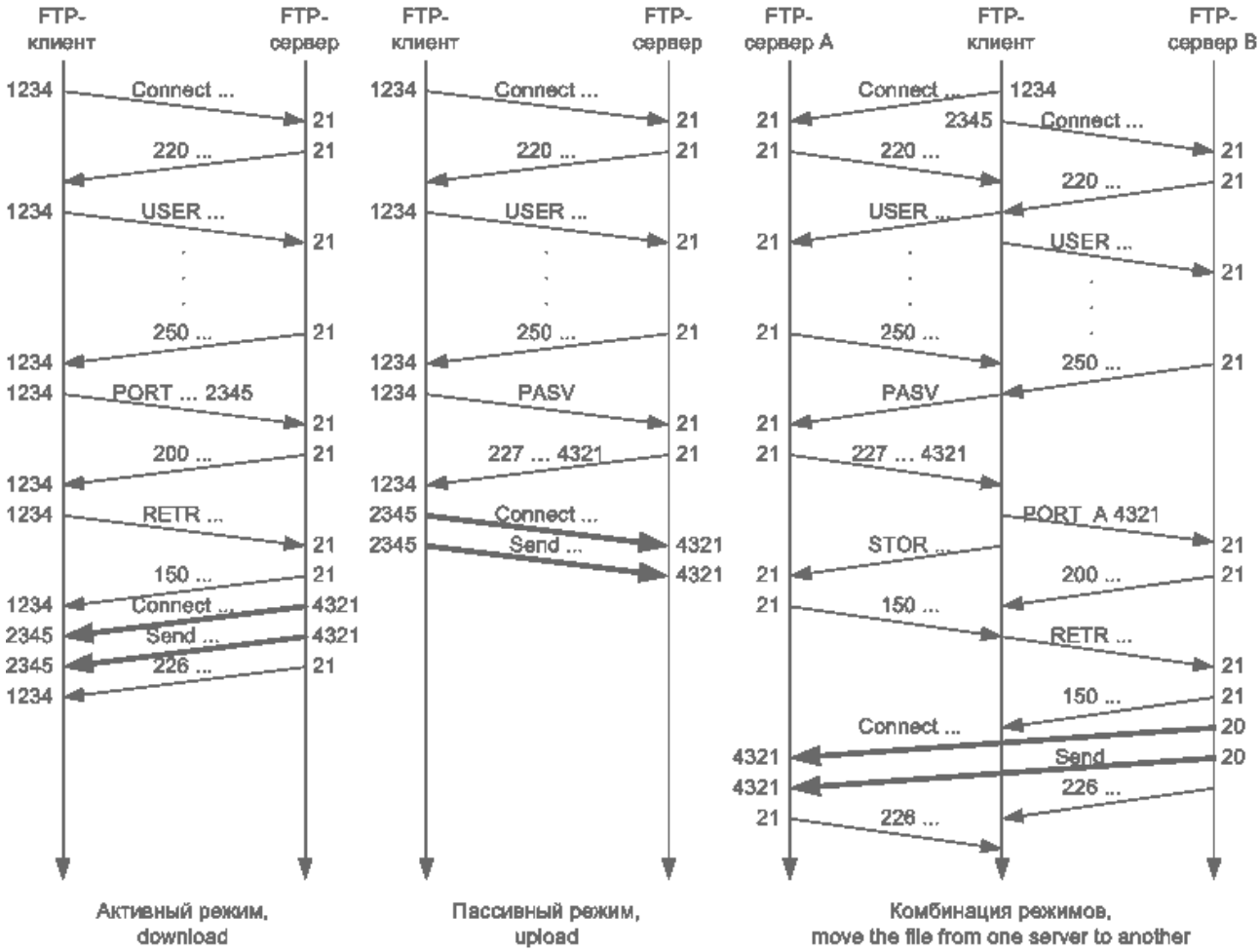


Рис. 2.7. Режимы обмена

Дополнительно показана пересылка файла между двумя FTP-серверами под управлением FTP-клиента. Тонкие стрелки отражают управляющие соединения, а толстые – информационные. Направления стрелок соответствуют направлениям передачи.

2.6.2. Тип файлового представления

Файловое представление необходимо для управления пересылкой информации из файловой системы передающей стороны в файловую систему принимающей стороны. Поскольку эти файловые системы зачастую несовместимы, для корректного сохранения данных операционная система должна знать, файл какого типа она принимает. Протокол FTP поддерживает четыре основных файловых представления:

1. ASCII – файл считается текстовым и пересылается посредством 7-мибитного кода NVT-ASCII (Network Virtual Terminal - American Standard Code for Information Interchange); 7 битов вкладываются в стандартный байт с нулем в старшем бите; передатчик выполняет преобразование из своего внутреннего представления в NVT-ASCII, а приемник – из NVT-ASCII уже в свое внутреннее представление; в качестве разделителей строк текстового файла используется пара символов `<CRLF>` ; это представление должно поддерживаться всеми реализациями.
2. EBCDIC – для пересылки текстового файла используется кодировка EBCDIC (Extended Binary-Coded Decimal Interchange Code) фирмы IBM.
3. Image – файл передается как непрерывный поток битов, упакованных в 8-мибитные байты.
4. Local byte size – файл передается как непрерывный поток битов между передатчиком и приемником, которые имеют различные размеры байтов.

И три дополнительных:

5. Non-print – считается, что файл не содержит никаких специальных символов вертикального форматирования; это представление должно поддерживаться всеми реализациями.
6. Telnet format effectors – файл содержит специальные символы управления вертикальным форматированием кодов ASCII/EBCDIC: `<CR>` , `<LF>` , `<NL>` , `<VT>` , `<FF>` .
7. Carriage Control – файл содержит специальные символы управления вертикальным форматированием ASA (FORTRAN): `blank` , `0`, `1`, `+` .

2.6.3. Структура файла

В дополнение к файловому представлению, в протоколе FTP предусмотрена возможность структурировать файл при его пересылке по информационному соединению. Поддерживаются три варианта:

1. File Structure – файл не имеет внутренней структуры и воспринимается как непрерывный поток байтов; поддержка должна быть включена во все реализации.
2. Record Structure – файл рассматривается как последовательность записей; структура приемлема только для текстовых файлов.
3. Page Structure – считается, что файл имеет страничную организацию; каждая страница имеет заголовок и индексацию; структура зависит от реализации.

2.6.4. Способ пересылки информации

Наконец, существует возможность установить способ пересылки. Возможны три варианта:

1. Stream – файл пересылается как непрерывный поток байтов; если файл не имеет внутренней структуры, то прием специального символа <EOF> означает, что пересылка окончена; для случаев со сложной структурой предусмотрены специальные коды для <EOR> и <EOF>; поддержка должна быть включена во все реализации.
2. Block – файл пересылается в виде последовательности блоков, каждый из которых имеет заголовок, в котором записываются счетчик байтов и специальные коды; способ поддерживается редко.
3. Compressed – файл пересылается в сжатом простейшими алгоритмами виде; способ поддерживается крайне редко по причине применения в случае надобности более прогрессивных алгоритмов еще до пересылки.

