Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра автоматизированных систем управления (АСУ)

А.Я. Суханов

Сети и телекоммуникации

Учебное пособие по выполнению лабораторных работ для студентов направления бакалавриата 09.03.01

2023

Суханов А.Я.

Сети и телекоммуникации: Учебное пособие по лабораторным работам для студентов направления бакалавриата 09.03.01, – 120 с.

Учебное методическое пособие содержит программу для проведения лабораторных занятий, перечень контрольных вопросов.

ОГЛАВЛЕНИЕ

Введение

Сети и телекоммуникации это обширная дисциплина изучающая взаимодействие удаленных абонентов в сети, включающая изучение как прикладных уровней взаимодействия, так и физических. При выполнении лабораторных работ упор делается на изучения прикладных и транспортных уровней взаимодействия сети интернет. Но приведенные в конце теоретические вопросы охватывают также физический, канальный и сетевой уровни.

1 Лабораторная работа №1 Работа с Socket

1.1 Определения

Socket (гнездо, разъем) - абстрактное программное понятие, используемое для обозначения в прикладной программе конечной точки канала связи с коммуникационной средой, образованной вычислительной сетью. При использовании протоколов TCP/IP можно говорить, что socket является средством подключения прикладной программы к порту локального узла сети.

1.2 Создание сервера

Для создания сокета в операционной системе служит системный вызов socket(). Для транспортных протоколов семейства TCP/IP существует два вида сокетов:

UDP-сокет – сокет для работы с датаграммами, и TCP сокет – для работы с каналами. Соответственно в стеке TCP/IP протокол TCP отвечает за надежную передачу потока данных, реализует он эту надежность за счет повторной отправки данных, на которые не пришли подтверждения за выделенное специальным таймером время, протокол UDP позволяет отправить дейтаграммы, но надежность передачи не обеспечивает, то есть дейтаграммы могут и не дойти.

При создании сокета необходимо точно специфицировать его тип. Эта спецификация производится с помощью трех параметров вызова socket(). Первый параметр указывает, к какому семейству протоколов относится создаваемый сокет, а второй и третий параметры определяют конкретный протокол внутри данного семейства.

Создание сокета осуществляется следующим системным вызовом (Unix):

Определение функции:

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

Аргумент domain задает используемый для взаимодействия набор протоколов (вид коммуникационной области), для стека протоколов TCP/IP он должен иметь символьное значение AF_INET или PF_INET.

Аргумент type задает режим взаимодействия:

SOCK_STREAM - с установлением соединения;

SOCK_DGRAM - без установления соединения.

Аргумент protocol задает конкретный протокол транспортного уровня (из нескольких возможных в стеке протоколов). Если этот аргумент задан равным 0, то будет

использован протокол "по умолчанию" (TCP для SOCK_STREAM и UDP для SOCK_DGRAM при использовании комплекта протоколов TCP/IP).

При удачном завершении своей работы данная функция возвращает дескриптор сокета - целое неотрицательное число, однозначно его идентифицирующее.

При обнаружении ошибки в ходе своей работы функция возвращает число "-1".

Далее текст кода для создания программы сервера или клиента будет отмечен цветом.

Указанный сокет можно соответственно подключив следующие библиотеки (POSIX):

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

Либо, в операционной системе Windows:

```
#include <winsock2.h>
```

и соответственно подключить библиотеку ws2_32.lib.

Прежде чем воспользоваться функцией socket необходимо проинициализировать процесс библиотеки wsock32.dll вызвав функцию WSAStartup например:

```
WSADATA WsaData;
int err = WSAStartup (0x0101, &WsaData);
if (err == SOCKET_ERROR) { printf ("WSAStartup() failed:
%ld\n", GetLastError ());
return 1;
}
```

Здесь 0x0101 версия библиотеки которую следует использовать.

Теперь объявить переменную типа SOCKET можно следующим образом:

Пример вызова:

```
int s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

Далее следует задать параметры для сокета (сервера) для этого нам необходимо объявить структуру SOCKADDR_IN sin примерно следующим образом:

```
SOCKADDR_IN sin;// или struct sockaddr_in sin;
sin.sin_family = AF_INET;
sin.sin_port = htons(80);
```

//преобразование из хостового в сетевой формат слова (см. также ntohs, ntohl, htonl – для одинакового восприятия числовых данных на всех архитектурах)

sin.sin_addr.s_addr = INADDR_ANY;

Изучите так же функцию gethostbyname.

Структура SOCKADDR_IN используется несколькими системными вызовами и функциями socket-интерфейса, ее определение в файле in.h выглядит следующим образом:

struct SOCKADDR_IN { short sin_family;

u_short sin_port;

struct in_addr sin_addr;

char sin_zero[8];

};

Поле sin_family определяет используемый формат адреса (набор протоколов), в нашем случае (для TCP/IP) оно должно иметь значение AF_INET.

Поле sin_addr содержит адрес (номер) узла сети.

Поле sin_port содержит номер порта на узле сети.

Поле sin_zero не используется.

Определение структуры in_addr (из того же файла in.h) таково:

struct in_addr { union { u_long S_addr;

/* другие (не интересующие нас) члены объединения */ } S_un;

#define s_addr S_un.S_addr };

Структура SOCKADDR_IN должна быть полностью заполнена перед выдачей системного вызова bind. При этом, если поле sin_addr.s_addr имеет значение INADDR_ANY, то системный вызов будет привязывать к сокету с номером (адресом) локального узла сети.

Для подключения сокета к коммуникационной среде, образованной вычислительной сетью, необходимо выполнить системный вызов bind, определяющий в принятом для сети формате локальный адрес канала связи со средой. В сетях TCP/IP socket связывается с локальным портом.

Системный вызов bind имеет следующий синтаксис:

int bind (SOCKET s, SOCKADDR_IN *addr, int addrlen)

Пример:

err = bind(s, (LPSOCKADDR)&sin, sizeof(sin)); // или int err = bind(s, (const struct sockaddr *)&sin, sizeof(sin));

Аргумент s задает дескриптор связываемого сокета.

Аргумент addr в общем случае должен указывать на структуру данных,

содержащую локальный адрес, приписываемый сокету. Для сетей TCP/IP такой структурой является SOCKADDR_IN.

Аргумент addrlen задает размер (в байтах) структуры данных, указываемой аргументом addr.

В случае успеха bind возвращает 0, в противном случае - "-1".

Для установления связи "клиент-сервер" используются системные вызовы listen и accept (на стороне сервера), а также connect (на стороне клиента). Для заполнения полей структуры sockaddr_in, используемой в вызове connect, обычно используется библиотечная функция gethostbyname, транслирующая символическое имя узла сети в его номер (адрес).

Системный вызов listen выражает желание выдавшей его программы- сервера ожидать запросы к ней от программ-клиентов и имеет следующий вид:

int listen (SOCKET s, int n);

Пример:

```
err = listen(s, SOMAXCONN);
```

Аргумент s задает дескриптор сокета, через который программа будет ожидать запросы к ней от клиентов. Socket должен быть предварительно создан системным вызовом socket и обеспечен адресом с помощью системного вызова bind.

Аргумент n определяет максимальную длину очереди входящих запросов на установление связи. Если какой-либо клиент выдаст запрос на установление связи при полной очереди, то этот запрос будет отвергнут.

Признаком удачного завершения системного вызова listen служит нулевой код возврата.

Перед тем как воспользоваться функцией accept, необходимо объявить ещё одну переменную типа SOCKET, например s1 .

```
SOCKADDR_IN from;
int fromlen=sizeof(from);
s1 = accept(s,(struct sockaddr*)&from,&fromlen);
```

Это сделано для того, что бы узнать IP адрес и порт удаленного компьютера.

Для приема запросов от программ-клиентов на установление связи в программах-серверах используется системный вызов accept, имеющий следующий прототип:

int accept (SOCKET s, sockaddr_in *addr, int *p_addrlen;

Аргумент s задает дескриптор сокета, через который программа-сервер получила запрос на соединение (посредством системного запроса listen).

Аргумент addr должен указывать на область памяти, размер которой позволял бы разместить в ней структуру данных, содержащую адрес сокета программы-клиента,

сделавшей запрос на соединение. Никакой инициализации этой области не требуется.

Аргумент p_addrlen должен указывать на область памяти в виде целого числа, задающего размер (в байтах) области памяти, указываемой аргументом addr.

Системный вызов accept извлекает из очереди, организованной системным вызовом listen, первый запрос на соединение и возвращает дескриптор нового (автоматически созданного) сокета с теми же свойствами, что и socket, задаваемый аргументом s. Этот новый дескриптор необходимо использовать во всех последующих операциях обмена данными.

Если очередь запросов на момент выполнения accept пуста, то программа переходит в состояние ожидания поступления запросов от клиентов на неопределенное время (хотя такое поведение accept можно и изменить).

Признаком неудачного завершения accept служит отрицательное возвращенное значение (дескриптор сокета отрицательным быть не может). Обычно для облуживания сервером краткосрочного взаимодействия со множеством клиентов можно реализовать «бесконечный» цикл в котором извлекается новый клиент из очереди, идет взаимодействие с ним по приему и передаче данных и последующее отключение, если же взаимодействие требует времени, или какой-то длительности по обмену сообщениями, то обслуживание клиента реализуется в новом потоке (либо организуется работа с не блокирующими сокетами (асинхронными), изучите также функцию select).

После установления соединения с клиентом можно передавать и получать данные. Для этого в операционной системе Window используются системные вызовы recv для чтения и send для записи.

Системные вызовы recv и send имеют следующие прототипы:

int recv (SOCKET s, void *buf, size_t len, int flags);

int send (SOCKET s, const void *buf, size_t len, int flags);

Возвращаемое значение:

число принятых или переданных байтов в случае успеха или -1 в случае ошибки, аргумент s задает дескриптор сокета, через который принимаются данные. Ноль возвращается, например, в случае если все данные были отправлены, приняты и соединение было закрыто (послано SHUTDOWN_WRITE или послан FIN сегмент (сегмент с флагом FIN)). В случае если соединение держится, то, например, recv попадает в режим ожидания, блокируя текущий поток команд. Все это вполне логично, если данные не идут, но соединение держится, вполне возможно, что они могут появиться и нужно их дождаться, если же отправитель отправил FIN, то данных больше не будет и recv может вернуть 0.

Аргумент buf для вызова recv указывает на область памяти, предназначенную для размещения принимаемых данных, а для вызова send - область памяти, содержащая передаваемые данные.

Аргумент len задает размер (в байтах) области buf.

Аргумент flags зависит от системы, но и UNIX, и Windows поддерживают следующие флаги:

MSG_OOB – следует послать или принять срочные данные.

MSG_PEEK – используется для просмотра поступивших данных без их удаления из приемного буфера. После возврата из системного вызова данные еще могут быть получены при последующем вызове recv.

MSG_DONTROUTE – сообщает ядру, что не надо выполнять обычный алгоритм маршрутизации. Как правило, используется программами маршрутизации или для диагностических целей.

Примеры:

```
char buf[]= “Hello world”;
int sz = send(s, buf, sizeof(buf), 0);
```

При работе с протоколом TCP вам ничего больше не понадобиться. Но при работе с UDP нужны еще системные вызовы recvfrom и sendto. Они очень похожи на recv и send, но позволяют при отправке датаграммы задать адрес назначения, а при приеме – получить адрес источника.

Системные вызовы recvfrom и sendto имеют следующие прототипы:

int recvfrom (SOCKET s, void *buf, size_t len, int flags, struct socketaddt *from, int *fromlen);

int sendto (SOCKET s, const void *buf, size_t len, int flags, const struct sockaddr *to, int tolen);

Возвращаемое значение:

число принятых или переданных байтов в случае успеха или -1 в случае ошибки Первые четыре аргумента – s, buf, len и flags – такие же, как и в вызовах recv и send. Аргумент from в вызове recvfrom указывает на структуру, в которую ядро помещает адрес источника пришедшей датаграммы. Длина этого адреса хранится в целом числе, на которое указывает аргумент fromlen. Обратите внимание, что fromlen – это указатель на целое.

Аналогично аргумент to в вызове sendto указывает на адрес структуры, содержащей адреса назначения датаграммы, а аргумент tolen – длина этого адреса.

Заметьте, что to - это целое, а не указатель.

Для закрытия ранее созданного сокета в системе Windows используется системный вызов closesocket, а в UNIX – системный вызов close.

Прототипы системных вызовов close и closesocket имеют следующий вид:

int close(SOCKET s);

int closesocket(SOCKET s);

Аргумент s задает дескриптор ранее созданного сокета.

Таким образом, мы видим, что для сокетов с установлением соединения (например, протокол TCP) требуется вызов дополнительных функций: listen, accept, connect позволяющие обслужить подключение клиентов, их у сервера может быть несколько, очередь соединений определяется с помощью listen. Функция accept является блокирующей, так же как и recv, только в ожидании подключения клиента, при этом вызов этих двух функций для разных клиентов приведет к возможной взаимной блокировке, что, в общем-то, потребует создания многопоточного приложения, это обычный вариант, но при создании многопоточного приложения есть свои недостатки.

1.3 Создание клиента

Программа клиента делается аналогично до момента создания сокетов.

Создайте сокет так, как описано выше, но не пользуйтесь командой bind:

SOCKADDR_IN anAddr;

anAddr.sin_family = AF_INET;

anAddr.sin_port = htons(80);

anAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");

Заполнение структуры производится почти также но во время инициализации переменной anAddr необходимо указать IP-адрес сервера (пример 127.0.0.1) .

Для обращения программы-клиента к серверу с запросом на установление логической соединения используется системный вызов connect, имеющий следующий прототип:

int connect(SOCKET s, const struct sockaddr *peer, int peer_len));

Аргумент s задает дескриптор сокета, через который программа обращается к серверу с запросом на соединение.

Аргумент addr должен указывать на структуру данных, содержащую адрес, приписанный сокету программы-сервера, к которой делается запрос на соединение. Для сетей TCP/IP такой структурой является sockaddr_in.

Пример:

```
err = connect( s, (LPSOCKADDR)&anAddr, sizeof(sin) );
// или int err = bind( s, (const struct sockaddr *)&anAddr, sizeof(anAddr) );
```

Для формирования значений полей структуры sockaddr_in удобно использовать функцию gethostbyname.

Аргумент addrlen задает размер (в байтах) структуры данных, указываемой аргументом addr.

Для того, чтобы запрос на соединение был успешным, необходимо, по крайней мере, чтобы программа-сервер выполнила к этому моменту системный вызов listen для сокета с указанным адресом.

При успешном выполнении запроса системный вызов connect возвращает 0, в противном случае - "-1" (устанавливая код причины неуспеха в глобальной переменной errno).

Примечание. В режиме взаимодействия без установления соединения необходимости в выполнении системного вызова connect нет. Однако, его выполнение в таком режиме не является ошибкой - просто меняется смысл выполняемых при этом действий: устанавливается адрес "по умолчанию" для всех последующих посылок дейтаграмм.

1.4 Технологии select, poll, epoll.

Данные технологии предназначены позволить выполнять обслуживание приема и передачи данных через сокетные соединения и от других дескрипторов не используя многопоточность. Недостаток многопоточного приложения в условиях большого количества соединений от пользователей связан с тем, что операционная система, переключаясь между потоками, тратит время на сохранение, так называемого контекста исполнения (например, общие регистры процессора). Каждый раз при переключении нужно сохранить состояние, потом возвратившись на этот же поток восстановить его (например, в стеке), на это, безусловно, тратится время процессора, что при большом числе потоков приводит к тому, что основное время тратится на восстановление и сохранение контекста исполнения, а не на выполнение команд самого потока.

Все названные технологии позволяют заблокировать для опроса состояния сразу множество дескрипторов сокетов.

В технологиях select и poll указывается список опрашиваемых дескрипторов (фактическим массив структур). Затем после их вызова проверяется изменение структур, готовы ли они на чтение или запись, и так далее, если, например, какой-то дескриптор

готов к чтению, то можно вызвать функцию recv(), при этом она уже, очевидно, не заблокирует текущий поток, так как есть данные на чтение для данного сокета. При этом мы вынуждены пройти весь список дескрипторов и проверить его, включая и неизмененные, на что естественно тратится время, что обуславливает сложность алгоритма O(n). Отличие select и poll в том, что select ограничивает список опрашиваемых дескрипторов до 1024, каждый раз требует обнуления измененных вызовом структур и требует два массива структур на чтение и запись, у poll данных ограничений нет.

Технология epoll в отличие от двух выше рассмотренных возвращает список только измененных дескрипторов и можно пробежать циклом только по данному списку, что обуславливает сложность алгоритма O(1).

1.5 Пример использования соккетов на python

Потоковый TCP соккет.

Сервер

```
import socket
sock = socket.socket()
sock.bind(('', 9090))
sock.listen(1)
conn, addr = sock.accept()
print 'connected:', addr
while True:
    data = conn.recv(1024)
    if not data:
        break
    conn.send(data.upper())
conn.close()
```

Клиент

```
import socket
sock = socket.socket()
sock.connect(('localhost', 9090))
sock.send('hello, world!')
data = sock.recv(1024)
sock.close()
print data
```

Пример дейтаграммного соккета.