2.7. Примеры

Типовой простейший сеанс с задействованием протокола FTP на пользовательском уровне содержит несколько шагов:

1. Вход в удаленную систему.
2. Просмотр удаленных каталогов и выбор удаленного файла.
3. Определение параметров обмена.
4. Выполнение собственно пересылки файла на локальную систему.
5. Выход из удаленной системы.

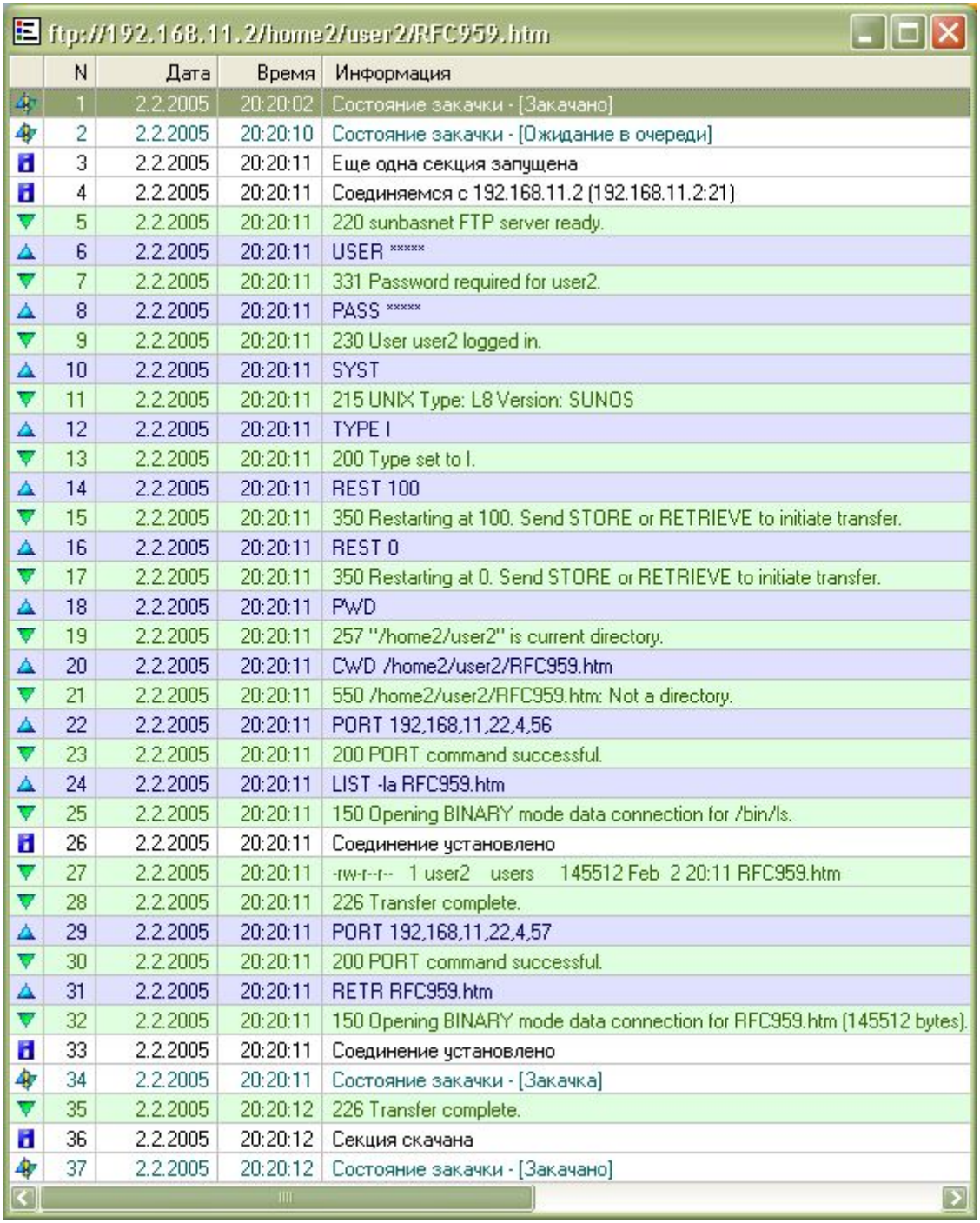
Отсюда вытекает минимальный набор команд интерпретатора и FTP-команд, которые должна поддерживать минимальная реализация.

Пример подробного лога скачивания файла с FTP-сервера SunOS 5.9 с помощью FTP-клиента, интегрированного в программу ReGet, в активном и пассивном (отличающийся фрагмент) режиме показаны на рис. 2.8 и 2.9 соответственно.

2.8. Задание

Реализовать FTP-клиент с интерпретатором либо FTP-сервер с поддержкой многопоточности. Реализовать минимальный набор FTP-команд (USER, PASS, RETR, STOR, REST, QUIT) и три любые дополнительные команды из заданной группы. Реализовать механизм тайм-аута. Пакеты должны иметь фиксированный размер.

Варианты указаны в табл. 2.6.



N	Дата	Время	Информация
1	2.2.2005	20:20:02	Состояние закачки - [Закачано]
2	2.2.2005	20:20:10	Состояние закачки - [Ожидание в очереди]
3	2.2.2005	20:20:11	Еще одна секция запущена
4	2.2.2005	20:20:11	Соединяемся с 192.168.11.2 (192.168.11.2:21)
5	2.2.2005	20:20:11	220 sunbasnet FTP server ready.
6	2.2.2005	20:20:11	USER *****
7	2.2.2005	20:20:11	331 Password required for user2.
8	2.2.2005	20:20:11	PASS *****
9	2.2.2005	20:20:11	230 User user2 logged in.
10	2.2.2005	20:20:11	SYST
11	2.2.2005	20:20:11	215 UNIX Type: L8 Version: SUNOS
12	2.2.2005	20:20:11	TYPE I
13	2.2.2005	20:20:11	200 Type set to I.
14	2.2.2005	20:20:11	REST 100
15	2.2.2005	20:20:11	350 Restarting at 100. Send STORE or RETRIEVE to initiate transfer.
16	2.2.2005	20:20:11	REST 0
17	2.2.2005	20:20:11	350 Restarting at 0. Send STORE or RETRIEVE to initiate transfer.
18	2.2.2005	20:20:11	PWD
19	2.2.2005	20:20:11	257 "/home2/user2" is current directory.
20	2.2.2005	20:20:11	CWD /home2/user2/RFC959.htm
21	2.2.2005	20:20:11	550 /home2/user2/RFC959.htm: Not a directory.
22	2.2.2005	20:20:11	PORT 192,168,11,22,4,56
23	2.2.2005	20:20:11	200 PORT command successful.
24	2.2.2005	20:20:11	LIST -la RFC959.htm
25	2.2.2005	20:20:11	150 Opening BINARY mode data connection for /bin/ls.
26	2.2.2005	20:20:11	Соединение установлено
27	2.2.2005	20:20:11	-rw-r--r-- 1 user2 users 145512 Feb 2 20:11 RFC959.htm
28	2.2.2005	20:20:11	226 Transfer complete.
29	2.2.2005	20:20:11	PORT 192,168,11,22,4,57
30	2.2.2005	20:20:11	200 PORT command successful.
31	2.2.2005	20:20:11	RETR RFC959.htm
32	2.2.2005	20:20:11	150 Opening BINARY mode data connection for RFC959.htm (145512 bytes).
33	2.2.2005	20:20:11	Соединение установлено
34	2.2.2005	20:20:11	Состояние закачки - [Закачка]
35	2.2.2005	20:20:12	226 Transfer complete.
36	2.2.2005	20:20:12	Секция скачана
37	2.2.2005	20:20:12	Состояние закачки - [Закачано]

Рис. 2.8. Пример лога скачивания в активном режиме

	19	2.2.2005	20:23:55	CWD /home2/user2/RFC959.htm
	20	2.2.2005	20:23:55	550 /home2/user2/RFC959.htm: Not a directory.
	21	2.2.2005	20:23:55	PASV
	22	2.2.2005	20:23:55	227 Entering Passive Mode (192,168,11,2,101,23)
	23	2.2.2005	20:23:55	LIST -la RFC959.htm
	24	2.2.2005	20:23:55	Connecting to (192.168.11.2:25879)
	25	2.2.2005	20:23:55	150 Opening BINARY mode data connection for /bin/ls.
	26	2.2.2005	20:23:55	-rw-r--r-- 1 user2 users 145512 Feb 2 20:11 RFC959.htm
	27	2.2.2005	20:23:55	226 Transfer complete.

Рис. 2.9. Пример лога скачивания в пассивом режиме (отличающийся фрагмент)

Таблица 2.6

№	Компонент	Режим обмена	Файловое представление	Группа команд
1.	клиент	активный	Image	Access Control
2.	сервер	активный	Image	Access Control
3.	клиент	пассивный	Image	Service
4.	сервер	пассивный	Image	Service
5.	клиент	активный	ASCII	Service
6.	сервер	активный	ASCII	Service
7.	клиент	пассивный	ASCII	Service
8.	сервер	пассивный	ASCII	Service

Лабораторная работа №3
ИЗУЧЕНИЕ СВОЙСТВ И АЛГОРИТМОВ РАСПРЕДЕЛЕННЫХ СИСТЕМ

Тема: Изучить типы, задачи и свойства распределенных систем. Реализовать распределенную систему, использующую изученные алгоритмы.
Длительность: 4 часа.

3.1Введение

Существует множество определений для распределенных систем, однако ни одно из них не является удовлетворительным и не согласуется с остальными, поэтому будем придерживаться следующего определения: *распределенная система* – это набор независимых компьютеров, представляющийся их пользователям единой объединенной системой.

От программного обеспечения намного больше, чем от аппаратного, зависит то, как система будет в конечном итоге выглядеть. Распределенные системы очень похожи на традиционные операционные системы. Прежде всего они работают как менеджеры ресурсов для существующего аппаратного обеспечения, а также скрывают природу аппаратного обеспечения, предоставляя виртуальные машины для выполнения приложений.

Операционные системы для распределенной системы можно разделить на 2 категории с точки зрения распределенности: *сильно связанные* и *слабо связанные*.

Сильно связанные называют *распределенными операционные системы* – DOS (distributed operating system). Такая операционная система старается работать с одним, глобальным представлением ресурсов, которыми она управляет, и используется для управления мультипроцессорными и гомогенными мультикомпьютерными системами. Основная цель, как и для обычных операционных систем, заключается в том, чтобы скрыть тонкости управления аппаратным обеспечением, которое одновременно используется множеством процессов.

Слабо связанные *сетевые операционные системы* – NOS (network operating system), представляются для пользователя как набор операционных систем, которые работают на локальных компьютерах и предоставляет свои службы для удаленных клиентов. Чаще всего такие операционные системы используются для управления гетерогенными мультикомпьютерными системами.

К сожалению, служб только операционной системы недостаточно, чтобы получилась распределенная система. Поэтому существуют дополнительные компоненты, предназначенные для организации поддержки прозрачности распределения. Эти средства называют *системой промежуточного уровня* (middleware, см. рис. 3.1), и они лежат в основе современных распределенных систем.

Существует несколько моделей систем промежуточного уровня, ниже приведены наиболее распространенные из них:

- представление всех объектов в виде файлов

Этот подход был введен в UNIX и соблюдался в Plan 9. Подобный подход применяется в программном обеспечении промежуточного уровня, построенном по принципу распределенной файловой системы. Этот подход достаточно популярен, хотя в большинстве систем прозрачность поддерживается только для обычных типов файлов.

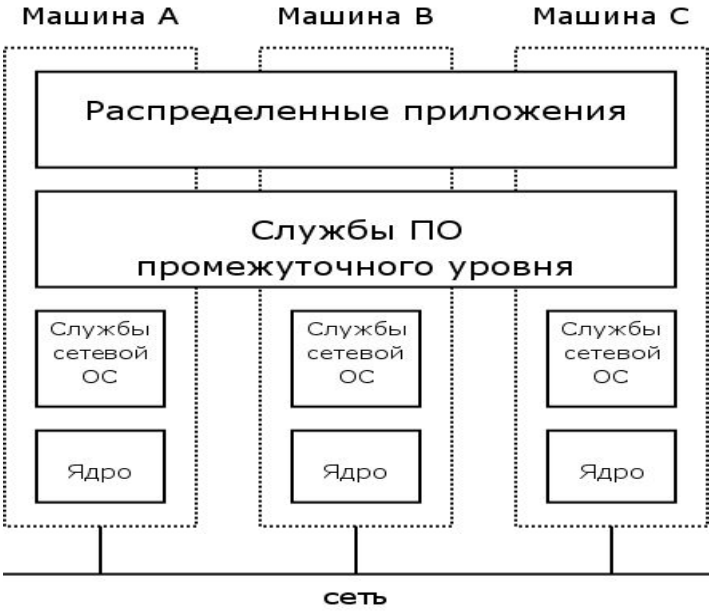


Рис.3.1. Распределенная система промежуточного уровня.

- модель программного обеспечения промежуточного уровня основанная на удаленных вызовах процедур – RPC (Remote Procedure Calls)
  В этой модели акцент делается на сокрытие сетевого обмена за счет того, что процессу разрешается вызывать процедуры, реализация которых находится на удаленной машине.
- модель распределенных объектов
  Если в этой модели вызов процедуры проходит через границы машины, он может быть представлен в виде прозрачного обращения к объекту, находящемуся на удаленной машине. Распределенные объекты часто реализуются путем размещения объекта на одной из машин и открытия доступа к его интерфейсу с множества других.
- модель распределенных документов
  Одна из наиболее простых и распространенных, применяется в таких системах как WWW, Lotus и др.

Таблица 3.1. Сравнение распределенных систем

Характеристика	**Распределенная ОС**		**Сетевая ОС**	**Распределенная система промежуточного уровня**
	Мультипроцессорная	**Мультикомпьютерная**		
Степень прозрачности	Очень высокая	Высокая	Низкая	Высокая
Идентичность ОС на всех узлах	Поддерживается	Поддерживается	Не поддерживается	Не поддерживается
Число копий ОС	1	N	N	N
Коммуникации на основе	Совместно используемой памяти	Сообщений	Файлов	В зависимости от модели
Управление ресурсами	Глобальное, централизованное	Глобальное, распределенное	Отдельно на узле	Отдельно на узле
Масштабируемость	Отсутствует	Умеренная	Да	В зависимости от модели
Открытость	Закрытая	Закрытая	Открытая	Открытая

3.2Задачи распределенных систем

3.2.1Соединение пользователей с ресурсами

Основная задача распределенной системы – облегчить пользователям доступ к удаленным ресурсам и обеспечить их совместное использование, регулируя этот процесс.