```
from socket import *
host = 'localhost'
port = 777
addr = (host,port)
#socket - функция создания сокета
#первый параметр socket_family может быть AF_INET или AF_UNIX
#второй параметр socket_type может быть SOCK_STREAM(для TCP) или SOCK_DGRAM(для UDP)
udp_socket = socket(AF_INET, SOCK_DGRAM)
#bind - связывает адрес и порт с сокетом
udp_socket.bind(addr)
#Бесконечный цикл работы программы
while True:
   #Если мы захотели выйти из программы
   question = input('Do you want to quit? y\\n: ')
   if question == 'y': break
   print('wait data...')
   #recvfrom - получает UDP сообщения
   conn, addr = udp_socket.recvfrom(1024)
   print('client addr: ', addr)
   #sendto - передача сообщения UDP
   udp_socket.sendto(b'message received by the server', addr)
udp_socket.close()
```

```
from socket import *
host = 'localhost'
port = 777
addr = (host,port)
#socket - функция создания сокета
#первый параметр socket_family может быть AF_INET или AF_UNIX
#второй параметр socket_type может быть SOCK_STREAM(для TCP) или SOCK_DGRAM(для UDP)
udp_socket = socket(AF_INET, SOCK_DGRAM)
```

```
#bind - связывает адрес и порт с сокетом
udp_socket.bind(addr)
#Бесконечный цикл работы программы
while True:
    #Если мы захотели выйти из программы
    question = input('Do you want to quit? y\\n: ')
    if question == 'y': break
    print('wait data...')
    #recvfrom - получает UDP сообщения
    conn, addr = udp_socket.recvfrom(1024)
    print('client addr: ', addr)
    #sendto - передача сообщения UDP
    udp_socket.sendto(b'message received by the server', addr)
udp_socket.close()
```

1.6 Задание на лабораторную работу

По указаниям из методички реализовать сначала сервер, затем клиент. На языке Си и на Python, обеспечить обмен сообщениями между клиентом и сервером.

Задания по вариантам. Int посылать не переводя в строку, использовать hton, ntoh.

1) Приложение клиент должно отправить два числа, одно в формате с плавающей запятой и другое в формате целого (int4) и получить от сервера сами числа и их сумму.

2) Отправить строку серверу, длинной не более 256 байт, вначале отправить длину строки в формате int4, от сервера получить половину этой строки.

3) Получать от сервера время в секундах в формате UNIX-времени.

4) Отправить на сервер массив целых чисел, отправить вначале размер массива, вернуть максимальное число, минимальное, сумму чисел массива.

5) Отправить на сервер строку, вернуть ее хэш и саму строку. Длины строк отправлять в формате int.

6) Получить от сервера структуру с годом, месяцем и днем, номером отправки.

7) Отправить на сервер две строки, получить их сумму.

8) Отправить на сервер два массива чисел, получить конкатенированный массив.

9) Отправить на сервер структуру с именем, отчеством, годом рождения,

номером группы, получить ответ с номером в списке по году рождения.

10) Получить от сервера по DNS адресу набор ip. Использовать gethostbyname.

11) Получить структуру с данными о хосте сервера.

12) Отобразить порты клиента и его ip адрес, вместе с ip адресом сервера. От сервера получить дату.

13) Получить от сервера произведение двух чисел в формате целого.

14) Получить от сервера остаток от деления и результат деления нацело от двух целых чисел.

15) Получить от сервера либо косинус, либо синус, либо тангенс числа. Что получить и число отправляет клиент.

16) Получить от сервера произведение целого и вещественного чисел посланных клиентом.

17) Вернуть корень из целого числа, либо квадрат целого числа в зависимости от запроса клиента.

18) Вернуть возведение в степень. Целую степень и число посылает сервер.

19) Вернуть все положительные числа в массиве посланным клиентом.

20) Получить от сервера результат побитового сложения, умножения, отрицания. Тип операции посылает клиент и либо два, либо одно число.

Контрольные вопросы.

Какие параметры содержит функция socket, что она возвращает, как назначается клиентский порт и нужен ли клиенту порт. Функция bind, функция accept, функция recv, send, connect. В чем отличие от UDP сокета. Что такое блокирующий и неблокирующий сокет. Отличие технологий select, poll и epoll.

1) Сервер послал клиенту 100 байт, на приемнике клиента стоит вызов функции

x1= recv(30,buf)

x2= recv(80,buf)

x3= recv(30,buf)

что вернется в каждом x, если сервер не закрывает соединение и если он его закроет послав SHUTDOWN_WRITE.

2) Как порт назначается клиенту, размер порта в байтах, из какого диапазона.

3) Что такое функции htons, ntohs, htonl, зачем они нужны.

4) Будет ли преимущество в использовании epoll по сравнению с пулом потоков.

5) Зачем нужна многопоточная обработка в сокетных приложениях.
6) Сколько занимает IP адрес байт.
7) Может ли send вернуть меньше чем указанный размер буфера.

2. Лабораторная работа №2 Протоколы SMTP и POP3

Лабораторная работа №2 выполняется после изучения материала, посвященного описанию принципов работы почтовых служб SMTP и POP3 [Компьютерные сети. / Э. Таненбаум, Коммпьютерные сети. / В. Олифер, Н. Олифер].

Цель работы:

написать GUI или консольное приложение для ОС Windows или Linux, реализующие работу протоколов SMTP или POP3 на стороне клиента.

Рекомендуемая литература:

Описание протокола SMTP в спецификации RFC-788 (https://www.ietf.org/rfc/rfc788.txt), перевод: https://rfc2.ru/5321.rfc (ESMTP).

Описание протокола POP3 в спецификации RFC-1939 (https://www.ietf.org/rfc/rfc1939.txt), перевод: https://rfc2.ru/1939.rfc.

Протокол электронной почты SMTP Протокол SMTP (Simple Mail Transfer Protocol) был разработан для обмена почтовыми сообщениями в сети Internet. SMTP не зависит от транспортной среды и может использоваться для доставки почты в сетях с протоколами, отличными от TCP/IP и X.25. Последнее обновление в RFC 5321 (2008) включает масштабируемое расширение — ESMTP (англ. Extended SMTP). В настоящее время под «протоколом SMTP» как правило подразумевают и его расширения.

Взаимодействие в рамках SMTP строится по принципу двусторонней связи, которая устанавливается между отправителем и получателем почтового сообщения. При этом отправитель инициирует соединение и посылает запросы на обслуживание, а получатель на эти запросы отвечает.

Фактически, отправитель выступает в роли клиента, а получатель - сервера. На рис. 2.1 приведена схема взаимодействия клиента и сервера по протоколу SMTP.

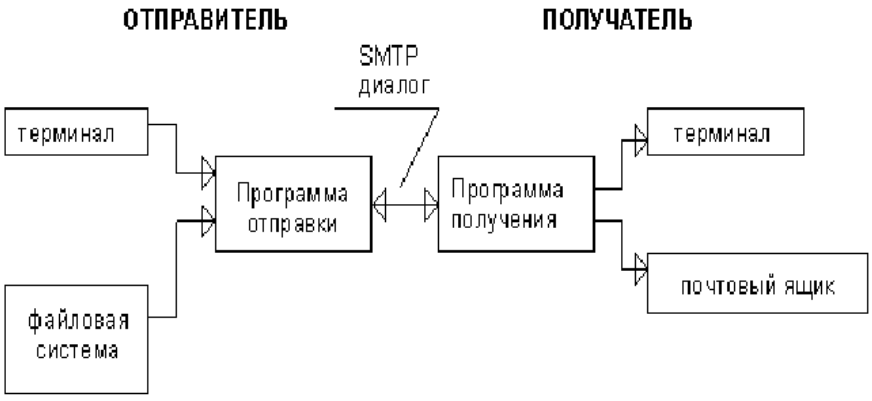

Рисунок 2.1 - Схема взаимодействия по протоколу SMTP

Канал связи устанавливается непосредственно между отправителем и получателем сообщения. При таком взаимодействии почта достигает абонента в течение нескольких секунд после отправки.

Обмен сообщениями и инструкциями в SMTP ведется в ASCII-кодах.

После установления соединения, как правило, используя 25 порт, клиент должен обязательно отправить на сервер команду

HELO <HOST>.

Эта команда используется для идентификации машины отправителя (HOST) на SMTP сервере.

Следующее командой должна идти команда MAIL, идентифицирующая отправителя:

MAIL <SP> FROM: <reverse-path> <CRLF>

Пример:

MAIL FROM: cat@australia.mail.au

Эта команда указывает SMTP-серверу начать новую транзакцию по приёму почты. В качестве аргумента, она передаёт на сервер почтовый адрес отправителя письма.

Если адрес отправителя правильный и не содержит ошибок, то сервер вернёт ответ «250 OK».

Следующей командой идёт команда

RCPT:

RCPT <SP> TO: <forward-path> <CRLF>

Пример:

RCPT TO: dog@switzerland.mail.sz

Эта команда передаёт на сервер почтовый адрес получателя письма.

Если адрес получателя не содержит ошибок, то тогда SMTP сервер вернёт ответ «250 OK». Если в адресе получателя есть ошибка, то сервер вернёт сообщение с кодом 550. Данная команда может повторяться сколь угодно долго по числу получателей, однако современные почтовые сервера вводят ограничения на количество одновременных получателей.

Следующей командой идёт команда DATA <CRLF>

Если она принимается сервером, то он возвращает сообщение с кодом 354, приглашающее продолжить отправку сообщения. После этого, на сервер можно передавать текст почтового сообщения. Признаком окончания передачи почтового сообщения является символ точки «.» в начале новой строки. Если сообщение принято к доставке, то сервер вернёт уведомление с кодом 250, а иначе – сообщение об ошибке.

После принятия сервером сообщения к отправке, клиент должен отправить команду QUIT, которая сигнализирует серверу, что больше отправки писем не будет. После принятия от сервера подтверждения этой команды, следует закрыть соединение с сервером.

Пример SMTP диалога, между отправителем (SENDER) и сервером (RECEIVER):

SENDER: MAIL FROM: <Smith@Alpha>

RECEIVER: 250 OK

SENDER: RCPT TO: <Jones@Beta>

RECEIVER: 250 OK

SENDER: RCPT TO: <Green@Beta>

RECEIVER: 550 No such user here

SENDER: RCPT TO: <Brown@Beta>

RECEIVER: 250 OK

SENDER: DATA

RECEIVER: 354 Start mail input; end with <CRLF>.<CRLF>

SENDER: Blah blah blah...

SENDER: ...etc. etc. etc.

SENDER: <CRLF>.<CRLF>

RECEIVER:

250 OK

Отличие расширенного протокола ESMTP включает набор различных способов аутентификации, AUTH-LOGIN, CRAM-MD5, DIGEST-MD5.

Наиболее распространенный 'AUTH LOGIN' механизм выглядит так

S: 220 esmtp.example.com ESMTP

C: ehlo client.example.com

S: 250-esmtp.example.com

S: 250-PIPELINING

S: 250-8BITMIME

S: 250-SIZE 255555555

S: 250 AUTH LOGIN PLAIN CRAM-MD5

C: auth login

S: 334 VXNlcm5hbWU6

C: avlsdkfj

S: 334 UGFzc3dvcmQ6

C: lkajsdfvlj

S: 535 authentication failed

Как видно из примера, сначала посылается команда auth login, в ответ от сервера в кодировке base64 приходит сообщение Username:, после чего посылается логин в base64, затем от сервера приходит сообщение Password:, клиент отправляет свой пароль в base64.

Следует учесть, что большинство серверов в настоящее время поддерживают только защищенное SSL, TLS соединение. То есть, передаваемые команды должны шифроваться и передаваться по TCP протоколу. Кодировка base64 использует соответственно 64 символа, символы латинского алфавита (заглавные, прописные) и еще некоторые. Соответственно 3 символа байтового алфавита заменяются на 4 символа base64, для выравнивания по границе используется символ =.

2.1 Формат электронного письма

Электронное письмо состоит из следующих частей:

Заголовков SMTP-протокола, полученных сервером. Эти заголовки могут включаться, а могут и не включаться в тело письма в дальнейшем, так что возможна ситуация, когда сервер обладает большей информацией о письме, чем содержится в самом письме. Так, например, поле RCPT TO указывает получателя письма, при этом в самом письме получатель может быть не указан. Эта информация передаётся за пределы сервера только в рамках протокола SMTP, и смена протокола при доставке почты (например, на узле-получателе в ходе внутренней маршрутизации) может приводить к потере этой информации. В большинстве случаев эта информация не доступна конечному получателю, который использует не SMTP протоколы (POP3, IMAP) для доступа к почтовому ящику. Для возможности контролировать работоспособность системы эта информация обычно сохраняется в журналах почтовых серверов некоторое время.

Самого письма (в терминологии протокола SMTP — 'DATA'), которое, в свою очередь, состоит из следующих частей, разделённых пустой строкой:

Заголовков письма, иногда называемых по аналогии с бумажной почтой конвертом. В заголовке указывается служебная информация и пометки почтовых серверов, через которые прошло письмо, пометки о приоритете, указание на адрес и имя отправителя и получателя письма, тема письма и другая информация. С термином «конверт» есть некоторая путаница, потому что в зависимости от ситуации «конвертом» называют либо

заголовок письма, либо информацию, которой располагает SMTP-сервер после получения письма.

Тело письма. В теле письма находится, собственно, текст письма. Согласно стандарту, в теле письма могут находиться только символы ASCII. Поэтому при использовании национальных кодировок или различных форм представления информации (HTML, RTF, бинарные файлы) текст письма должен кодироваться по стандарту MIME и не может быть прочитан человеком без использования декодера или почтового клиента с таким декодером.

Заголовок SMTP

Заголовок SMTP содержит в себе следующую информацию:

имя отправляющего узла (не имя отправителя, а имя сервера или компьютера пользователя, который обратился к серверу) — параметр сообщения HELO/EHLO, обычно дополняющийся «объективной» информацией самим сервером (HELO может содержать произвольное имя, а IP отправителя подделать существенно сложнее), по IP-адресу осуществляется поиск PTR-записи в DNS, всё это вместе позволяет идентифицировать отправителя на сетевом уровне.

Поле MAIL FROM:, содержащее адрес отправителя. Адрес может быть произвольным (в том числе с несуществующих доменов, однако этот адрес может также проверяться при первичной проверке на спам).

Поле RCPT TO: — наиболее важное поле для доставки почты, содержит электронный адрес получателя. Большинство почтовых систем в случае возможности проверяет, существует ли пользователь и может отказаться принимать почту, если пользователь, указанный в RCPT TO не существует.

Заголовок письма

Заголовок отделяется от тела письма пустой строкой. Заголовок используется для журналирования прохождения письма и служебных пометок. В Microsoft Outlook этот заголовок называется «Заголовки Интернет». В заголовке обычно указываются: почтовые серверы, через которые прошло письмо (каждый почтовый сервер добавляет информацию о том, от кого он получил это письмо), информацию о том, похоже ли это письмо на спам, информацию о проверке антивирусами, уровень срочности письма (может меняться почтовыми серверами). Также в заголовке обычно пишется программа, с помощью

которой было создано письмо. Чаще всего почтовые клиенты скрывают заголовки от пользователя при обычном использовании почтовой системой, но предоставляют возможность увидеть заголовки, если возникает потребность в более детальном анализе письма. В случае, если письмо из SMTP формата конвертируется в другой формат (например, в Microsoft Exchange 2007 письма конвертируются из SMTP-формата в MAPI), то заголовки сохраняются отдельно, для возможности диагностики.

Заголовки обычно добавляются снизу вверх (то есть каждый раз, когда к сообщению нужно добавить заголовок, он дописывается первой строкой, перед всеми предыдущими).

Помимо служебной информации, заголовки письма также хранят и показываемую пользователю информацию, это обычно отправитель письма, получатель, тема и дата отправки.

Часто используемые поля

Return-Path — обратный адрес. Может отличаться от MAIL FROM (то есть обратный адрес может быть указан отличным от адреса отправителя).

Received — строчка журналирования прохождения письма. Каждый почтовый сервер (MTA) помечает процесс обработки этим сообщением. Если сообщение проходит через несколько почтовых серверов (обычная ситуация), то новые сообщения дописываются над предыдущими (и журнал перемещения читается в обратном порядке, от ближайшего узла к самому дальнему).

MIME-Version — версия MIME, с которым это сообщение создано. Поскольку сообщение создаётся раньше всех остальных событий с письмом, то этот заголовок обычно самый первый (то есть последний в списке).

From: — Имя и адрес отправителя (именно в этом заголовке появляется текстовое поле с именем отправителя). Может не совпадать с return-path и даже не совпадать с заголовком SMTP MAIL FROM:.

Sender: — Отправитель письма. Добавлено для возможности указать, что письмо от чьего-то имени (from) отправлено другой персоной (например, секретаршей от имени начальника). Некоторые почтовые клиенты показывают сообщение при наличии sender и

from как «сообщение от 'sender' от имени 'from'». Sender является информационным заголовком (и также может отличаться от заголовка SMTP MAIL FROM).

To: — Имя и адрес получателя. Может содержаться несколько раз (если письмо адресовано нескольким получателям). Может не совпадать с полем SMTP RCPT TO.

cc: — (от англ. carbon copy). Содержит имена и адреса вторичных получателей письма, к которым направляется копия.

bcc: — (отангл. blind carbon copy).Содержит имена и адреса получателей письма, чьи адреса не следует показывать другим получателям. Это поле обычно обрабатывается почтовым сервером (и приводит к появлению нескольких разных сообщений, у которых bcc содержит только того получателя, кому фактически адресовано письмо). Каждый из получателей не будет видеть в этом поле других получателей из поля bcc.

Reply-To: — имя и адрес, куда следует адресовать ответы на это письмо. Если, например, письмо рассылается ботом, то в качестве Reply-To будет указан адрес персоны, готовой принять ответ на письмо.

Message-ID: — уникальный идентификатор сообщения. Состоит из адреса узла-отправителя и номера (уникального в пределах узла). Алгоритм генерации уникального номера зависит от сервера/клиента. Выглядит примерно так: AAB77AA2175ADD4BACECE2A49988705C0C93BB7B4A@example.com. Вместе с другими идентификаторами используется для поиска прохождения конкретного сообщения по журналам почтовой системы (почтовые системы фиксируют прохождение письма по его Message-ID) и для указания на письмо из других писем (используется для группировки и построения цепочек писем). Обычно создаётся первым почтовым сервером (MTA) в момент принятия почты от пользователя.

In-Reply-To: — указывает на Message-ID, для которого это письмо является ответом (с помощью этого почтовые клиенты могут легко выстраивать цепочку переписки — каждый новый ответ содержит Message-ID для предыдущего сообщения).

Subject: — тема письма.

Date: — дата написания письма.

Content-Type: — тип содержимого письма. С помощью этого поля указывается тип (HTML, RTF, Plain text) содержимого письма и кодировка, в которой создано письмо (см ниже про кодировки).

Помимо стандартных, почтовые клиенты, серверы и роботы обработки почты могут добавлять свои собственные заголовки, начинающиеся с «X-» (например, X-Mailer, X-MyServer-Note-OK или X-Spamassasin-Level).

Multipurpose Internet Mail Extensions (MIME) — стандарт, описывающий передачу различных типов данных по электронной почте, а также, шире, спецификация для кодирования информации и форматирования сообщений таким образом, чтобы их можно было пересылать по Интернету.

MIME определяет механизмы для передачи разного рода информации внутри текстовых данных (в частности, с помощью электронной почты), а именно: текст на языках, для которых используются кодировки, отличные от ASCII, и нетекстовый контент, такой как картинки, музыка, фильмы и программы. MIME является также фундаментальным компонентом коммуникационных протоколов, таких как HTTP, которым нужно, чтобы данные передавались в контексте сообщений подобных e-mail, даже если данные реально не являются e-mail.

Основной формат электронных сообщений определен в RFC 5322, который является обновленной версией RFC 2822 (который, в свою очередь, является обновленной версией RFC 822). Эти стандарты определяют похожие форматы для текстовых e-mail-заголовков и содержимого и правил, относящихся к общеиспользуемым полям, таким как «To:», «Subject:», «From:» и «Date:». MIME определяет набор e-mail-заголовков для определения дополнительных атрибутов сообщения, включая тип контента, и определяет множество кодировок, которые могут быть использованы для представления 8-битных бинарных данных, используя символы из 7-битного ASCII множества. MIME также определяет правила для кодирования не-ASCII символов в заголовках e-mail-сообщения, таких как «Subject:».

MIME расширяем для новых типов — его определение включает метод для

регистрации новых типов контента и других атрибутов.

Тело письма

Тело письма отделяется от заголовка пустой строкой, а заканчивается (согласно стандартам SMTP) строчкой, состоящей из единственной точки (и символа перевода строки). Часть почтовых клиентов (например, Thunderbird) показывают эту точку, часть нет. В не-smtp стандартах формат письма зависит от стандарта системы (например, MAPI), но перед «выходом» письма за пределы MAPI-совместимой системы (например, перед пересылкой через Интернет) обычно приводится к SMTP-совместимому виду (иначе маршрутизация письма была бы невозможной, так как стандартом передачи почты в Интернете является SMTP).

Одним из существенных ограничений стандартов на почтовую пересылку является применение 7-битной кодировки (ASCII). Для английского текста это не представляет особой проблемы, однако, большинство неанглоязычных языков используют 8 (и более) битные кодировки, передача которых без искажений не гарантируется. Для целей совместимости, все не 7-битные кодировки приводятся в 7-битный вид (используя различные методы кодирования текста).

Пример тела письма после команды DATA:

Received: from alpha.bieberdorf.edu (alpha.bieberdorf.edu [124.211.3.11]) by mail.bieberdorf.edu (8.8.5) id 004A21; Tue, Mar 18 1997 14:36:17 -0800 (PST)

From: rth@bieberdorf.edu (R.T. Hood)

To: tmh@immense-isp.com

Date: Tue, Mar 18 1997 14:36:14 PST

Message-Id: <rth031897143614-00000298@mail.bieberdorf.edu>

X-Mailer: Loris v2.32

Subject: Lunch today?

Do you have time to meet for lunch?

--rth

2.2 Протокол электронной почты POP3 POP3 (**Post** Office Protocol v.3)

Это простейший протокол для работы пользователя с содержимым своего почтового ящика. Он позволяет только забрать почту из почтового ящика сервера на рабочую станцию клиента и удалить ее из почтового ящика на сервере. Всю дальнейшую обработку почтовое сообщение проходит на компьютере клиента.

Многие концепции, принципы и понятия протокола POP выглядят и функционируют подобно SMTP. Команды POP практически идентичны командам SMTP. На рис. 4.2 изображена модель взаимодействия клиента и сервера по протоколу POP. Сервер POP находится между агентом пользователя и почтовыми ящиками.

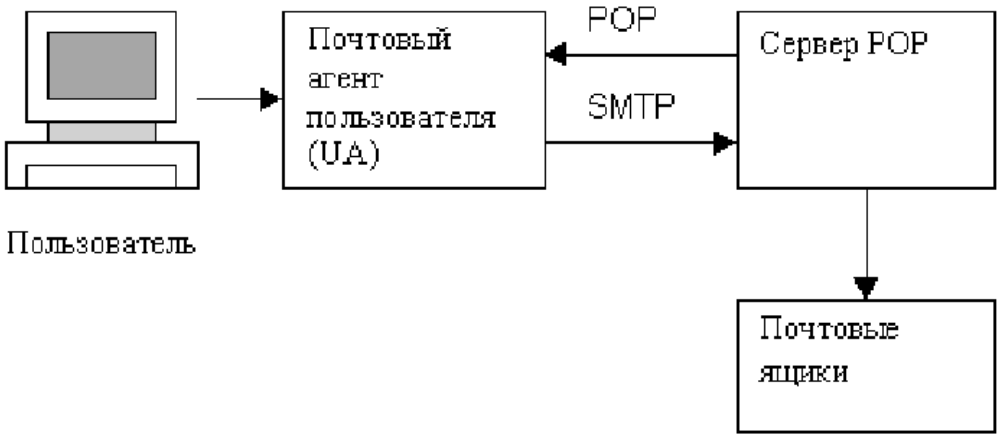

Рисунок 2.2 – Конфигурация модели клиент-сервер по протоколу POP3

В протоколе POP3 оговорены три стадии процесса получения почты:

авторизация, транзакция и обновление. После того как сервер и клиент POP3 установили соединение, начинается стадия авторизации. На стадии авторизации клиент идентифицирует себя для сервера. Если авторизация прошла успешно, сервер открывает почтовый ящик клиента и начинается стадия транзакции. В ней клиент либо запрашивает у сервера информацию (например, список почтовых сообщений), либо просит его совершить определенное действие (например, выдать почтовое сообщение). Наконец, на стадии обновления сеанс связи заканчивается.

Авторизация пользователя. После того как программа установила TCP-соединение с портом протокола POP3 (официальный номер 110), необходимо послать команду USER с именем пользователя в качестве параметра. Если ответ сервера будет +OK, нужно послать команду PASS с паролем этого пользователя:

Пример:

CLIENT: USER ivan

SERVER: +OK

CLIENT: PASS secret

SERVER: +OK ivan's maildrop has 2 messages (320 octets)

Последняя строчка ответа означает, что в почтовом ящике ivan есть 2 сообщения (320 байтов).

Транзакции POP3. После того как стадия авторизации окончена, обмен переходит на стадию транзакции. В следующих примерах демонстрируется возможный обмен сообщениями на этой стадии.

Команда STAT возвращает количество сообщений и количество байтов в сообщениях:

CLIENT: STAT

SERVER: +OK 2 320

Команда LIST (без параметра) возвращает список сообщений в почтовом ящике и их размеры:

CLIENT: LIST

SERVER: +OK

SERVER: 2 messages (320 octets)

SERVER: 1 120

SERVER: 2 200

...

Команда LIST с параметром возвращает информацию о заданном сообщении:

CLIENT: LIST 2

SERVER: +OK 2 200

...

CLIENT: LIST 3

SERVER: -ERR no such message, only 2 messages in maildrop

Команда TOP возвращает заголовок, пустую строку и первые десять строк тела сообщения:

CLIENT: TOP 10

SERVER: +OK

SERVER: <the POP3 server sends the headers of the message, a blank line, and the first 10 lines of the message body> (сервер POP высылает заголовки сообщений, пустую строку и первые десять строк тела сообщения)

SERVER:

....

CLIENT: TOP 100

SERVER: -ERR no such message

Команда NOOP не возвращает никакой полезной информации, за исключением позитивного ответа сервера. Однако позитивный ответ означает, что сервер находится в соединении с клиентом и ждет запросов:

CLIENT: NOOP

SERVER: +OK

Следующие примеры показывают, как сервер POP3 выполняет действия.

Например, команда RETR извлекает сообщение с указанным номером и помещает его в буфер местного UA (почтового агента):

CLIENT: RETR 1

SERVER: +OK 120 octets

SERVER: <the POPS server sends the entire message here> (POP3-сервер высылает сообщение целиком)

SERVER:

. . . . . .

Команда DELE отмечает сообщение, которое нужно удалить:

CLIENT: DELE 1

SERVER: +OK message 1 deleted ...

(сообщение 1 удалено) CLIENT:

DELE 2

SERVER: -ERR message 2 already deleted (сообщение 2 уже удалено)

Команда RSET снимает метки удаления со всех отмеченных ранее сообщений:

CLIENT: RSET

SERVER: +OK maildrop has 2 (в почтовом ящике 2 сообщения (320 байтов)) messages (320 octets)

Как и следовало ожидать, команда QUIT закрывает соединение с сервером:

CLIENT: QUIT

SERVER: +OK dewey POP3 server signing off

CLIENT: QUIT

SERVER: +OK dewey POP3 server signing off (maildrop empty)

CLIENT: QUIT

SERVER: +OK dewey POP3 server signing off (2 messages left)

Обратите внимание на то, что отмеченные для удаления сообщения на самом деле не удаляются до тех пор, пока не выдана команда QUIT и не началась стадия обновления. В любой момент в течение сеанса клиент имеет возможность выдать команду RSET, и все

отмеченные для удаления сообщения будут восстановлены.

2.3 Возможности реализации на различных языках программирования.

Если вы выбрали в качестве языка программирования C#, то можно воспользоваться компонентом TcpClient. Который соответственно использует поток чтения и записи.