3.2.2Прозрачность

Эта задача состоит в том, чтобы скрыть от пользователей тот факт, что процессы и ресурсы физически распределены на множестве компьютеров. Соответственно распределенные системы, которые представляются пользователям и пользовательскому программному обеспечению в виде единой компьютерной системы, называются *прозрачными* (transparent).

Хотя распределенная система должна маскировать от пользователя ресурсы, полное сокрытие иногда не представляется возможным, либо нежелательно. Достижение прозрачности распределения – это разумная цель при проектировании и разработке РС, однако она не должна рассматриваться в отрыве от других характеристик системы.

3.2.3Открытость

Открытая распределенная система – это система, предлагающая службы, вызов которых требует стандартный синтаксис и семантику. В распределенных системах службы обычно определяются через интерфейсы, для которых точно определены имена доступных функций, типы параметров, возвращаемых значений, исключительные ситуации и т.д. Наиболее сложно описать то, что делает эта служба, т.е. семантику интерфейсов. Обычно она описывается неформально.

3.2.4Масштабируемость

Масштабируемость может измеряться по трем параметрам:

- по отношению к размеру распределенной системы, что означает легкость подключения новых пользователей и ресурсов;
- географически, т.е. когда пользователи и ресурсы разнесены в пространстве;
- в административном смысле, т.е. простота в управлении во множестве административно независимых организаций.

Существуют три основные технологии масштабирования:

1. Сокрытие времени ожидания - нужно постараться по возможности избегать ожидания ответа на запрос. Фактически это означает, что желательно использование полностью асинхронной связи.
   В комплексе с этим, можно перенести часть функций сервера на клиент, что позволит уменьшить время передачи и обработки (например java-апплет).
2. Распределение – предполагает разбиение компонентов на мелкие части и последующее разнесение этих частей по системе. В качестве примера можно привести распределенную систему DNS.
3. Репликация – для решения проблем масштабирования, связанных с падением производительности. При использовании репликации появляется дополнительная проблема *непротиворечивости*, поскольку изменение в одной копии требует изменений в остальных. При этом может понадобится вносить изменения в копии строго по порядку производимых изменений, что предполагает механизмы глобальной синхронизации, реализовать на практике которую часто представляется невозможно.

3.3Синхронизация в распределенных системах

3.3.1Синхронизация времени

В централизованных системах время определяется однозначно. Для получения текущего времени применяется системный вызов, при этом последующий вызов не может вернуть значение меньше, чем предыдущий. В распределенных системах синхронизация по времени значительно сложнее.

Существует несколько алгоритмов синхронизации по времени в распределенных системах. Ниже представлены наиболее распространенные из них:

- Алгоритм Кристиана. Наиболее простой алгоритм, тем не менее повсеместно применяемый для синхронизации локальных часов с удаленным сервером времени. Суть алгоритма заключается в том, что клиент посылает серверу времени запрос, на который он отвечает, передавая свое текущее время. При этом учитывается время, затраченное на передачу запроса, и получение результатов. Для этого алгоритма требуется сервер точного времени.
- Алгоритм Беркли. Для этого алгоритма на каждой машине устанавливается демон (сервис), который периодически опрашивает все машины в сети и устанавливает локальное время равное усредненному значению, полученному от других машин. В отличие от алгоритма Кристиана, сервер

точного времени здесь не нужен - система, при достаточно больших размерах сети, получается самосинхронизирующаяся.

В приведенных выше алгоритмах время переводить назад нельзя, т.к. это может повлиять на программы, которые требуют строгой синхронизации, поэтому, вместо перевода локальных часов назад, их “замедляют” на какое-то время.

Для распределенных систем, как правило, абсолютное время не имеет значения, однако последовательность действий процессов должна каким-то образом синхронизироваться. В таких случаях применяют так называемые логические часы.

В качестве примера логических часов можно привести *отметки времени Лампорта*.

Для синхронизации логических часов Лампорт определил отношение под названием “происходит раньше”. Выражение a–>b читается как “a происходит раньше b” и означает, что все процессы согласны с тем, что первым происходит событие a, а позже – b. Отношение “происходит раньше” исполняется в двух случаях:

- если a и b – события, происходящие в одном и том же процессе, и a происходит раньше чем b, то отношение a->b истинно
- Если a – событие отсылки сообщения одним процессом, а b – событие получения того же сообщения другим процессом, то отношение a->b также истинно. Сообщение не может быть получено раньше или в то же время, т.к. на отсылку требуется конечное ненулевое время.

Отношение “происходит раньше” - транзитивное, т.е. если a->b и b->c, то a->c. Если события x и y происходят в разных процессах, которые не обмениваются между собой сообщениями, то отношение x->y не истинно, так же как и y->x. Такие события называются параллельными (concurrent). Рассмотрим алгоритм подстройки часов на рис. 3.2, который основывается на этих правилах.

В системах, где два события не могут протекать одновременно, используются следующие простые правила: к текущему времени мы можем добавить номер процесса и отделить его от целой части десятичной точкой (40.1, 50.25 и т.д.).

Используя этот способ получаем возможность указания времени для всех событий в распределенной системе, которые попадают под условия:

- если a происходит раньше b в том же процессе, то C(a)<C(b);
- если a и b – отправка и получение сообщения, то C(a)<C(b);
- для всех различимых событий a и b выполняется соотношение C(a)≠C(b).