```
TcpClient tcpClient = new TcpClient();
//подключения по IP  и по указанному номеру порта.
tcpClient.Connect("x.x.x.x", 9999);
//получение потока
networkStream = tcpClient.GetStream();
//поток для чтения
clientStreamReader = new StreamReader(networkStream);
//поток для записи
clientStreamWriter = new  StreamWriter(networkStream);
while(true)
{
  clientStreamReader.Read()
}
```

Очевидно, что необходимо проверять, когда поток окажется пустым или в соответствии с протоколом закончатся данные, что является следствием организации протокола TCP.

Для защищенного протокола SSL, типичный простой код без каких-либо проверок:

```
string server = "smtp.server.com";
TcpClient client = new TcpClient(server,25)
var stream = client.GetStream()
var sslStream = new SslStream(stream)
sslStream.AuthenticateAsClient(server);
var writer = new StreamWriter(sslStream)
var reader = new StreamReader(sslStream)
writer.WriteLine("EHLO " + server);
writer.Flush();
Console.WriteLine(reader.ReadLine());
```

Ниже приведен пример реализации соединения и передаче данных по протоколу ESMTP на python.

```
import socket
import ssl
from socket import *
mailserver = 'smtp.mail.ru'
cSock = socket(AF_INET, SOCK_STREAM)
cSock.connect((mailserver, 465))
cSockSSL = ssl.wrap_socket(cSock)
recv = cSockSSL.recv(1024)
print(recv)
cSockSSL.send("EHLO host\r\n")
recv = cSockSSL.recv(1024)
print(recv)
cSockSSL.close()
cSock.close()
```

Для тех, кто будет использовать STARTTLS, можно воспользоваться помощью следующего кода.

```
import ssl
from socket import *
mailserver = 'smtp.mail.ru'
cSock = socket(AF_INET, SOCK_STREAM)
# лучше использовать 587, так как 25 используется во взаимодействии сервер-сервер. Хотя похоже работает и 25 (но это не точно).
# 587 используется для взаимодействия клиента с мейл трансфер агентом MTA.
cSock.connect((mailserver, 25))

cSock.send(b"EHLO host\r\n")
recv = cSock.recv(1024)
print(recv)
input()
```

```
cSock.send(b"STARTTLS\r\n")
recv = cSock.recv(1024)
print(recv)
input()
recv = cSock.recv(1024)
print(recv)
input()
# отключаем хендшейк при коннекте и используем tls, хотя можно и не отключать.
cSockSSL = ssl.wrap_socket(cSock,ssl_version=ssl.PROTOCOL_TLS,do_handshake_on_connect=False)
cSockSSL.do_handshake()
cSockSSL.send(b"EHLO host\r\n")
recv = cSockSSL.recv(1024)
print(recv)
cSockSSL.send(b"AUTH LOGIN\r\n")
recv = cSockSSL.recv(1024)
print(recv)
cSockSSL.close()
cSock.close()
```

2.4 Задание на лабораторную работу

Заведите почтовый ящик на каком-либо почтовом сервере, например, mail.ru. Осуществите подключение по протоколу TCP. Например, настройки почтового сервера smtp для mail.ru можно найти на https://help.mail.ru/mail-help/mailer/popsmtp. Соответственно, сервер smtp.mail.ru, порт 465, шифрование SSL/TLS. Описание формата mime https://www.opennet.ru/docs/RUS/mime_rfc/.

Ознакомившись с протоколами SMTP, описанным в RFC 788 и POP3, описанным в RFC 1939 выполнить нижеприведенные задания по вариантам, необходимо создать как минимум два отдельных приложения.

Использовать сокеты, никаких готовых компонент.

1) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту. Реализовать так же консольное приложение для получения письма. Вытащить заголовки письма, если они есть.

2) Создать приложение, в котором будут вводиться текст письма, текст письма

по протоколу SMTP должен быть отправлен на почту. Добавить тему и дату письма. Реализовать так же консольное приложение для получения письма.

3) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту с mime формате в кодировке UTF-8. Добавить тему и дату письма. Реализовать так же консольное приложение для получения письма.

4) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту в виде html (использовать mime). Добавить тему и дату письма. Реализовать так же консольное приложение для получения письма.

5) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту в виде текста utf-8 (использовать mime), а также короткий звуковой файл. Реализовать так же консольное приложение для получения письма.

6) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту в виде текста utf-8 (использовать mime), а также короткий текстовый файл. Реализовать так же консольное приложение для получения письма.

7) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту в виде текста utf-8 (использовать mime), а также файл jpeg. Реализовать так же консольное приложение для получения письма.

8) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту. Получить текстовое сообщение и найти в нем boundary. Просмотреть список писем.

9) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту. Получить текстовое сообщение. Просмотреть список писем. Удалить письмо, просмотреть состояние почтового ящика.

10) Создать приложение, в котором будут вводиться текст письма, текст письма по протоколу SMTP должен быть отправлен на почту в виде текста utf-8 (использовать mime), а также файл png. Реализовать так же консольное приложение для получения письма.

Указанные приложения можно реализовать как в консольном варианте, так и с применением GUI. Для реализации можно использовать Java-Swing, AWT, C# - .Net , C++ - QT, Python.

3 Лабораторная работа №3 Работа с HTTP и FTP

Лабораторная работа №3 выполняется после изучения материала, посвященного описанию принципов использования C#, Java, Python, C++ для управления соединениями с сервером по протоколам HTTP и FTP.

Цель работы:

Написать приложения для ОС Windows или ОС Linux, представляющие собой простые HTTP и FTP – клиенты. Для отправки команд нужно использовать сокетные соединения, для парсинга можно использовать готовые компоненты. Лабораторная считается сделанной, если создано подключение и возможно скачать файл FTP или сделать просмотр файлов и каталогов, для HTTP или HTTPS можно запросить html страницу, в ней найти ссылку и скачать какой-либо объект по этой ссылке.

Приложение FTP (FTPS, rfc4217, https://tools.ietf.org/html/rfc4217) клиента использует указанные протоколы. Не путайте FTPS (SSL FTP) с FTP через SSH (туннелирование обычного FTP через соединение SSH, защита реализуется только для пассивного клиентского соединения из-за особенности работы FTP в активном режиме требующего подключения сервера к клиенту). Так же есть SFTP (SHH FTP, защищенное через SHH FTP соединение, SSH2, отдельный от FTP протокол, продолжение SSH), а также с SFTP (Simple FTP, урезанная версия FTP).

Желающие могут сделать GUI десктоп приложение. Например, используя C# и Visual Studio это сделать достаточно просто, путем размещения соответствующих компонент ввода текстовых полей, кнопок на форме и задав им соответствующие события, и используя свойства данных компонент.

Использование QT требует бОльших затрат на изучение данной технологии, но может дать неоценимый опыт при изучении концепции использования сигналов и слотов, при реализации возможностей обработки событий. Для создания приложений на QT рекомендуется использовать QTCreator и QT Designer. GUI приложения на Java могут использовать swing, апплеты awt, библиотека javafx.

Очень простой пример на Python с использованием wxPython. Так же может понадобиться установка python-wxgtk-webview3.0 под Linux. Здесь реализована одна простая функция перехода по ссылке, при этом нет обработки события открытия нового окна, назад, вперед и многих других функций.

```
import wx
import wx.html2

class TestFrame(wx.Frame):
        go = []
        address = []
        browser = []
#обработчик события нажатия кнопки
        def onGoButton(self,event):
                print("ButtonClick")
                print(self.address.GetValue())
#загрузка страницы, адрес берется из строки ввода
                self.browser.LoadURL(self.address.GetValue())
                return

        def __init__(self, parent, title):
                wx.Frame.__init__(self, parent, id=-1, title=title)
                text = wx.StaticText(self, label=title)
# компонент бокссайзер позволяет управлять размещением объектов на форме
#(в данном случае вертикально)
                sizer = wx.BoxSizer(wx.VERTICAL)
#объект для создания минибраузера
                self.browser = wx.html2.WebView.New(self)
#объект для текстовой строки ввода
                self.address = wx.TextCtrl(self, value="http://")
#объект кнопка
                self.go = wx.Button(self, label="Go", id=wx.ID_OK)
#добавляем объекты для пропорционального размещения в соотношении 90, 5, 5
                sizer.Add(self.browser, proportion = 90, flag = wx.EXPAND, border = 10)
                sizer.Add(self.address, proportion = 5, flag = wx.EXPAND, border = 10)
                sizer.Add(self.go, proportion = 5, flag = wx.EXPAND, border = 10)
                self.SetSizer(sizer)
                self.SetSize((1000, 800))
#связываем обработчик события onGoButton и кнопку go
                self.Bind(wx.EVT_BUTTON,self.onGoButton,self.go)
```

```
app = wx.App()
frame = TestFrame(None, "Simple browser")
frame.Show()
app.MainLoop()
```

Пример ftp клиента на python, показывающего некоторые компоненты, которые можно использовать при выполнении задания, нужно проверить на наличие блокировок, добавить по заданию нужные действия, можно делать на другом языке программирования. Пример работает не на всех серверах, Вы должны это исправить, в том числе для вашего варианта, чтобы на всех серверах работало, так, например, не учтен первый прием данных с кодом xyz-, не учтен пробел в конце кода xyz, не учтено что может быть получен пустой буфер, не учтено что код xyz может находиться на последней строке полученного буфера.

Сервера которые вы можете использовать и на которых должно работать ваше приложение:

1) test.rebex.net (логин demo, пароль password)
2) ftp.gnu.org (логин anonymous, пароль anonymous)
3) ftp.pureftpd.org
4) ftp.vim.org
5) ftp.slackware.com
6) in1.hostedftp.com

```
import socket
import re
import time
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
   res = s.connect(("url to server", 21))
   print("Connection OK")
except:
   print("Error connection")

buf = s.recv(1024)
print(buf.decode('utf-8'))
```

```
CMD_USER = b"USER anonymous\r\n"
CMD_PASS = b"PASS anonymous@anon.org\r\n"
CMD_PASV = b"PASV\r\n"
CMD_LIST = b"LIST\r\n"
CMD_QUIT = b"QUIT\r\n"

def send_cmd(cmd: bytes):
    time.sleep(0.1)
    s.send(cmd)
    print(cmd[:-2].decode('utf-8'))
    buf = s.recv(1024)
    print(buf.decode('utf-8'))
    # Если многострочный прием, по rfc начинается с xyz- заканчивается xyz<SP>
    if buf[3] == ord("-"):
        res = buf[0:3].decode('utf-8')
        while(True):
            buf = s.recv(1024)
            print(buf.decode('utf-8'))
            if(buf[0:3].decode()==res):
                break
    # Если требуется продолжение действий (читать rfc по кодам ответов ftp)
    if(buf[0]== ord("1")):
        buf = s.recv(1024)
        print(buf.decode('utf-8'))
    return buf[:-2].decode('utf-8')



send_cmd(CMD_USER)
send_cmd(CMD_PASS)
```

```
ret = send_cmd(CMD_PASV)
# ищем подстроку соответствующую регулярному выражению
match = re.search(r"(\d+,\d+,\d+,\d+,\d+,\d+)",ret)
# разбиваем через запятую получая список чисел соответствующих ip port
match = re.split(r",",match[0])
ip = ".".join(match[0:4])
port = int(match[4])*256+int(match[5])
print(f"ip {ip} port {port}")


s1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
    res = s1.connect((ip, port))
    print("Connection OK")
except:
    print("Error connection")

send_cmd(CMD_LIST)
buf = s1.recv(1024)
print(buf)
send_cmd(CMD_QUIT)
s.close()
```

Пример HTTP. Данный пример показывает подключение к веб серверу и получение html страницы. Затем парсит ее. Пример приведен с целью показать основные компоненты, которые можно использовать. Если необходим https, можно воспользоваться примером из второй лабораторной с использованием ssl.

```
from socket import *
http_server = 'example.org'
cSock = socket(AF_INET, SOCK_STREAM)
cSock.connect((http_server, 80))
```

```
cSock.send(b"GET /index.html HTTP/1.0\r\n")
cSock.send(b"HOST: example.org\r\n")
cSock.send(b"\r\n")

from http_parser.http import HttpStream
from http_parser.reader import SocketReader

r = SocketReader(cSock)
p = HttpStream(r)
print(p.headers())
body = p.body_file().read()

from html.parser import HTMLParser

class tHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print("Start tag:", tag)
        for attr in attrs:
            print("     attr:", attr)
parser = tHTMLParser()
parser.feed(body.decode('utf8'))
```

Общее задание. Создать простейший ftp и http клиент. Обеспечить подключение к серверу и выполнение команд по заданию.

3.1 Варианты заданий.

1) FTP клиент должен просматривать файлы, переходить в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти по ссылке и скачать документ находящийся по этой ссылке.

2) FTP клиент должен просматривать файлы, выдавать размера файла, переходить в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти к изображению и скачать файл находящийся по ссылке.

3) FTP клиент должен просматривать файлы, переименовать файл, переходить

в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать титульные заголовки html.

4) FTP клиент должен просматривать файлы, закачать файл, переходить в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать титульные заголовки html.

5) FTP клиент должен просматривать файлы, закачать файл, переходить в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать титульные заголовки html.

6) FTP клиент должен просматривать файлы, просматривать время модификации, переходить в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать список заголовков ответа, и найти все img ссылки.

7) FTP клиент должен просматривать файлы в кратком формате, просматривать время модификации, переходить в другую папку, скачивать файл. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать весь список тэгов.

8) FTP клиент должен просматривать файлы в кратком формате, создать каталог, удалить каталог, переходить в другую папку. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать весь список тэгов.

9) FTP клиент должен просматривать файлы в кратком формате и подробном формате, удалить каталог, переходить в другую папку. HTTP должен выдать заголовки запроса, перейти к ссылке и скачать файл html находящийся по ссылке, выдать A теги.

10) FTP клиент должен просматривать файлы и каталоги и скачать два файла, переходить в другую папку. HTTP должен выдать заголовки запроса, перейти по ссылке и скачать файл html находящийся по ссылке, выдать его img теги.

4. Лабораторная работа 4. Перехват сетевых пакетов

Реализовать программу (сниффер) для перехвата пакетов в сети. Реализовать удобный вывод данных о перехваченном пакете, подсветку.

Любым программам для мониторинга сети, инструментам безопасности необходим перехват сетевых пакетов. На языке C# и для Фреймворка Net существует по крайней мере две библиотеки: SharPcap и WinPKFilter. Так же есть реализации на других языках.

Можно использовать RAW сокеты и самостоятельно анализировать, получаемые данные. Большинство этих библиотек требуют администраторских прав. Кроме того, представленные ниже примеры могут устареть, в связи с обновлением библиотек, поэтому желательно рассмотреть примеры, представленные в официальной документации.

Так как, большинство библиотек используют RAW сокеты, то ваше приложение для запуска потребует root прав или запуска под sudo, если вас не устраивают подобные обстоятельства, то можно установить виртуальную машину или виртуальную машину с контейнерной виртуализацией, например, Docker.

Далее приведен пример использования RAW сокетов и библиотеки scapy. Сначала рассмотрим RAW сокеты.

4.1 Пример использования библиотеки RAW сокетов.

Для доступа к низкоуровневым сокетам требуются root права, потому в виртуальному машине нужно будет запускать установку приложений pip через sudo, как и интерпретатор Python. Для упрощения работы создадим файл python с нашим приложением, поместив туда начальный код:

```
import socket
# 0x800 номер перехватываемого протокола при получении кадров Ethernet, в данном случае будут ловиться кадры содержащие пакеты IP.
rawSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))
# получаем один кадр
data, addr = rawSocket.recvfrom(2048)

print(addr)
print(data)
```

Создадим shell файл со следующими командами. Здесь ! – имя процесса запущенного в фоновом режиме через команду &. Второй процесс сейчас не обязательно уничтожать, так как он итак прекращается, но на будущее если это будет снифер, мы можем оставить эту команду.

```
sudo python3 snif_sr.py & APP_ID1=$!
ping 127.0.0.1 & APP_ID2=$!

sleep 5
kill -TERM $APP_ID2
kill -TERM $APP_ID1
```

Теперь, нужно дополнить наше простое приложение разбором полученных данных. Для этого подойдет модуль struct.

Попробуйте данный пример.

```
import socket
import struct
import binascii
rawSocket = socket.socket(socket.PF_PACKET, socket.SOCK_RAW, socket.htons(0x0800))

data, addr = rawSocket.recvfrom(2048)

# считываем заголовок Ethernet пакета

eth_hdr = struct.unpack("!6s6s2s", data[0:14]) # 6 dest MAC, 6 host MAC, 2 ethType

# преобразуем MAC в 16-иричный формат
print(eth_hdr)
print(binascii.hexlify(eth_hdr[0]))
print(binascii.hexlify(eth_hdr[1]))
print(binascii.hexlify(eth_hdr[2]))
```

```
# считываем заголовок ip пакета
ipHeader = data[14:34]

ip_hdr = struct.unpack("!12s4s4s", ipHeader)
# преобразуем адреса в сетевом порядке в строковое представленеи и выводим на экран
print(f"Source IP: {socket.inet_ntoa(ip_hdr[1])}")
print(f"Dest IP: {socket.inet_ntoa(ip_hdr[2])}")
```

Далее рассмотрены возможности модуля struct. Сам модуль содержит метод pack и unpack.

Например,

```
# Упаковать список чисел. Метод pack()
pack_obj = struct.pack('>4i', LS[0], LS[1], LS[2], LS[3])

# Вывести упакованный объект pack_obj
print('pack_obj = ', pack_obj)

# Распаковать список чисел. Метод unpack().
#  Результатом есть кортеж T2
T2 = struct.unpack('>4i', pack_obj) # T2 = (1, 3, 9, 12)
```

В языке Python способ упаковки строки определяется на основе первого символа строки формата. Этот символ определяет:

- порядок байт, который формируется с помощью символов **@**, **=**, **<**, **>**, **!**. Если не указать этого параметра, то принимается символ **@**;
- размер в байтах упакованных данных. В этом случае первыми используются цифры, которые обозначают число;
- выравнивание, устанавливаемое системой.

В соответствии с документацией Python в строке формата порядок байт, размер и выравнивание формируются согласно первому символу формата. Возможные первые символы формата отображены в следующей таблице.

Символ	Порядок байт	Размер	Выравнивание
@	Естественный (зависит от хост-системы)	Естественный	Естественное
=	Естественной	Стандартный	Отсутствует
<	little-endian	Стандартный	Отсутствует
>	big-endian	Стандартный	Отсутствует
!	сетевой (аналог big-endian)	Стандартный	Отсутствует

Значение порядка байт может быть одним из 4-х:

- естественной порядок (native). Этот порядок может быть или little-endian или big-endian. Данный порядок определяется в зависимости от хост-системы;
- порядок типа little-endian. При таком порядке первым обрабатывается младший байт, а затем уже старший байт;
- порядок типа big-endian. В этом случае первым обрабатывается старший байт, а затем уже младший байт;
- сетевой порядок (network), который по умолчанию равен порядку big-endian.

Размер упакованных данных может быть одним из двух:

- естественный – определяется с помощью инструкции sizeof компилятора языка C;
- стандартный – определяется на основе символа формата в соответствии с нижеследующей таблицей.

Таблица 1 - Определение стандартного размера упакованных данных в зависимости от символа формата

Формат	Тип в языке C	Тип в языке Python	Стандартный размер
x	pad byte	без значения	
c	char	байты длиной 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
n	ssize_t	integer	
N	size_t	integer	

e	float (экспоненциальный формат)	float	2
f	float	float	4
d	double	float	8
s	char[]	bytes	
p	char[]	bytes	
P	void*	integer	

Примеры форматированных строк для разных типов данных

ii - два числа типа int

2i - два числа типа int

10f - 10 чисел типа float

>i8s - порядок байт big-endian, число типа int, строка из 8 символов

8dif - 8 чисел типа double, 1 число типа int, 1 число типа float

=bi - естественной порядок, значение типа bool, число типа int

4.2 Пример использования Scapy

Scapy — это удобный инструмент для создания пакетов вручную. Утилита написана с использованием языка Python, автором является Philippe Biondi. Возможности утилиты довольны широкие — это и сборка пакетов с последующей отправкой их в сеть, и захват пакетов, и чтение их из сохраненного ранее дампа, и исследование сети. Всё это можно делать как в интерактивном режиме, так и создавая скрипты.

С помощью Scapy можно проводить сканирование, трассировку, исследования, атаки и обнаружение хостов в сети.

Scapy предоставляет среду или даже фреймворк, чем-то похожий на Wireshark, только без красивой графической оболочки.

Утилита разрабатывается под UNIX-подобные операционные системы, но тем не менее, ее можно запустить и в среде Windows.

Можно, например, выяснить поддерживаемые протоколы и поля протоколов.

```
import scapy.all as sc
# все поддерживаемые протоколы
print(sc.ls())
# поля протокола IP
print(sc.ls(sc.IP))
```

Сформировать пакеты, вложение указывается через /.