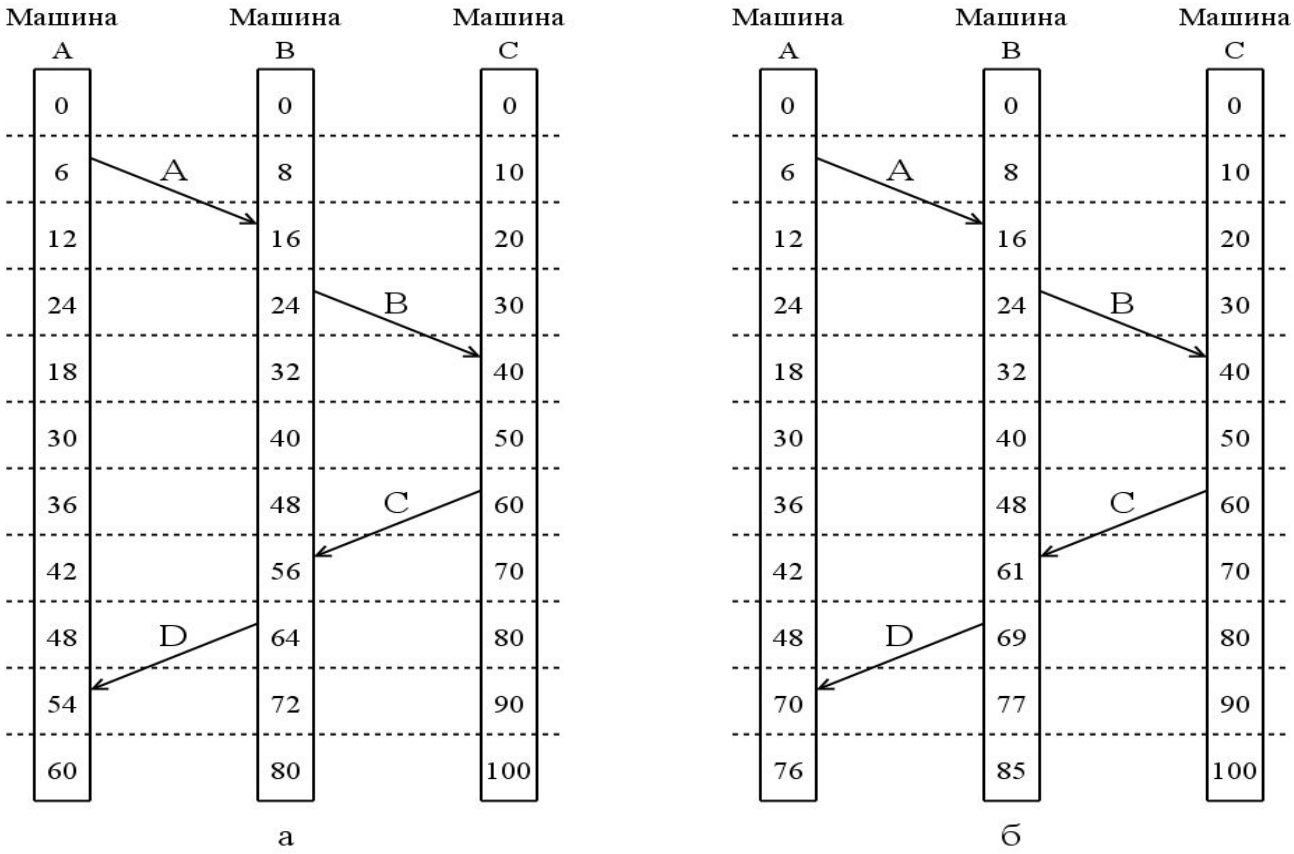


Рис.3.2. Три процесса, каждый с собственными часами, которые ходят с разной скоростью (а). Подстройка часов по алгоритму Лампорта (б)

3.3.2Алгоритмы голосования

Многие распределенные алгоритмы требуют, чтобы один из процессов был координатором, инициатором или выполнял другую роль. Не важно какой именно это будет процесс, главное, чтобы он существовал.

Если все процессы одинаковы, то способа выбрать один из них не существует. Поэтому применяются различные системы голосования. При этом считаем, что у них есть какие-либо уникальные идентификаторы и они не знают, какие процессы работают, а какие – нет.

Алгоритм забияки.

Когда один из процессов замечает, что координатор не отвечает, он инициирует голосование:

1. Процесс рассылает сообщение для всех процессов с большим, чем у него номером.
2. Если никто не отвечает, то процесс выигрывает голосование и становится координатором, если же отвечает, то работа текущего процесса на этом закончена

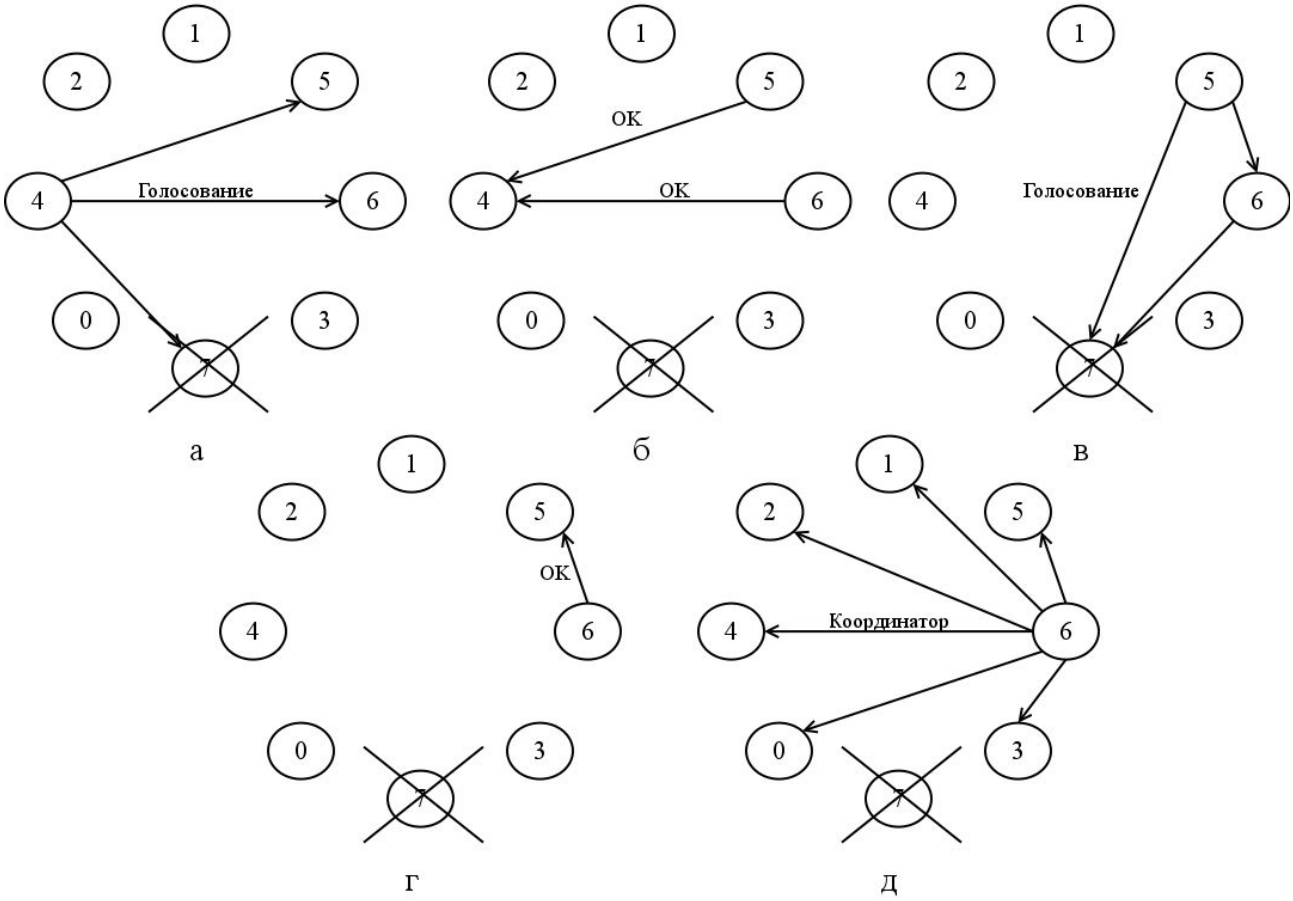


Рис.3.3. Голосование по алгоритму забияки.

Кольцевой алгоритм.

1. Когда один из процессов обнаруживает, что координатор не работает, он запускает по кольцу сообщение ГОЛОСОВАНИЕ**.**
2. каждый процесс добавляет к сообщению свой номер и пересылает дальше по кольцу адресату, способному принять сообщение.
3. начальный процесс получает маркер назад и вычисляет нового координатора, после чего запускает еще одно сообщение КООРДИНАТОР, в котором объявляется координатор.