```
print("Формируем пакет, уровни разделены --")
packet = sc.IP(dst="127.0.0.1")/sc.TCP(dport=80)/"Hello"
```

```
# выводим информацию о пакете
print(sc.ls(packet))
# краткая информация о пакете
print(packet.summary())
# более подробная информация
print(packet.show())
# доступ к какому либо слою и полю слоя (например, сегменту TCP и полю flags)
print(f"TCP flags {packet[sc.TCP].flags}")
```

Перечислим некоторые функции библиотеки.

функция send() – отправляет пакеты, используя сетевой (L3) уровень, никак не обрабатывая ответы. Используется принцип — отправили и забыли;

функция sendp() – отправляет пакеты, используя канальный (L2) уровень, учитываются указанные параметры и заголовки Ethernet кадров. Ответы всё так же не ожидаются и не обрабатываются;

функция sr() – является аналогичной send(), исключение составляет то, что она уже ожидает ответные пакеты;

функция srp() – отправляет и принимает пакеты, уровень L2

функция sr1() – отправляет пакет третьего уровня и забирает только первый ответ, множество ответов не предусматривается;

функция srp1() – аналогично sr1(), только уже канальный уровень.

Пример отправки сформированного пакета.

```
sc.send(packet)
```

Получить все пакеты (10 первых) приходящие на интерфейсы (снифер).

```
def get_packet(pkt):
    print(pkt.summary())
    return
sc.sniff(prn = get_packet, count = 10)
```

Вывести на экран интерфейсы и получать пакеты с заданного интерфейса с указанным адресом.

```
print(sc.ifaces)
sc.sniff(prn = get_packet, filter = "ip host 127.0.0.1", count = 10, iface = "lo", timeout =10)
```

4.3 Пример работы с Docker.

Рассмотрим, как можно запустить ваше приложение в Docker, все ниже перечисленные операции приведены для Ubuntu.

Установим Docker.

Можно зайти по адресу https://docs.docker.com/desktop/install/ubuntu/ и установить путем скачивания deb файла. Либо инсталлировать используя репозиторий как указано на сайте https://docs.docker.com/engine/install/ubuntu/#set-up-the-repository.

При ручной установке сначала нужно скачать еще файлы в зависимости от версии ubuntu, версию можно посмотреть командой:

lsb_release -a

Скачиваем, в зависимости от версии linux, например, focal

https://download.docker.com/linux/ubuntu/dists/focal/pool/stable/amd64/.

containerd.io_<version>_<arch>.deb

docker-ce_<version>_<arch>.deb

docker-ce-cli_<version>_<arch>.deb

docker-compose-plugin_<version>_<arch>.deb

Устанавливаем

```
sudo dpkg -i ./containerd.io_<version>_<arch>.deb \
  ./docker-ce_<version>_<arch>.deb \
  ./docker-ce-cli_<version>_<arch>.deb \
  ./docker-compose-plugin_<version>_<arch>.deb
```

Проверяем, что установка прошла успешно

```
sudo docker run hello-world
```

В команде указать название файла который вы скачали, либо через sudo dpkg –i название.deb.

```
sudo apt-get update
sudo apt-get install ./docker-desktop-<version>-<arch>.deb
```

Просмотр информации и команд docker

```
sudo docker
```

Просмотр общесистемной информации о docker.

sudo docker info

Просмотр информации о команде docker

sudo docker команда –help

Например

sudo docker exec –help

Поиск доступных образов операционных систем, скрипт пробежится по Docker Hub (репозиторий по умолчанию) и вернет список всех образов с именами, совпадающими со строкой запроса:

sudo docker search ubuntu

В столбце OFFICIAL OK указывает на образ, созданный и поддерживаемый компанией, реализующей проект. После того как вы определили образ, который хотели бы использовать, вы можете загрузить его на свой компьютер с помощью субкоманды pull, например, с учетом установленной у вас операционной системы:

sudo docker pull ubuntu:20.04

Просмотреть имеющиеся образы

sudo docker images

Образы, которые вы используете для запуска контейнеров, можно изменить и использовать для создания новых образов, которые затем могут быть загружены (помещены) на Docker Hub или другие реестры Docker.

Контейнер hello-world служит примером контейнера, который запускается и завершает работу после отправки тестового сообщения. Контейнеры также могут быть интерактивными.

В качестве примера запустим контейнер с нашим образом Ubuntu. Сочетание переключателей -i и -t предоставляет вам доступ к интерактивной командной оболочке внутри контейнера:

sudo docker run -it ubuntu:20.04

Кстати, если запустить sudo docker run -it ubuntu, то установится последняя версия образа latest и добавиться в список образов, можно будет их просмотреть. Выход из контейнера команда exit.

Просмотр активных и неактивных контейнеров:

sudo docker ps

sudo docker ps -a

Запуск и остановку контейнера можно реализовать командами start и stop указав CONTAINER ID или NAME.

Например,

sudo docker start strange_cerf

sudo docker ps

sudo docker stop 8331dcd04f50

Удаление контейнера docker rm имя_контейнера.

Можно обновить состояние контейнера установив что-либо там, в результате появится еще один контейнер с установленным уже там дополнительным ПО.

Например в контейнере запустить:

apt update

apt install python

После выхода появится еще один контейнер.

Запустив его получим, контейнер с установленным там Пайтоном (указывайте имя своего контейнера).

sudo docker start -i eager_heyrovsky

Для того, чтобы скопировать файлы из локальной машины в докер контейнер нужно воспользоваться:

docker cp путь_на_локальной_машине имя_контейнера:путь_в_контейнере

Для обратного процесса выполняем команду:

docker cp имя_контейнера:путь_в_контейнере путь_на_локальной_машине

Например, скопируем все нужные файлы с приложением снифера в наш контейнер с python3.

sudo docker cp /home/user/prog/. eager_heyrovsky:/home/prog

Запустим контейнер и затем наше приложение.

sudo docker start -i eager_heyrovsky

Придется установить внутри контейнера

apt install libpcap-dev

apt install pip

apt install python3

pip install scapy

Если есть проблемы с кодировкой, то в запускаемом файле Python можно поставить

вначале:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

4.4 Библиотека SharpPcap

SharpPcap — библиотека для .NET, которая позволяет перехватывать пакеты. По сути, это обертка над библиотекой Pcap, которая используется во многих популярных продуктах. Например, сниффер Wireshark, IDS Snort.

С SharpPCap также поставляется замечательная библиотека для парсинга пакетов — Packet.Net.

Packet.Net поддерживает следующий протоколы:

* Ethernet
* LinuxSLL
* Ip (IPv4 and IPv6)
* Tcp
* Udp
* ARP
* ICMPv4 и ICMPv6
* IGMPv2
* PPPoE
* PTP
* Link Layer Discovery Protocol (LLDP)
* Wake-On-LAN (WOL)

Работать с библиотекой достаточно просто:

```
// метод для получения списка устройств
 CaptureDeviceList deviceList = CaptureDeviceList.Instance;
 // выбираем первое устройство в спсике (для примера)
 ICaptureDevice captureDevice = deviceList[0];
 // регистрируем событие, которое срабатывает, когда пришел новый пакет
 captureDevice.OnPacketArrival += new
PacketArrivalEventHandler(Program_OnPacketArrival);
 // открываем в режиме promiscuous, поддерживается также нормальный режим
 captureDevice.Open(DeviceMode.Promiscuous, 1000);
```

```
// начинаем захват пакетов
captureDevice.Capture();
```

Теперь в обработчике события
device_OnPacketArrival

мы можем работать с пакетом:

```
static void Program_OnPacketArrival(object sender, CaptureEventArgs e)
{
// парсинг всего пакета
Packet packet = Packet.ParsePacket(e.Packet.LinkLayerType, e.Packet.Data);
// получение только TCP пакета из всего фрейма
var tcpPacket = TcpPacket.GetEncapsulated(packet);
// получение только IP пакета из всего фрейма
var ipPacket = IpPacket.GetEncapsulated(packet);
if (tcpPacket != null && ipPacket != null)
{
DateTime time = e.Packet.Timeval.Date;
int len = e.Packet.Data.Length;
// IP адрес отправителя
var srcIp = ipPacket.SourceAddress.ToString();
// IP адрес получателя
var dstIp = ipPacket.DestinationAddress.ToString();
// порт отправителя
var srcPort = tcpPacket.SourcePort.ToString();
// порт получателя
var dstPort = tcpPacket.DestinationPort.ToString();
// данные пакета
var data = tcpPacket.PayloadPacket;
}
}
```

Так же библиотека позволяет создавать пакеты и отправлять, работать с дампами и многое другое. Без проблем работает на mono. У SharpPcap есть конкурент Pcap.net. По описанию возможности совпадают.

4.5 Драйвер WinPKFilter

WinPKFilter — NDIS драйвер для перехвата пакетов. Поддерживаются различные операционные системы:x/ME/NT/2000/XP/2003/Vista/2008/Windows 7/2008R2. Плюсом драйвера является то, что он позволяет модифицировать и блокировать пакеты. Для некоммерческих проектов библиотека бесплатна.

Для удобной работы с драйвером предоставляется библиотека. На сайте можно скачать обертки для этой библиотеки для следующих языков — C#, Delphi, VB, MS VC++, C++ Builder.

Работать с WinPkFilter сложнее, чем SharpPcap, нужны хорошие знания в работе с неуправляемым кодов и в маршалинге. Да и размер кода получается намного больше. На официальном сайте можно задать вопрос, на который Вы без проблем получите от автора (кстати русский).

4.6 Еще один перехватчик с использованием библиотеки scapy на python.

Далее описан сканер, который будет осуществлять пассивное сканирование сети методом перехвата SSID-идентификаторов.

Весь проект занимает несколько строчек кода на Python, в основу положена библиотека Scapy (www.secdev.org/projects/scapy), предназначенная для манипуляции сетевыми пакетами. К сожалению, под Windows это реализуется на несколько порядков сложнее, поэтому в качестве платформы мы выберем Linux.

Установка scapy

cd /scapy/ & python setup.py install.

Далее можно приступить к написанию сканера. Сканер будет просматривать специальные фреймы, которые содержат уникальный идентификатор сети SSID (Service Set Identifier) и рассылаются точкой доступа. Их также называют Beacon-фреймами. По ним мы и будут определяться найденные сети.

```
import sys

from scapy import *

print "Wi-Fi SSID passive sniffer"

interface = sys.argv[1] #задаем название интерфейса в качестве дополнительного
```

консольного аргумента

```
def sniffBeacon(p):
    if p.haslayer(Dot11Beacon):
    print p.sprintf("%Dot11.addr2%[%Dot11Elt.info%|%Dot11Beacon.cap%]")
sniff(iface=interface,prn=sniffBeacon)
```

Вывод программы содержит информацию о перехваченных Beacon-фреймах:

```
skvoz@box: sniffssid.py eth1
00:12:17:3c:b6:ed['netsquare4'|short-slot+ESS]
00:30:bd:ca:1e:1e['netsquare7'|ESS+privacy]
```

Также можно поэкспериментировать с выбором haslayer (параметра мониторинга), поменяв его значение Dot11Beacon на: Dot11AssoResp, Dot11ProbeReq, Dot11ATIM, Dot11Auth, Dot11ProbeResp, Dot11Addr2MACField, Dot11Beacon, Dot11ReassoReq, Dot11Addr3MACField, Dot11Deauth, Dot11ReassoResp, Dot11Addr4MACField, Dot11Disas, Dot11WEP, Dot11AddrMACField, Dot11Elt, Dot11AssoReq, Dot11PacketList. В этом случае можно перехватить не только SSID точки доступа, но и всю остальную информацию о сети. К примеру, узнать информацию о физических идентификаторах пользователей и сетевых обращениях. Для этого мы задействуем протокол ARP:

```
import sys, os
from scapy import *
interface = raw_input("enter interface") #пользователь задает интерфейс сети
os.popen("iwconfig interface mode monitor") #перевод карты в режим монитора на заданном интерфейсе
#функция перехвата MAC
def sniffMAC(p):
    if p.haslayer(Dot11):
```

```
        mac = p.sprintf("[%Dot11.addr1%)|(%Dot11.addr2%)|(%Dot11.addr3%)]")
        print mac
#функция перехвата IP-адресов и показа ARP сообщений
def sniffarpip(p):
        if p.haslayer(IP):
        ip = p.sprintf("IP - [%IP.src%)|(%IP.dst%)]")
        print ip
        elif p.haslayer(ARP):
        arp = p.sprintf("ARP - [%ARP.hwsrc%)|(%ARP.psrc%)]-
[%ARP.hwdst%)|(%ARP.pdst%)]")
        print arp
#уровни, которые мы мониторим
sniff(iface=interface,prn=sniffMAC, prn=sniffarpip)
```

Вывод