На рис. 3.4 показана ситуация, когда 2 процесса (2-й и 5-й) одновременно обнаруживают, что координатор вышел из строя.

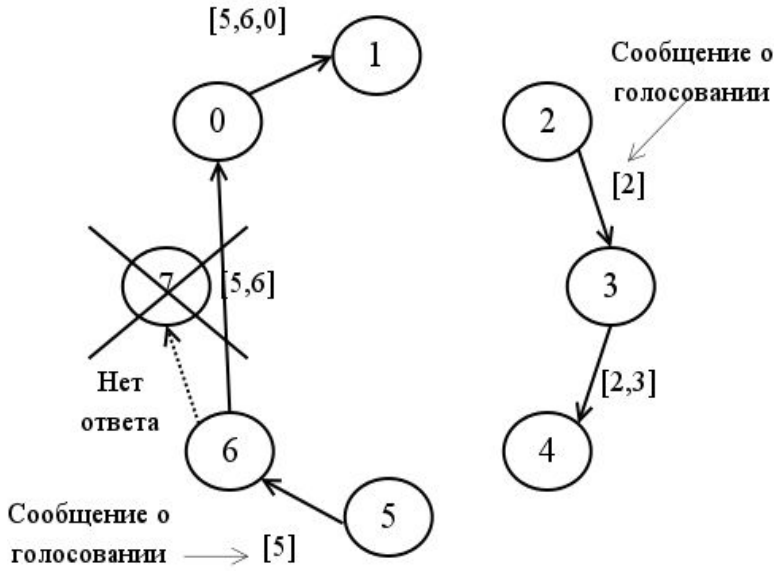


Рис.3.4. Голосование по кольцевому алгоритму.

3.3.3 Взаимное исключение

Системы, состоящие из множества процессов, проще всего программировать используя критические области. В обычных операционных системах такие взаимные исключения регулируются с помощью семафоров, мониторов и других конструкций подобного рода. Для распределенных систем существует множество алгоритмов, призванных реализовать такие конструкции, однако практически все они базируются на нескольких алгоритмах.

Централизованный алгоритм

Наиболее простой способ организации взаимных исключений в распределенной системе состоит в том, чтобы использовать методы, принятые в однопроцессорных системах. Один из процессов выбирается координатором. Процесс не являющийся координатором, при входе в критическую секцию сначала уведомляет координатора, сообщая в какую критическую область он собираются войти, и спрашивает на это разрешение. Если критическая область не используется другим процессом, то координатор разрешает ее использовать, если же используется – запрещает. При этом конкретные способы разрешения/запрета зависят от используемой системы.

Распределенный алгоритм

В этом алгоритме требуется наличие полной упорядоченности событий в системе (один из способов – алгоритм Лампорта). Алгоритм работает следующим образом:

1. Когда процесс собирается войти в критическую область, он создает сообщение, в котором указывает свой номер, имя критической области и текущее время. Затем это сообщение рассылается всем процессам.
2. При получении процессом сообщения с запросом алгоритм делится на 3 варианта:

- если получатель не находится в критической области и не собирается ее

использовать, то отсылает сообщение `OK;`
- если получатель находится в критической области, то не отвечает, а помещает запрос в очередь;
- если получатель собирается войти в критическую область, но еще не сделал этого, он сравнивает метку времени пришедшего сообщения с меткой времени своего сообщения. Выигрывает минимальное. Если пришедшее сообщение имеет меньший номер, то получатель отвечает OK. Если же метка времени меньше на своем сообщении, то пришедшее сообщение ставится в очередь, при этом ничего не отсылается.

Алгоритм маркерного кольца

Создается логическое кольцо, в котором каждому процессу назначен свой логический номер.

При инициализации кольца процесс 0 получает маркер (токен). Маркер циркулирует по кольцу. Когда процесс получает маркер, он проверяет, не нужно ли ему войти в критическую область. Если нужно, то он входит в критическую область, работает там и выходит. После этого маркер передается дальше. Вход в другую критическую область, используя один и тот же маркер запрещен.

Таблица 3.2. Сравнение алгоритмов взаимного исключения

Алгоритм	**Число сообщений на вход-выход**	**Задержки перед входом (в сообщениях)**	**Возможные проблемы**
Централизованный	3	2	Сбой координатора
Распределенный	2*(n-1)	2*(n-1)	Сбой в одном из процессов
Маркерного кольца	От 1 до ∞	От 1 до n-1	Потеря маркера, сбой в одном из процессов

3.4Задание

Создать распределенную сеть согласно варианту. Количество узлов в сети может динамически изменяться. Передача сообщений между узлами происходит с использованием протокола TCP; для поиска и добавления новых узлов используется режим широковещательных рассылок протокола UDP.

В задачу координатора входит собирать статистику от узлов распределенной системы и передавать данные по запросу клиента.

Клиент находит координатора с помощью широковещательного запроса.

	Алгоритм выбора координатора	**Функция узла сети**
1	Алгоритм забияки	Ведение статистики количества запущенных процессов на хост-системе
2	Алгоритм забияки	Ведение статистики использования памяти на хост-системе
3	Алгоритм забияки	Ведение статистики использования дисковой подсистемы на хост-системе
4	Алгоритм забияки	Ведение статистики использования центрального процессора на хост-системе
5	Кольцевой алгоритм	Ведение статистики количества запущенных процессов на хост-системе
6	Кольцевой алгоритм	Ведение статистики использования памяти на хост-системе
7	Кольцевой алгоритм	Ведение статистики использования дисковой подсистемы на хост-системе
8	Кольцевой алгоритм	Ведение статистики использования центрального процессора на хост-системе

Лабораторная работа №4
ИЗУЧЕНИЕ ВЫСОКОУРОВНЕВЫХ СРЕДСТВ СВЯЗИ РАСПРЕДЕЛЕННЫХ СИСТЕМ НА ПРИМЕРЕ ТЕХНОЛОГИИ MPI

Тема: Изучить технологию MPI – интерфейс передачи сообщений. Научиться практически использовать связь в распределенных системах с использованием сообщений.
Длительность: 4 часа.

4.1Введение

Предварительные предложения, известные как MPI-1, были сделаны Dongarra, Hempel, Hey, Walker в ноябре 1992 года и после ревизии эта версия была завершена в феврале 1993 года. Стандарт MPI-1 включал главное из того, что на рабочем совещании в Вильямсбурге было признано необходимым иметь в стандарте передачи сообщений. MPI-1 первоначально был в основном сфокусирован на парных обменах (point-to-point communications). MPI-1 поднял много важных проблем стандартизации, но не включал никаких процедур для коллективных обменов и не поддерживал обработку потоков.

В ноябре 1992 года в Минеаполисе состоялась встреча рабочей группы по MPI, на которой было решено поставить процесс стандартизации на более формальную основу и вообще адаптировать процедуры и организацию the High Performance Fortran Forum. На встрече были сформированы подкомитеты по главным компонентам сферы стандартизации и для каждого подкомитета был установлен дискуссионный почтовый сервис. В дополнение к этому, срок представления проекта стандарта MPI был установлен на осень 1993 года.

Чтобы достичь этой цели, рабочая группа MPI встречалась каждые 6 недель на два дня первые 9 месяцев 1993 года и представила проект стандарта на конференции ``Supercomputing 93'' в ноябре 1993 года. Эти встречи и дискуссии по почте вместе создали “MPI Форум”, членство в котором стало открытым для всех членов сообщества высоких компьютерных технологий.
Главное преимущество создания стандарта передачи сообщений состоит в его мобильности (portability) и простоте использования. В коммуникационной среде с распределенной памятью, в которой высший уровень процедур и/или абстракций построен над слоем процедур передачи сообщений, выгода стандартизации особенно очевидна. Более того, определение стандарта обеспечивает производителей четко определенным набором процедур, которые они могут эффективно реализовать или в некоторых случаях обеспечить аппаратную поддержку для них, увеличивая тем самым масштабируемость.

Целью MPI является создание широко используемого стандарта для написания программ на основе передачи сообщений. Следовательно, интерфейс должен быть практичным, мобильным, эффективным и гибким стандартом для передачи сообщений.

Полный список целей следующий:

- Разработать прикладной программный интерфейс (не обязательно для

компиляторов или библиотеки системной реализации);
- Обеспечить эффективный обмен, который позволял бы избегать копирования память-память, допускал совмещение операций обмена с вычислениями и разгружал коммуникационный сопроцессор, где это возможно;
- Обеспечить удобные для интерфейса описания аргументов для языков Си и ФОРТРАН77;
- Обеспечить надежный коммуникационный интерфейс: с неисправностями должен справляться не пользователь, а нижележащие слои коммуникационной подсистемы;
- Определить интерфейс, который не слишком отличается от уже существующих, таких как PVM, NX, Express, p4 и так далее и обеспечить расширения, которые допускают большую гибкость;
- Определить интерфейс, который можно реализовать на большинстве платформ поставщиков без существенных изменений в нижележащей коммуникации и системном программном обеспечении;
- Семантика интерфейса не должна зависеть от используемого языка программирования;
- Интерфейс должен быть спроектирован так, чтобы позволять безопасное выполнение потоков.

В стандарте MPI описаны:
- Парные обмены (point-to-point communication);
- Коллективные операции (collective operations);
- Группы процессов (process groups);
- Коммуникационные контексты (communication contexts);
- Топологии процессов (process topologies);
- Привязки для языков ФОРТРАН77 и Си;
- Управление вычислительной средой и анализ (environmental management and inquiry);
- Интерфейс профилирования (profiling interface).

Стандарт не описывает:
- Явные операции с разделяемой памятью;
- Операции, которые требуют большей поддержки операционной системы, чем есть в существующем стандарте; например, прием с прерыванием, удаленное исполнение или активные сообщения;
- Пакеты для конструирования программ;
- Возможности отладки;
- Явную поддержку потоков;
- Поддержку для управления задачами;
- I/O функции.

В марте 1995 года образовался Форум MPI-2. Первым действием Форума было исправление ошибок и разъяснение множества проблем, из оригинала стандарта MPI-1.0, выпущенного в июле 1994. В мае 1995 года была выпущена незначительная модификация этого документа, объединенная в расширение

стандарта MPI-1.1.

В 1999 году было предложено следующее расширение стандарта – MPI-2, в котором кроме расширения стандартных для MPI областей были добавлены три совершенно новые: параллельный ввод - вывод, операции с удаленной памятью, и динамическое управление процессами. Кроме того были добавлены привязки к языку C++.

4.2Соответствия типов данных

В стандарте MPI определены свои собственные типы данных. Это сделано для того, чтобы не привязываться к определенной аппаратной платформе. Например тип int для одних машин равен двум байтам, а для других – четырем, поэтому при передача такого типа между машинами с разными платформами может привести к потере данных, их искажению и даже к “зависанию” программы.

Использование типов данных MPI позволяет обойти эту проблему за счет того, что приведение типов данных для разных аппаратных платформ возлагается на реализацию MPI, и программисту не приходится заботиться об этой проблеме. Соответствия типов данных MPI и типов данных языка C приведены в таблице 4.1.

Тип данных MPI	***Тип данных C***
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	нет
MPI_PACKED	нет

Таблица 4.1. Соответствие типов данных MPI и C

4.3 Структура программ MPI

Структура программы MPI очень похожа на структуру программ для обычных однопроцессорных систем с использованием параллельных процессов или потоков. И в этом нет ничего удивительного, т.к. одной из целей систем промежуточного уровня, к которым можно отнести и MPI, является сокрытие реального аппаратного обеспечения. Поэтому все процессы выполняются в “виртуальной” машине, которая связывает узлы

распределенной системы с помощью сообщений.

Каждая программа MPI содержит директиву препроцессора

```
#include <mpi.h>
```

Файл `mpi.h` содержит определения, макроопределения и прототипы функций, необходимых для компиляции программ MPI. Прежде чем вызывать любые другие функции MPI, нужно однократно вызвать функцию `MPI_Init`. Ее аргументы - это указатели на параметры функции `main` - `argc` и `argv`. Они позволяют системе выполнять любую специальную настройку, чтобы использовать библиотеку MPI. После того, как программа, использующая библиотеку MPI, закончилась, необходимо вызвать `MPI_Finalize`. Эта функция завершает все незавершенные действия MPI - например, ожидание передач, которые никогда не закончатся. Типичная программа MPI имеет следующую структуру:

```
#include <mpi.h>
. . . .
main(int argc, char **argv){
. . . .
/* Функции MPI нельзя вызывать до этого момента*/
MPI_Init(&argc, &argv);
. . . .
MPI_Finalize();
/* Функции MPI нельзя вызывать после этого момента */

}/*main*/
```

Вызовы `MPI_Init` и `MPI_Finalize` для языка С выглядят следующим образом:

```
int MPI_Init(int *argc, char ***argv)
int MPI_Finalize()
```

MPI поддерживает выделение нескольких процессов в отдельные группы, внутри которых передача сообщений осуществляется с помощью указания коммуникатора. *Коммуникатор* – это механизм, который обеспечивает рассмотрение подмножества процессов MPI в виде коммуникационной области. Изначально все процессы принадлежат общему предопределенному коммуникатору `MPI_COMM_WORLD`.

Для определения ранга (номера) процесса в коммуникаторе используется вызов `MPI_Comm_rank`, а для определения количества процессов:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Здесь следует пояснить, что все вызовы функций MPI стандартизованы.

Это выражается в общем подходе к названию вызовов – все вызовы начинаются с префикса MPI_ , и передаче аргументов, которые делятся на 3 типа – IN, OUT и IN/OUT для передачи значений в функцию, для возвращаемых значений и значений, которые могут изменяться функцией, соответственно. При успешном вызове вызов возвращает MPI_SUCCESS, при ошибке возвращаемое значение зависит от типа ошибки.

Для передачи сообщений от одного процесса к другому используются вызов MPI_Send, а также его варианты для асинхронной, буферизированной и передачи по готовности. Для приема сообщений используется вызов MPI_Recv:

```
int MPI_Send(void* message, int count,
           MPI_Datatype datatype, int dest, int tag,
           MPI_Comm comm);
int MPI_Recv(void* message, int count,
           MPI_Datatype datatype, int source, int tag,
           MPI_Comm comm, MPI_Status* status).
```

Чтобы сообщение было успешно передано, система должна добавить немного информации к данным, которые желает передать прикладная программа. Эта дополнительная информация формирует конверт сообщения. В MPI конверт содержит следующую информацию:

- ранг получателя;
- ранг отправителя;
- тэг сообщения;
- коммуникатор.