```
skvoz@puffy: python sniff.py eth1
[ff:ff:ff:ff:ff:ff)|(00:30:bd:ca:1e:1e)|(00:30:bd:ca:1e:1e)]
IP - [192.168.7.41)|(192.168.7.3)]
ARP - [00:0f:a3:1f:b4:ff)|(192.168.7.3)]-[00:00:00:00:00:00)|(192.168.7.41)]
```

4.7 Варианты заданий.

1) Сделать программу для перехвата ICMP и TCP, указать флаги TCP, адрес назначения.

2) Сделать программу для перехвата ICMP и TCP, указать порты TCP, адрес отправителя.

3) Сделать программу для перехвата ICMP и UDP, указать порты UDP, адрес отправителя и получателя.

4) Сделать программу для перехвата ICMP и TCP, указать порты UDP, адрес отправителя и получателя.

5) Сделать программу для перехвата IGMP (формировать самостоятельно) и UDP, указать порты UDP, адрес отправителя и получателя.

6) Сделать программу для перехвата ICMP и TCP, указать адреса.

7) Сделать программу для перехвата ICMP и UDP и TCP указать порты, адрес отправителя и получателя.

8) Сделать программу для перехвата ICMP, сформировать TCP и пакет со временм жизни.

9) Сделать программу для перехвата IP и UDP, указать порты UDP, адрес отправителя и получателя.

10) Сделать программу для перехвата UDP, TCP указать порты UDP, адрес отправителя и получателя.

11) Сделать программу для перехвата IP и TCP, указать порты, адрес отправителя и получателя.

12) Сделать программу для перехвата DNS и UDP, указать порты UDP, адрес отправителя и получателя.

13) Сделать программу для перехвата DNS и ICMP.

14) Сделать программу для перехвата IP и DNS, указать порт, адрес отправителя и получателя.

15) Сделать программу для перехвата TCP и UDP, указать флаги TCP, адрес отправителя и получателя.

16) Сделать программу для перехвата ICMP, IGMP и TCP, указать адрес отправителя и получателя.

17) Сделать программу для перехвата UDP, IP, ICMP, указать порты UDP, адрес отправителя и получателя.

18) Сделать программу для перехвата ARP.

19) Сделать программу для перехвата DNS и ICMP, указать адрес отправителя и получателя, тип запроса ответа ICMP

20) Сделать программу для перехвата ICMP и IP, TCP, указать порты адрес отправителя и получателя, для IP указать NextHeader.

5. Лабораторная работа 5. Аутоинтефикация и авторизация

Реализовать аутоинтефикацию, используя основные механизмы протоколов аутентификации.

5.1 SASL

SASL (англ. Simple Authentication and Security Layer — простой уровень аутентификации и безопасности) — это фреймворк (каркас) для предоставления аутентификации и защиты данных в протоколах на основе соединений. Он разделяет механизмы аутентификации от прикладных протоколов, в теории позволяя любому механизму аутентификации, поддерживающему SASL, быть использованным в любых прикладных протоколах, которые используют SASL. Фреймворк также предоставляет слой защиты данных. Для использования SASL протокол включает команду для идентификации и аутентификации пользователя на сервере и для опциональной защиты переговоров последующей интерактивности протокола. Если это используется в переговорах, то слой безопасности вставляется между протоколом и соединением.

В 1997 Джон Гардинер Майерс (John Gardiner Myers) написал изначальную спецификацию SASL (RFC 2222) при университете Карнеги-Меллона (Carnegie Mellon University). В 2006 году этот документ утратил силу после введения RFC 4422, под редакцией Алексея Мельникова (Alexey Melnikov) и Курта Зейлинга (Kurt Zeilenga).

Механизмы SASL реализуют серию запросов и ответов. Определенные SASL механизмы включают:

«EXTERNAL», используется, когда аутентификация отделена от передачи данных (например, когда протоколы уже используют IPsec или TLS);

«ANONYMOUS», для аутентификации гостевого доступа (RFC 4505);

«PLAIN», простой механизм передачи паролей открытым текстом. PLAIN является заменой устаревшему LOGIN ;

«OTP», механизм одноразовых паролей. OTP заменяет устаревший механизм SKEY;

«SKEY», система одноразовых паролей (устаревший);

«CRAM-MD5»;

«DIGEST-MD5»;

«NTLM»;

«GSSAPI»;

GateKeeper (& GateKeeperPassport), разработана Microsoft для MSN Chat;

«KERBEROS IV» (устаревший).

Семья механизмов GS2 поддерживает произвольные GSSAPI механизмы в SASL. Это сейчас стандартизовано в RFC 5801.

5.2 HTTP аутентификация и SASL(Simple Authentication and Security Layer) аутентификация

Basic — базовая аутентификация, при которой имя пользователя и пароль передаются в заголовках *http-пакетов*. Пароль при этом не шифруется и присутствует в чистом виде в кодировке *base64*. Для данного типа аутентификации использование *SSL* является обязательным.

Digest — дайджест-аутентификация, при которой пароль пользователя передается в хешированном виде. По уровню конфиденциальности паролей этот тип мало чем отличается от предыдущего, так как атакующему все равно, действительно ли это настоящий пароль или только *хеш* от него: перехватив удостоверение, он все равно получает доступ к конечной точке. Для данного типа аутентификации использование *SSL* является обязательным. Но в данном случае можно перехватывать только текущую сессию, для другой сессии данный хэш не поможет, так как сервер посылает случайную строку, которая хэшируется с логином и паролем. Если говорить более точно, то можно привести в пример CRAM-MD5 или DIGEST-MD5. Напомним, что хэш MD5 является «однозначным» преобразованием строки в псевдослучайную строку, при этом обратного преобразования не существует, хотя в большинстве случаев это уже не так и MD5 не столь надежен, лучше использовать другие хэши, применяемые, например, в технологии блокчейн (SHA256 и другие). Первый протокол предполагает отправку случайной строки сервером, после чего эта строка хэшируется паролем пользователя, не будем вдаваться в подробности того, как преобразуется эта строка предварительно, но в итоге сервер должен таким же образом хэшировать отправленную им самим строку с паролем пользователя и соответственно его знать, очевидно, здесь можно перехватить сессию и при этом пароль хранится в открытом виде на сервере. Digest предполагает отправку клиенту случайной строки, но предварительно клиент хэширует вместе пароль и логин пользователя, затем полученный хэш хэширует вместе со случайной строкой сервера, затем все это вместе хэширует со своей случайной строкой, отправляет серверу полученный хэш и свою случайную строку в открытом виде, сервер проводит ту же операцию вместе со строкой

полученной от клиента, при этом он может хранить хэш логина пароля, а не открытый логин пароль. Перехват от клиента хэша позволит перехватить сессию, типичная атака MITM (man in the middle, человек по середине). Все это выглядит на стороне клиента так H(H(login, pass)+server_string)+client_string. Здесь приведен пример, не учитывающий различные параметры протоколов, которые так же входят в хэш, например, realm, но пример приведен для понимания концепции.

Integrated — интегрированная аутентификация, при которой клиент и сервер обмениваются сообщениями для выяснения подлинности друг друга с помощью протоколов *NTLM* или *Kerberos*.

Этот тип аутентификации защищен от перехвата удостоверений пользователей, поэтому для него не требуется протокол *SSL*. Здесь используются доверенные центры распределения ключей.

Только при использовании данного типа аутентификации можно работать по схеме *http*, во всех остальных случаях необходимо использовать схему *https*.

Параметры требующиеся для Basic и Digest аутентификации передаются в заголовках сервера.

Например,

GET /images/image1.jpg HTTP/1.1

Authorization: Basic Zm9vOmJhcg==

Здесь в base64 передается логин и пароль пользователя, foo: bar.

Пример с Digest будет посложнее. Например, вы обратились к ресурсу и вам вернулося следующий ответ.

HTTP/1.0 401 Unauthorized

WWW-Authenticate: Digest realm="my@my.com",

qop="auth,auth-int",

nonce="xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",

opaque="yyyyyyyyyyyyyyyyyyyyyyyyyyyy"

В ответ вы уже пытаетесь пройти процедуру аутентификации:

GET /dir/index.html HTTP/1.0

Host: localhost

Authorization: Digest username="my_user",

realm="my@my.com",

nonce=" xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx",
uri="/dir/index.html",
qop=auth,
nc=00000001,
cnonce="0a4f113b",
response="calculated by hash function response",
opaque="yyyyyyyyyyyyyyyyyyyyyyyyyyyy"

Вычисление ответа response осуществляется следующим образом.
HA1 = MD5("my_user:my@my.com:password")

HA2 = MD5("GET:/dir/index.html")
Response = MD5("HA1:
cmFuZG9tbHlnZW5lcmF0ZWRub25jZQ:
nc:cnonce:auth:
HA2")

5.3 Cookie-based авторизация

Cookie-based авторизация - это форма авторизации, которая использует куки или небольшие данные, хранящиеся в браузере пользователя, чтобы аутентифицировать его и держать его в системе. При входе в систему на компьютере пользователя создается куки и отправляется на сервер. Когда пользователь посещает сайт в будущем, сервер будет искать куки и, если она существует, позволит пользователю получить доступ к сайту, не выходя из системы.

Cookie-based авторизация работает следующим образом: при аутентификации на основе cookies уникальный идентификатор (файл cookie) создается на стороне сервера и отправляется в браузер. Этот файл cookie содержит уникальный идентификатор сеанса, который сохраняется на компьютере пользователя и используется для проверки подлинности при каждом запросе к серверу. Когда пользователь входит на сайт, сервер отправляет ему файл cookie, который содержит уникальный идентификатор сеанса. Этот идентификатор сохраняется на компьютере пользователя и используется для проверки подлинности при каждом запросе к серверу. Пользователь может оставаться авторизованным до тех пор, пока не произойдет выход из системы или пока файл cookie не

будет удален.

Для реализации cookie-based авторизации на Python можно использовать библиотеку Requests. Для этого нужно отправить POST-запрос на страницу входа, передав в теле запроса учетные данные пользователя (логин и пароль). Затем необходимо сохранить полученный файл cookie и использовать его для последующих запросов к сайту. Пример кода для аутентификации на сайте с помощью библиотеки Requests:

```
import requests

# Адрес страницы входа
login_url = 'https://example.com/login'

# Учетные данные пользователя
payload = {
    'username': 'user',
    'password': 'pass'
}

# Отправляем POST-запрос на страницу входа
session = requests.Session()
response = session.post(login_url, data=payload)

# Сохраняем файл cookie
cookie = session.cookies.get_dict()

# Используем файл cookie для последующих запросов к сайту
response = session.get('https://example.com/profile', cookies=cookie)
```

5.4 JWT токен авторизация

JWT (JSON Web Token) - это открытый стандарт для создания токенов доступа, основанный на формате JSON. Он используется для передачи информации между клиентом и сервером в зашифрованном виде. JWT состоит из трех частей: заголовка, полезной нагрузки и подписи. Заголовок содержит информацию о типе токена и используемом алгоритме шифрования. Полезная нагрузка содержит данные, которые нужно передать между клиентом и сервером. Подпись гарантирует целостность данных.

Примеры использования JWT-токенов включают в себя авторизацию пользователей

на сайтах, API-интерфейсы и мобильные приложения.

При работе с JWT-токенами происходят следующие шаги:

1. Пользователь отправляет запрос на сервер для авторизации.

2. Сервер проверяет учетные данные пользователя и создает JWT-токен.

3. Сервер отправляет JWT-токен обратно пользователю.

4. Пользователь сохраняет JWT-токен в локальном хранилище (например, в localStorage).

5. При каждом последующем запросе к серверу пользователь отправляет JWT-токен

Для создания JWT-токена в Python можно использовать библиотеку PyJWT

Вот пример кода, который создает JWT-токен:

```
import jwt

# Определение заголовка и полезной нагрузки токена
header = {'alg': 'HS256'}
payload = {'sub': '1234567890', 'name': 'John Doe', 'iat': 1516239022}