Эти детали могут использоваться приемником, чтобы различать поступающие сообщения. Аргумент source используется, чтобы отличить сообщения, полученные от разных процессов. Тэг представляет собой указанное пользователем значение int, которое используется, чтобы отличить сообщения, полученные от одного процесса.

Таким образом простейшая программа с использованием технологии MPI будет выглядеть следующим образом:

```
#include "mpi.h"
main(argc, argv)
int argc;
char **argv;
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0)    /* код для процесса 0 */
    {
        strcpy(message,"Hello, there");
        MPI_Send(message, strlen(message), MPI_CHAR, 1,
                   99, MPI_COMM_WORLD);
```

```
    }
    else                    /* код для процесса 1 */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99,
                    MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

При программировании часто возникает ситуация, когда нужно убедиться в том, что все процессы достигли какой-то определенной точки выполнения в программе. Это можно сделать и с помощью сообщений, однако в этом случае очень легко может возникнуть взаимная блокировка процессов. Для такой, общей внутри одного коммуникатора, синхронизации используется функция `MPI_Barrier`, которая блокирует процесс до тех пор, пока все процессы не достигнут вызова этой функции:

```
int MPI_Barrier ( MPI_Comm comm ).
```

Возможность выделения отдельных процессов в группы позволяет использовать коллективное взаимодействие между процессами. Коллективная операция исполняется путем вызова всеми процессами в группе коммуникационных функций с соответствующими аргументами. Синтаксис и семантика коллективных операций определяются подобно синтаксису и семантике операций парного обмена между процессами. Это в частности означает, что в коллективных операциях используются основные типы данных и они должны должны совпадать у процесса-отправителя и процесса-получателя. Один из ключевых аргументов - это коммуникатор, который определяет группу участвующих в обмене процессов и обеспечивает контекст для этой операции.

К операциям коллективного обмена относятся:

- барьерная синхронизация всех процессов группы;
- широковещательная передача (broadcast) от одного процесса всем остальным процессам группы;
- сбор данных из всех процессов группы в один процесс;
- рассылка данных одним процессом группы всем процессам группы;
- вариант сбора данных, когда все процессы группы получают результат;
- раздача / сбор данных из всех процессов группы всем процессам группы;
- глобальные операции редукции, такие как сложение, нахождение максимума, минимума, или определённые пользователем функции, где результат возвращается всем процессам группы или в один процесс;
- составная операция редукции и раздачи;
- префиксная операция редукции, при которой в процессе i появляется результат редукции 0,1, ..., i, $i < n$, где n - число процессов в группе.

Различные коллективные операции, такие как широковещание и сбор данных, имеют единственный процесс-отправитель или процесс-получатель. Такие процессы называются корневыми (root). Некоторые аргументы в коллективных функциях определены как “существенные только для корневого процесса” и игнорируются для всех других участников операции.

4.4 Запуск пользовательских программ MPI

В настоящее время существует множество реализаций MPI, большая часть из которых является платной и привязанной к определенной платформе. Однако существует два открытых и свободных проекта по реализации стандарта MPI – это MPICH и LAM/MPI. Для проведения лабораторных работ можно использовать любой из них.

На примере проекта LAM опишем запуск среды выполнения. Для инициализации кластерной системы нужно инициализировать демона `lamd` на каждом узле системы: для этого используется программа `lamboot`. Если не указывать никаких параметров, то инициализация системы пройдет в соответствии с файлом конфигурации, который по умолчанию находится здесь: `/etc/lam/lam-bhost.def`.

Однако для пользовательских конфигураций можно указывать другой файл конфигурации в качестве параметра командной строки:

```
lamboot mybhost
```

В файле конфигурации построчно задается список машин на которых будет запускаться среда выполнения:

```
Amox.evm        cpu=2   user=lam
```

где первая запись обозначает адрес машины, вторая – количество процессоров на ней и третья – пользователь, от имени которого будет запущен демон `lamd`, причем обязательным является только имя машины.

Для того, чтобы узнать сколько узлов и процессоров задействовано для текущей среды выполнения, нужно запустить программу `lamnodes`.

Именами компиляторов для MPI-программ являются mpicc (для языка C), mpiCC и mpic++ (для языка C++) и mpif77 (для программ на Fortran). Пример:

```
mpicc -g -c foo.c
mpicc -g -c bar.c
mpicc -g foo.o bar.o -o my mpi program
```

Отметим, что для корректной компиляции и линковки MPI не требуются дополнительные флаги компиляторов и линкеров. В результате `my_mpi_program` готова к выполнению в рабочей среде LAM.

Для запуска параллельных программ стандартом MPI рекомендовано использовать программу `mpiexec`:

```
mpiexec <global_options> <cmd>
```

где `<global_options>` - глобальные опции, воздействующие на все запускаемые команды, а `<cmd>` - имя запускаемого файла вместе с параметрами. Для получения полного списка глобальных опций можно

использовать команду `man mpiexec` , но для выполнения лабораторной работы может понадобиться лишь один из них: `-n`, указывающий количество запускаемых копий программы, например так будет выглядеть запуск 3-х копий программы `testmpi`:

```
mpiexec -n 3 testmpi
```

После окончания работы с текущей средой выполнения, она должна корректно завершаться с помощью команды `lamhalt`. Однако, при некоторых, достаточно редких условиях, `lamhalt` может отработать неудачно, например, когда любая из станций из среды выполнения LAM “разрушилась” еще до запуска `lamhalt`. В таких случаях нужно использовать команду `wipe` для гарантированного закрытия среды выполнения LAM:

```
wipe -v mybhost
```

где `mybhost` – это файл конфигурации, используемый при загрузке.

4.5 Задание

С помощью технологии MPI создать распределенное приложение, указанное в задании.

	Алгоритм
1	Умножение матриц с использованием ленточного алгоритма.
2	Умножение матриц с использованием алгоритма Фокса.
3	Умножение матриц с использованием алгоритма Кеннона.
4	Вычисление выражения (a+b)^n, с использованием формулы бинома Ньютона.
5	Быстрое преобразование Фурье.
6	Преобразование Уолша.
7	Преобразование Хаара.
8	Сеть Хопфилда.

5. **Литература**

«Компьютерные сети» 4-е издание, Э. Тенненбаум – СПб.: Питер, 2003. – 992 с.: ил. – (Серия «Классика Computer Science»).

Распределенные системы. Принципы и парадигмы / Э. Тенненбаум, М. Ван Стеен. – Спб: Питер,2003. – 877 с.: ил. – (Серия «Классика Computer Science»).

Сети TCP/IP, том 3. Разработка приложений типа клиент/сервер для Linux/POSIX.: Пер. с англ. – М.: издательский дом «Вильямс», 2002. – 592 с.: ил. – Парал. тит. англ.

Средства параллельного программирования в ОС Linux: Учебное пособие / под ред. Р.Х. Садыхова. – Мн.: ЕГУ, 2004. – 476 с.

J. Poster, J. Reynolds, Network Working Group Request for Comments: 959 «File Transfer Protocol (FTP)», ISI, October 1985.