# Определение секретного ключа для подписи токена
secret_key = 'mysecretkey'

# Создание JWT-токена с помощью PyJWT
jwt_token = jwt.encode(payload, secret_key, algorithm='HS256', headers=header)

print('JWT Token:', jwt_token)
```

В этом примере мы определяем заголовок и полезную нагрузку токена в виде словарей Python. Затем мы определяем секретный ключ, который будет использоваться для подписи токена. Мы используем функцию jwt.encode() из библиотеки PyJWT для создания JWT-токена. Функция принимает на вход полезную нагрузку, секретный ключ и алгоритм хэширования (в данном случае HS256). Мы также передаем заголовок токена через параметр headers.После выполнения кода выше будет напечатано значение JWT-токена.Обратите внимание, что при создании JWT-токенов необходимо использовать безопасные методы генерации случайных чисел для генерации секретных ключей.

5.5 HTTP Authorization

Заголовок HTTP-запроса Authorization используется для передачи учетных данных, которые аутентифицируют пользовательский агент на сервере.

Этот заголовок может использоваться для различных схем аутентификации, таких как Basic и Bearer.

В заголовке Authorization обычно указывается тип схемы аутентификации (например, Basic или Bearer) и учетные данные пользователя (например, имя пользователя и пароль или JWT-токен).

Например, в случае с JWT-токеном заголовок Authorization может выглядеть следующим образом:

Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0I

5.6 Авторизация Auth 2.0

Является открытым фреймворком (протоколом) авторизации, позволяющим получить сторонним приложениям ограниченный доступ к ресурсам HTTP-сервиса, позволяет предоставить права на использование некоторого ресурса (например, API какого-либо сервиса). При этом в общем случае нельзя определить, кто в настоящий момент пользуется правами.

Схематично сценарий протокола OAuth 2.0 представлен на рисунке. Он описывает взаимодействие между упомянутыми выше четырьмя сторонамии включает следующие шаги.

1) Клиент запрашивает авторизацию у владельца ресурса. Запрос авторизации может быть направлен владельцу ресурса напрямую, как показано на рисунке, или косвенно через сервер авторизации. Предпочтительным является второй вариант.

2) Клиент получает разрешение на доступ (grant), структуру данных, представляющую авторизацию владельца ресурса, выраженную с использованием одного из четырех типов разрешений: код авторизации (authorization code), неявное разрешение (implicit), пароль владельца ресурса (resource owner password credentials) и учетные данные клиента (client credentials), так же есть еще Device authorization для доступа устройств (если нет веб-браузера, RFC 8628) . Тип разрешения на доступ зависит от метода, используемого клиентом для запроса авторизации, и типов разрешений, поддерживаемых сервером авторизации. Типы разрешений, поддерживаемые сервером авторизации, определяются при его разработке, исходя из его прикладных целей и задач. Настоящий

стандарт регламентирует использование в качестве типа разрешения код авторизации.

3) Клиент запрашивает токен доступа посредством аутентификации на сервере авторизации и предоставления разрешения на доступ.

4) Сервер авторизации аутентифицирует клиента, проверяет разрешение на доступ и, если оно действительно, выдает токен доступа.

5) Клиент запрашивает защищенный ресурс на сервере ресурсов и аутентифицируется, представляя токен доступа.

6) Сервер ресурсов проверяет токен доступа и, если он действителен, обслуживает запрос.

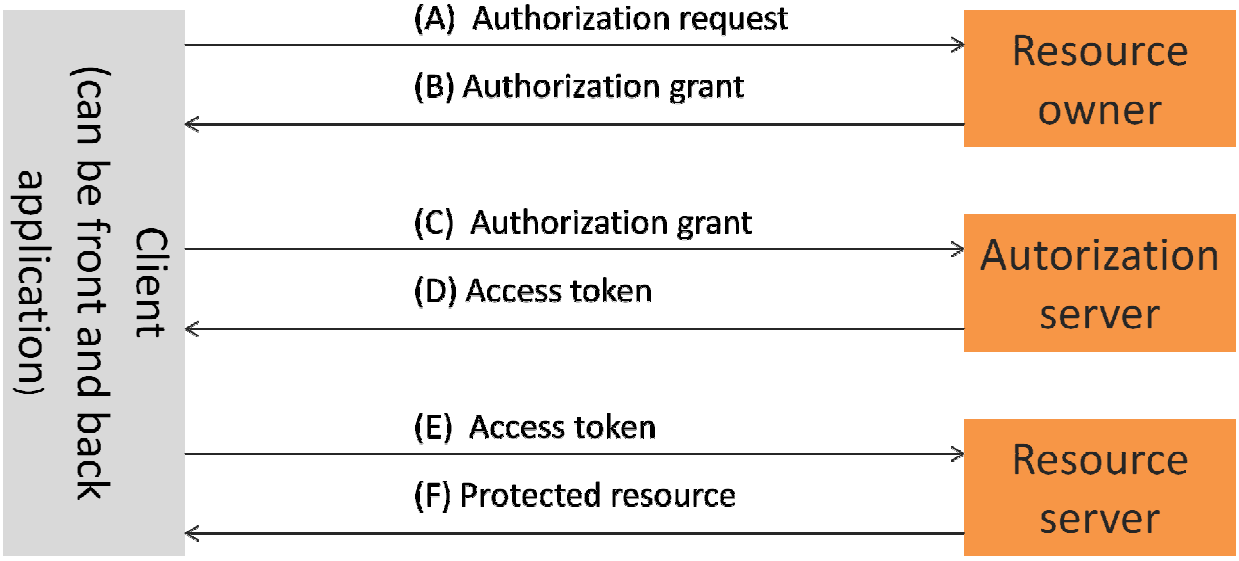

Рисунок 5.1 –Абстрактное описание протокола

Стандарт OAuth 2.0 определяет следующие четыре роли:

• владелец ресурса — сущность, обладающая правом на выдачу доступа к защищенным ресурсам. В случае, если владелец является человеком, его называют конечным пользователем;

• сервер ресурсов — сервер, содержащий защищаемые ресурсы и обладающий возможностью получения и формирования ответа на запросы к защищаемым ресурсам посредством использования маркера доступа;

• клиент — приложение, осуществляющее доступ к защищенным ресурсам от имени Владельца. Термин "клиент" явно не определяет какое-либо конкретное исполнение (будь то сервер, персональный компьютер или мобильное приложение); Приложением может являться браузер, толстый клиент или мобильное приложение. Клиент может быть public и confedential. Public не может безопасно хранить свои учётные данные. Этот клиент работает на устройстве владельца ресурса, например, это браузерные или мобильные приложения. Confedential может безопасно хранить свои учетные данные, например

бэкенд приложения.

• сервер авторизации — сервер, осуществляющий выпуск маркеров доступа для клиентских приложений после успешной аутентификации и авторизации Владельца ресурсов.

Протокол OAuth обладает возможностью аутентификации не только Пользователя, но и клиентского приложения, осуществляющего доступ к ресурсам.

Рассмотрим пять способов получения доступа (grant) в auth 2.0, начиная с самого простого.

5.6.1 Client credentials grant flow

Предполагает самый простой способ получения прав доступа к ресурсам. Клиент отправляет на сервер авторизации client id и client secret, на что возвращается access token с которым клиент может обратиться к серверу ресурсов. Данный способ используется для доступа к собственным ресурсам или предоставление доступа к серверу ресурсов согласованному с сервером авторизации, когда клиент совпадает с владельцем ресурсов. Допускается только для защищенных клиентов (confedential, клиент, который может безопасно хранить свои учётные данные, к такому типу клиентов относят web-приложения, имеющие backend.).

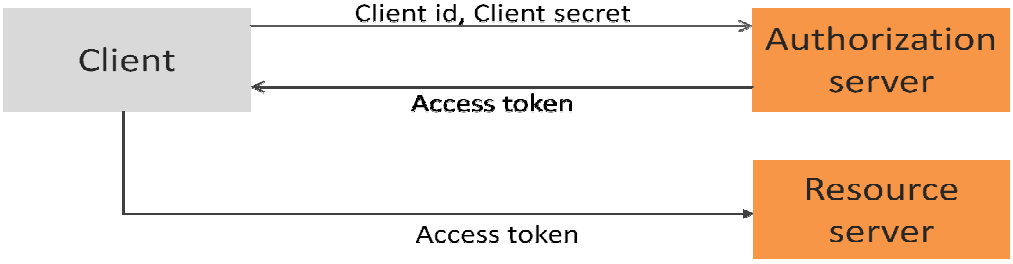

Рисунок 5.2 – Схема AUTH 2.0 взаимодействия где клиент является владельцем ресурсов

Здесь не указано как и в других примерах взаимодействие сервера и авторизации и сервера ресурсов, но очевидно оно должно быть для согласования режима доступа передаваемого в scope. Кроме того, сервер ресурсов и сервер авторизации могут быть одним и тем же. Auth 2.0 не накладывает органичения на роли указанных взаимодействующих частей.

5.6.2 Authorization code flow

Далее рассмотрим Authorization code flow, самая распространенная схема и одна из защищенных, за счет отправки дополнительно между клиентом и сервером авторизации client_id и client_secret, но и она может быть подвергнута атакам, связанным с неправильным использованием redirect_uri. На рисунке приводится схема взаимодействия с учтом того, что клиент представляет собой веб приложение с фронт и бэк частью.

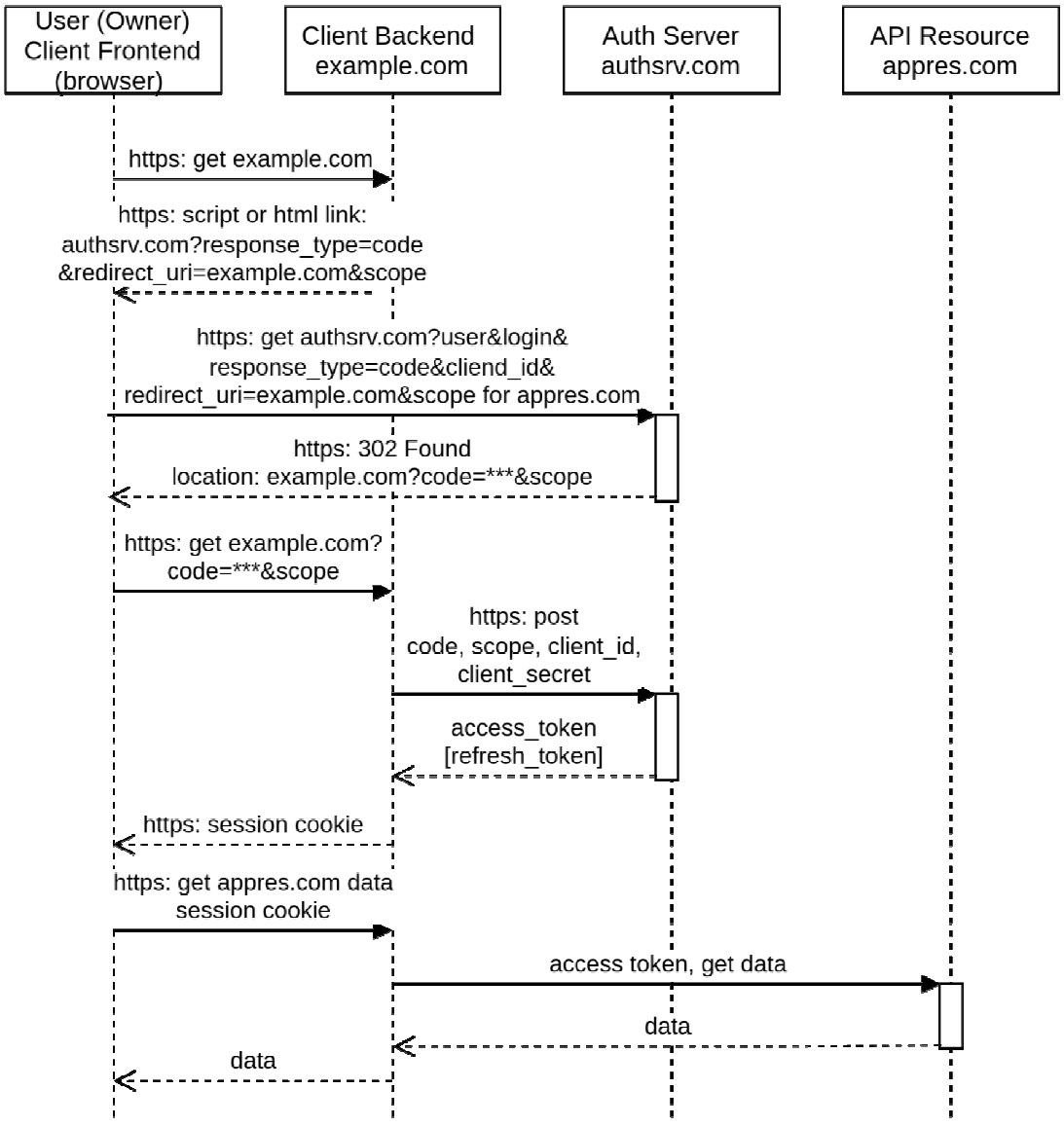

Рисунок 5.3 – Схема взаимодействия получения прав доступа Authorization code flow

На первом шаге клиент перенаправляет resource owner с помощью user-agent на страницу аутентификации Authorization server. В URI он указывает client ID и redirection URI. Redirection URI используется для понимания, куда вернуть resource owner после того, как авторизация пройдёт успешно (resource owner выдаст разрешение на scope, запрашиваемый клиентом).

Взаимодействуя с сервером авторизации через user-agent, resource owner проходит аутентификацию на сервере авторизации.

Resource owner проверяет права, которые запрашивает клиент на consent screen и разрешает их выдачу.

Resource owner возвращается клиенту с помощью user-agent обратно на URI, который был указан как redirection URI. В качестве query-параметра будет добавлен authorization code — строка, подтверждающая то, что resource owner выдал необходимые права сервису.

С этим authorization code клиент отправляется на сервер авторизации, чтобы получить в ответ access token (ну и refresh token, если требуется).

Сервер авторизации валидирует authorization code, убеждаясь, что токен корректный и выдаёт клиенту access token (и опционально refresh token). С его помощью клиент сможет получить доступ к заветному ресурсу.

Еще одним способом доступа является implicit grant flow (неявный доступ), но он не рекомендуется. В данном случае вместо кода сразу возвращается аксес и рефреш токен, это небезопасно.

В другом варианте, который уже запрещен пароль владельца ресурса (resource owner password credentials) пароль и логин передается через клиента на сервер авторизации, что, совсем небезопасно потому, что пароль и логин пользователя становятся известны клиентскому приложению.

Появились схемы, которые усиливают защиту доступа по авторскому коду.

5.6.3 Authorization Code Flow with Proof Key for Code Exchange (PKCE).

Это усовершенствованный вариант Authorization code grant flow. В нем происходит генерация еще двух случайных кодов: Code Verifier и Code Challenge. Авторизация реализуется с посылкой Code Challenge, после введения пользователем логина и пароля авторизационным сервером отдается Authorization Code, далее идет обращение за аксес токеном с только что выданным Authorization Code и с Code Verifier. В результате Identity-провайдер (Авторизационный сервер) валидирует на основании трех кодов.

Если злоумышленник получит один код из системы, он ничего не сможет сделать с ним — нужны все три кода. Примерами, где можно задействовать такой flow, являются Single Page Applications: Angular приложения, React JS приложения, и нативные приложения под Android/iOS.

Полный процесс описанный в rfc7637:

A. Клиент создает и записывает секрет с именем «code_verifier» и получает преобразованную версию "t(code_verifier)" (или «code_challenge»), которая отправляется по протоколу OAuth 2.0 в запросе авторизации вместе с методом преобразования "t_m".

B. Конечная точка авторизации отвечает как обычно, отправляя код авторизации, но записывает "t(code_verifier)" и метод преобразования "t_m".

C. Затем клиент отправляет код авторизации как обычно в точку токена авторизации, но включает сгенерированный секрет «code_verifier» из шага (A).

D. Сервер авторизации преобразует «code_verifier» и сравнивает его в "t(code_verifier)" из (B). Доступ запрещен, если они не равны.

Злоумышленник, перехватывающий код авторизации в точке (B), не может обменять его на токен доступа, так как не владеет секретом "code_verifier".

То есть перед запросом авторизации клиенту надо сформровать code_verifier и t(code_verifier) = code_challenge (на бэке приложения) и выслать на аторизирующий сервер вместе с методом преобразования t. Затем сервер авторизации вышлет code (запомнив code_challenge). Приложение-клиент (бэк приложения) далее должно выслать на авторизационный сервер code и code_verifier. Авторизационный сервер проверит code и t(code_verifier) сравнив с code_challenge. Обратно высылает access_token.

Данный вид авторизации использует два вида токенов, access и refresh токен. Позволяет удобно реализовать концепцию микросервисной архитектуры, где будет существовать отдельный сервер авторизации. Данный сервер выдает клиенту или клиентскому приложению access токен и refresh токен (хотя рефреш токен опционален). Добавление refresh токена позволяет миновать процедуру запроса пароля пользователя и ввода его вручную. Обычно предполагается создание авторизирующего сервера, который на логин и пароль выдает данные токены. Первый выдается на короткое время и служит для того, чтобы аутентифицироваться на ресурсных серверах, когда его время заканчивается, то используется уже refresh токен для его обновления. Обычно в такие токены включают сведения о пользователе, например его идентификатор и прочую информацию, чтобы не искать ее в базе. Refresh токен как уже было сказано нужен для того, чтобы обновление jwt токена было реализовано автоматически, по прошествии времени. Очевидно, что тут нужно использовать асинхронные методы обмена. При этом если злоумышленник получил каким-то образом access токен, то через какое-то время он станет невалидным, так как будет выпущен новый токен и старый уже не будет приниматься, в этом случае доступ может быть заморожен. При получении злоумышлинником refresh токена, он может обновить токен, но опять же у легального

пользователя устареет его access токен и в то же время его refresh токен и ему придется авторизоваться, так же можно получив от того же пользователя устаревший рефреш токен можно заморозить доступ на всех устройствах. Для дополнительной защиты часто используют так называемый finger print, уникальную информацию с устройства пользователя или клиента, дополнительно проверяя и ее.

Хотя refresh токен не обязательно делать невалидным после обновления, но в большинстве случаев отправляют и новый refresh и access токен и время жизни и после истечения этого времени, аксес и рефреш токен обновляются, либо по отправке рефреш токена, либо при заходе со страницы авторизации (хотя рефреш токен можно оставлять прежним, а можно как уже было сказано вовсе не использовать). Таким образом, например, если аксес токен стал невалидным по истечении времени, то пользователя или клиент пользователя отправят страницу авторизации или будет выслан рефреш токен. Если, например, злоумышленник перехватил рефреш токен, то у легального пользователя рефреш токен станет невалидным и его отправят на страницу авторизации с требованием ввести логин и пароль.

5.7 Open ID Connect

Протокол OpenID Connect (OIDC)

OIDC – семейство протоколов, являющихся расширением протоколов OAuth 2.0, позволяющих расширить их функционал путем более точного описания процесса аутентификации владельца ресурса и возможности клиенту получить информацию о нем. Это этапы (1) – (4) протокола OAuth 2.0.

Схематично протокол OpenID Connect можно описать следующим образом

1) Клиент отправляет серверу авторизации запрос аутентификации.

2) Сервер авторизации аутентифицирует конечного пользователя и получает согласие пользователя на доступ клиента к запрошенному ресурсу.

3) Сервер авторизации отвечает клиенту ID токеном и (опционально) токеном доступа.

4) Клиент может отправить серверу авторизации запрос информации о пользователе по токену доступа.

5) Сервер авторизации возвращает клиенту информацию о конечном пользователе.

Сервер авторизации OpenID Connect поддерживает три сценария аутентификации, реализующие этот сценарий: с генерацией кода авторизации (Authorization Code Flow), неявный сценарий (Implicit Flow), гибридный сценарий (Hybrid Flow). Передача сообщений между клиентом и сервером авторизации (на конечных точках авторизации,

токена и UserInfo) должна производиться с использованием протокола TLS.

5.8 Задание.

Общее задание: реализовать Authorization Code Flow with Proof Key for Code Exchange и далее по вариантам.

1) Реализовать HTTP Authorization Digest
2) Реализовать CRAM-MD5
3) Реализовать Authorization Basic и Cookie-based
4) Реализовать Authorization Bearer
5) Реализовать Authorization Code flow Auth2.0
6) Реализовать Client credentials flow Auth2.0
7) Реализовать implicit flow Auth2.0
8) Реализовать проверку фингерпринта при авторизации устройств
9) Реализовать авторизацию, используя jwt.
10) Реализовать выдачу access и refresh и их обновление со временем

6. Лабораторная работа №6 распределенный UDP сервер/ UDP клиент

Цель работы: написать приложения клиенты и серверы, имитирующие работу grid системы, обеспечить надежность доставки данных или пакетов, автоматическое обнаружение расчетчиков;

User Datagram Protocol

Транспортный протокол для передачи данных в сетях IP без установления соединения. Он является одним из самых простых протоколов транспортного уровня модели OSI.

В отличие от TCP, UDP не подтверждает доставку данных, не заботится о корректном порядке доставки и не делает повторов. Поэтому аббревиатуру UDP иногда расшифровывают как Unreliable Datagram Protocol (протокол ненадёжных датаграмм). Зато отсутствие соединения, дополнительного трафика и возможность широковещательных рассылок делают его удобным для применений, где малы потери, в массовых рассылках локальной подсети, в медиапротоколах и т.п.

Распределенный UDP сервер/ UDP клиент

Задача написать приложение раздающее задание, которое посылало бы данные на несколько приложений расчетчиков, затем все расчетчики производили бы какие-либо преобразования над данными и отсылали их обратно. В случае, если приложение раздающее задание не получило обработанные данные обратно, оно должно отправить начальные данные другому расчетчику.

Обеспечение надежности доставки

Для обеспечения надежности доставки можно использовать следующие методы: хэш-функции, кода Хэмминга, контрольные суммы. На примере хэш-функции:

Вычисляем хэш-функцию от данных, отправляем ее вместе с данными, получатель вычисляет свою хэш-функцию от полученных данных, сравнивает ее с полученной хэш-функцией, если все передалось успешно - они совпадут.

Автоматическое обнаружение расчетчиков

Например:

Создаем сервер с заранее известным портом. И несколько клиентов, один из которых будет раздавать задания, а остальные производить расчеты. Все клиенты регистрируются на сервере, как «расчетчики» и один, как «раздающий задания». Затем сервер передает данные о расчетчиках раздающему задания, тот в свою очередь посылает данные для расчета напрямую расчетчикам.

Данный вариант не является эталонным. Принимаются любые другие, лишь бы выполняли поставленную задачу.

Рекомендуемая литература:

http://helper10.narod.ru/i27.htm

http://tools.ietf.org/html/rfc768

http://docs.oracle.com/javase/1.4.2/docs/api/java/net/DatagramSocket.html

http://docs.oracle.com/javase/1.4.2/docs/api/java/net/DatagramPacket.html

Пример Java:

Для работы с датаграммными сокетами приложение должно создать сокет на базе класса DatagramSocket, а также подготовить объект класса DatagramPacket, в который будет записан принятый от партнера по сети блок данных. Канал, а также входные и выходные потоки создавать не нужно. Данные передаются и принимаются методами send и receive, определенными в классе DatagramSocket.

В классе DatagramSocket определены два конструктора, прототипы которых представлены ниже:

public DatagramSocket(int port);

public DatagramSocket();

Первый из этих конструкторов позволяет определить порт для сокета, второй предполагает использование любого свободного порта.

Обычно серверные приложения работают с использованием какого-то заранее определенного порта, номер которого известен клиентским приложениям. Поэтому для серверных приложений больше подходит первый из приведенных выше конструкторов.

Клиентские приложения, напротив, часто применяют любые свободные на локальном узле порты, поэтому для них годится конструктор без параметров.

Прием и передача данных на датаграммном сокете выполняется с помощью методов receive и send, соответственно:

public void receive(DatagramPacket p);

public void send(DatagramPacket p);

В качестве параметра этим методам передается ссылка на пакет данных (соответственно, принимаемый и передаваемый), определенный как объект класса DatagramPacket. Этот класс будет рассмотрен позже.

Еще один метод в классе DatagramSocket, которым вы будете пользоваться, это метод close, предназначенный для закрытия сокета:

public void close();

Перед тем как принимать или передавать данные с использованием методов receive и send вы должны подготовить объекты класса DatagramPacket. Метод receive запишет в такой объект принятые данные, а метод send - перешлет данные из объекта класса DatagramPacket узлу, адрес которого указан в пакете.

Подготовка объекта класса DatagramPacket для приема пакетов выполняется с помощью следующего конструктора:

```
public DatagramPacket(byte ibuf[],
 int ilength);
```

Этому конструктору передается ссылка на массив ibuf, в который нужно будет записать данные, и размер этого массива ilength.

Если вам нужно подготовить пакет для передачи, воспользуйтесь конструктором, который дополнительно позволяет задать адрес IP iaddr и номер порта iport узла назначения:

```
public DatagramPacket(byte ibuf[],
  int ilength,
  InetAddress iaddr, int iport);
```

Таким образом, информация о том, в какой узел и на какой порт необходимо доставить пакет данных, хранится не в сокете, а в пакете, то есть в объекте класса DatagramPacket.

Помимо только что описанных конструкторов, в классе DatagramPacket определены четыре метода, позволяющие получить данные и информацию об адресе узла, из которого пришел пакет, или для которого предназначен пакет.

Метод getData возвращает ссылку на массив данных пакета:

```
public byte[] getData();
```

Размер пакета, данные из которого хранятся в этом массиве, легко определить с помощью метода getLength:

```
public int getLength();
```

Методы getAddress и getPort позволяют определить адрес и номер порта узла, откуда пришел пакет, или узла, для которого предназначен пакет:

```
public InetAddress getAddress();
public int getPort();
```

Если вы создаете клиент-серверную систему, в которой сервер имеет заранее известный адрес и номер порта, а клиенты - произвольные адреса и различные номера портов, то после получения пакета от клиента сервер может определить с помощью методов getAddress и getPort адрес клиента для установления с ним связи.

Варианты заданий.

1) Расчет определенного интеграла методом левых прямоугольников.

2) Расчет определенного интеграла методом правых прямоугольников.

3) Расчет определенного интеграла методом трапеций.

4) Сглаживание с помощью метода скользящего среднего (сумма соседних точек)

5) Медианное сглаживание (упорядочивание в окне и получение значения центрального элемента)

6) Сортировка методом Хоара (быстрая сортировка). Дерево расчетчиков.

7) Генетический алгоритм. Поиск минимума квадратичной функции. Если сложно, то вместо этого умножение матриц (распараллелить).

8) Клональный алгоритм. Поиск минимума квадратичной функции. Если сложно, то распараллелить сложение матриц.

9) Поиск значения в упорядоченном массиве. Дерево расчетчиков.

10) Поиск значения хэш-суммы (от случайной строки и данных) соответствующей заданному условию сравнения.

7. Лабораторная работа №7 Прокси сервера

Цель работы: написать консольные приложения - кэширующий HTTP прокси-сервер и SOCKS5, SOCKS4 прокси-сервер, HTTP Connect proxy;

7.1 HTTP Кэширующий Прокси

HTTP прокси реализуется в соответствии с HTTP протоколом и позволяет клиентам получать через себя доступ к веб серверам, таким образом, клиент по HTTP протоколу отправляет прокси- серверу те же HTTP запросы, какие бы он посылал на требуемый сервер.

Задача прокси -сервера транслировать эти запросы на запрашиваемый сервер и осуществлять транспорт HTTP трафика через себя клиенту, при этом прокси сервер может изменять параметры HTTP запросов клиентов и ответа сервера и транслировать клиенту модифицированные объекты и заголовки ответа. Задача кэширующего прокси сервера сохранять проходящие через него страницы и объекты, и если клиенты запросили объект, который уже был сохранен на сервере, то сразу отправлять этот объект клиенту не скачивая его с веб сервера в случае если этот объект на веб сервере не менялся, при этом если объект на веб сервере обновился, информацию об этом необходимо получать с помощью заголовков if-modified-since, if-none-match.

В языках программирования уже есть готовые компоненты для работы с HTTP сервером, которые позволяет переопределять HTTP обработчик запросов и получать доступ к передаваемым данным, полям запроса, анализировать их и делать новый запрос к серверу в качестве HTTP клиента.

7.2 SOCKS5 прокси-сервер

Сокс сервер просто пробрасывает через себя TCP и UDP трафик. В данном приложении необходимо реализовать проброску tcp, используя протокол SOCKS5.

С его помощью можно решать самые разные задачи: организовывать защищенный доступ к службам, расположенным за межсетевым экраном (firewall), скрывать свой истинный IP- адрес во время работы с недружелюбными сетевыми ресурсами или реализовать универсальный прокси-сервер, поддерживающий любые протоколы прикладного уровня (HTTP, FTP, POP3/SMTP, ICQ и т.д.). Основная задача данного протокола – внедрить в «нормальный» процесс обмена данными некоего посредника, называемого SOCKS-сервером или SOCKS-прокси. Когда клиент (поддерживающее

SOCKS приложение: web-браузер Mozilla, клиент ICQ Miranda IM и др., см. ниже) желает отправить какую-либо информацию по сети, он устанавливает соединение не с реальным адресатом, а с SOCKS-сервером, который, в свою очередь, пересылает данные по назначению, но уже от своего имени. С точки зрения «настоящего» сервера (например, web-узла, который пользователь желает просмотреть в Mozilla Firefox) SOCKS-прокси является самым обыкновенным клиентом. Таким образом, сущность (IP-адрес) истинного клиента оказывается скрытой от обслуживающего его сервера. Это весьма удобное обстоятельство таит в себе потенциальную опасность (можете спрятаться вы, но ведь могут и от вас), поэтому реально существующие SOCKS-сервера имеют развитые схемы контроля доступа (запрет входящих и исходящих соединений по заданному перечню адресов) и поддерживают авторизацию пользователей по паролю (см. ниже).

Отметим, что коль скоро SOCKS работает на более низком по сравнению с прикладным (а именно, транспортном) уровне модели OSI, его поддержка не потребует никаких изменений в логике работы клиента, а тем более – сервера. Действительно, все что нужно – это модифицировать реализацию функций, отвечающих за создание сетевого подключения и отправку данных: connect(), bind(), send() и т.п. На практике это обычно достигается перехватом системных вызовов с их последующей подменой поддерживающими SOCKS пользовательскими аналогами. Никаких правок в исходном коде клиентских приложений, а тем более самого доступа к исходным текстам, как правило, не требуется. Эта мощная процедура известна как «соксификация» и будет подробно рассмотрена ниже.

Теперь, когда мы получили общее представление о SOCKS, можно перейти к болеедетальному рассмотрению данного протокола.

Спецификация SOCKS5

Протокол SOCKS5 подробно описан в RFC1928.

Как уже отмечалось ранее, SOCKS5 является протоколом транспортного уровня. Его «соседи» – TCP и UDP непосредственно используются для передачи данных, поступающих с прикладного уровня (от пользовательских приложений), а значит, SOCKS-прокси должен уметь корректно работать с каждым из них. Отметим также, что протокол ICMP,

используемый утилитами ping и traceroute, находится ниже транспортного уровня, а потому соксификации, к сожалению, не поддается.

Прежде чем отправлять какие-либо данные, клиент должен пройти процедуру авторизации на SOCKS-сервере. Для этого он открывает TCP-соединение с портом 1080 (значение по умолчанию) SOCKS-сервера и отправляет по нему сообщение, содержащее

кодовые номера поддерживаемых им методов аутентификации. SOCKS-сервер выбирает один из методов по своему усмотрению и сообщает его номер клиенту. аутентификация может отсутствовать (на практике это скорее всего означает, что SOCKS-сервер различает клиентов по их IP-адресам) или производиться на основании имени пользователя и пароля. В последнем случае возможно большое количество различных вариантов, от тривиального «Username/Password Authentication» (RFC 1929) предусматривающего передачу пароля в открытом виде до куда более безопасного CHAP (зашифрованный пароль, открытые данные) и GSSAPI (RFC 1961), которое может использоваться для полной криптографической защиты трафика. После успешной авторизации клиент получает возможность посылать запросы (команды), устанавливать исходящие соединения и даже принимать входящие.

Установка исходящего TCP-соединения

Для установки исходящего TCP-соединения, клиент отправляет SOCKS-серверу запрос «CONNECT», в котором указывает адрес и порт доставки. Для идентификации узла- получателя могут использоваться как IP-адреса (поддерживаются IPv4/IPv6), так и полноценные доменные имена. В последнем случае SOCKS-сервер берет на себя заботу по их разрешению, так что сеть, в которой работает клиент, в принципе может обходиться и без DNS-сервера. В ответном сообщении SOCKS-сервер сообщает код ошибки (как обычно, 0 обозначает, что операция прошла успешно), а также IP-адрес (BND.ADDR) и TCP-порт (BND.PORT), которые будут использоваться для фактической связи с запрошенным узлом.

Поскольку SOCKS-сервера, как правило, имеют более одного сетевого интерфейса, данный IP-адрес может отличаться от того, с которым было установлено управляющее соединение. После этого клиент открывает новый TCP-сеанс с BND.ADDR:BND.PORT и осуществляет отправку данных. Исходящее TCP-соединение разрывается одновременно с закрытием управляющей сессии. Заметим, что запрос «CONNECT» может быть отклонен SOCKS- прокси, если адреса источника (клиента) или получателя (сервера) запрещены.

Установка исходящего UDP-соединения

В отличие от потокового протокола TCP, подразумевающего установку сеанса, протокол UDP является датаграммным, а потому несколько более сложным в обращении. Его поддержка появилась лишь в SOCKS5.

Перед отправкой UDP-датаграмм клиент запрашивает у SOCKS-сервера UDP-ассоциацию, используя для этого команду «UDP ASSOCIATE». UDP-ассоциация – это своего рода виртуальный сеанс между клиентом и SOCKS-сервером. В исходящем запросе клиент указывает предполагаемые адрес и порт, которые будут выступать в качестве

источника будущих UDP-датаграмм. Если на момент установки UDP-ассоциации эта информация еще не известна, клиент должен использовать комбинацию 0.0.0.0:0 (или, скажем, x.x.x.x:0, если неизвестен только номер порта). В ответном сообщении SOCKS-сервер указывает IP-адрес (BND.ADDR) и UDP-порт (BND.PORT), на которые следует направлять исходящие датаграммы. При этом адрес и порт их реального получателя указываются прямо в теле (можно сказать, что имеет место UDP-инкапсуляция). Эти параметры, наряду с адресом и портом отправителя используются для принятия решения о допустимости отправки датаграммы. это создает дополнительную нагрузку на SOCKS-сервер: правила фильтрации необходимо применять к каждой UDP-датаграмме, тогда как в случае TCP-соединения его легитимность оценивается один раз, в момент исполнения SOCKS-сервером команды «CONNECT». Согласно требованиям стандарта, SOCKS-сервер должен следить за тем, чтобы IP-адрес отправителя датаграммы совпадал с адресом узла, создавшего UDP-ассоциацию.

UDP-ассоциация разрушается одновременно с закрытием управляющего TCP-сеанса, в рамках которого была послана команда «UDP ASSOCIATE». Многие из существующих SOCKS-серверов испытывают серьезные проблемы, если между ними и клиентом, запросившим UDP-ассоциацию, располагается межсетевой экран с функцией NAT (Network Address Translation). Причина этого кроется в изменении адреса и порта отправителя, которое происходит в тот момент, когда UDP-датаграмма пересекает межсетевой экран. Как следствие, сервер и ничего не подозревающее клиентское приложение начинают говорить на разных языках: предполагаемый адрес и порт источника, указанные в команде «UDP ASSOCIATE» перестают соответствовать реальным параметрам получаемых

SOCKS-сервером датаграмм. В результате они оказываются отброшенными как не принадлежащие UDP-ассоциации. Проблему можно было бы решить, указав в качестве предполагаемого источника 0.0.0.0:0 (см. выше), что должно интерпретироваться SOCKS-сервером как «любая UDP-датаграмма, пришедшая с того же адреса, что и команда на создание ассоциации». К сожалению, большинство из реально существующих SOCKS-серверов трактуют стандарт более узко и не позволяют одновременно установить в ноль и предполагаемый адрес, и порт отправителя. Из протестированных автором реализаций описанный здесь «фокус с пробросом UDP через NAT» позволяет проделать лишь одна – Dante.

Прием входящих соединений

Эта достаточно оригинальная возможность может оказаться полезной в случаях, когда клиент и «настоящий» сервер в описанной выше схеме меняются местами, что

может произойти, например, в протоколах типа FTP. В целях дальнейшего рассмотрения будем предполагать, что между “клиентом” (стороной, собирающейся принять входящее соединение) и “сервером” (стороной, инициирующей входящее соединение) уже установлен “прямой” канал связи при помощи команды “CONNECT”. Для открытия “обратного” канала “клиент” должен послать SOCKS-серверу команду “BIND”, указав в ее параметрах IP-адрес и порт, которые будут использоваться им для приема входящего соединения. В ответ на это SOCKS- сервер сообщает IP-адрес и порт, выделенные им для поддержания “обратного” канала. Предполагается, что «клиент» передаст эти параметры «серверу» , используя средства, предоставляемые протоколами прикладного уровня (например, команду «PORT» протокола FTP). После того, как SOCKS-сервер примет (или отбросит) входящее соединение, он повторно уведомляет об этом «клиента», сообщая ему IP-адрес и порт, используемые “сервером”. Отметим, что прием входящих соединений может осуществлять лишь приложение, разработчики которого позаботились о поддержке SOCKS еще на этапе проектирования. В противном случае (если приложение работает с SOCKS-сервером через программу-соксификатор), оно не сможет предоставить корректную информацию об адресе ожидающего “обратной связи” сокета (т.е. сформирует неверную команду “PORT” в рассмотренном выше примере с FTP).

«Цепочки» SOCKS

Архитектура протокола SOCKS5 позволяет легко объединять SOCKS-сервера в каскады, или, как их еще называют «цепочки» («chains»). Примечательно, что все необходимые для этого действия могут быть произведены на стороне клиента. К «звеньям» цепочки предъявляется единственное требование: они должны «доверять» друг другу (т.е. допускать установку входящих и исходящих соединений). Если образующие каскад SOCKS-сервера не являются анонимными (т.е. используют схемы аутентификации Username/Password, CHAP или подобные), необходимо также, чтобы пользователь мог успешно пройти процедуру авторизации на каждом из них.

Предположим, что у нас имеется набор из N SOCKS-серверов с именами socks1, socks2, ...,socksN, удовлетворяющих всем вышеперечисленным требованиям. Тогда для создания каскада клиент может поступить следующим образом:

В случае исходящего TCP-соединения: клиент подключается к socks1, проходит (при необходимости) процедуру авторизации и посылает команду «CONNECT», указав в качестве адреса доставки socks2. Выполняя этот запрос, socks1 создаст новое соединение с socks2 и будет исправно передавать всю идущую по нему информацию клиенту, при этом socks2 не будет даже догадываться, с кем он общается на самом деле. Далее процедура повторяется до тех пор, пока не будет установлено соединение между socks(N-1) и socksN.

Последний сервер каскада подключается к непосредственно узлу, который интересует клиента. Передача данных происходит в обычном режиме: клиент отправляет пакет на сервер socks1, который, в свою очередь, передает его socks2, ... и так до тех пор, пока не будет достигнут конечный узел. В случае исходящего UDP-соединения: клиент подключается к socks1, проходит процедуру авторизации и последовательно посылает две команды: “CONNECT” (адрес доставки – socks2) и “UDP ASSOCIATE”. Таким образом, создаются два новых соединения: виртуальный UDP-канал между клиентом и socks1, а также TCP-сессия между socks1 и socks2. Используя эту TCP-сессию, клиент (от имени socks1) посылает команду “UDP ASSOCIATE” на сервер socks2 (открывает UDP-канал между socks1 и socks2) и “CONNECT” на сервер socks3. Процедура продолжается до тех пор, пока между всеми SOCKS-серверами каскада не будут установлены виртуальные UDPканалы.

Чтобы отослать какие-либо данные, клиент предварительно производит N-кратную инкапсуляцию UDP-датаграммы, указывая в качестве адреса доставки последовательно: socks1, socks2, socks3, socksN и адрес реального получателя, а затем отправляет ее на сервер socks1. Отметим, что на практике данный вариант каскадирования встречается крайне редко. Это связано с тем, что SOCKS-сервера, как и NAT Firewall'ы, могут изменить порт источника датаграммы, что приведет к проблемам, подробно описанным в разделе “Установка исходящего UDP-соединения”.

Используя цепочки SOCKS-серверов, не требующих аутентификации, клиент может значительно повысить анонимность работы в Интернете (см. эпиграф). В Сети можно найти множество программ, реализующих описанные здесь схемы. Таковыми, например, являются SocksChain (http://www.ufasoft.com/socks/) для Windows или ProxyChains (http://proxychains.sourceforge.net) для Unix. Каскадирование SOCKS-серверов является также неотъемлемой частью некоторых соксификаторов, в первую очередь, FreeCap (http://www.freecap.ru/).

Таблица 1. Некоторые методы SOCKS-аутентификации

Код

Название метода

0x00

No Authentication Required (Аутентификация отсутствует)

0x01

GSSAPI (RFC 1961)

0x02

Username/Password (RFC 1929)

0x03

Challenge-Handshake Authentication Method (CHAP)

0x05

Challenge-Handshake Authentication Method (CRAM)

Рекомендуемая литература:

http://ru.wikipedia.org/wiki/SOCKS

http://msdn.microsoft.com/ru-ru/library/system.net.sockets.tcplistener.aspx

http://msdn.microsoft.com/ru-ru/library/system.net.sockets.tcpclient.aspx

http://rfc2.ru/1928.rfc

7.3 SOCKS4 Proxy

Протокол SOCKS4 (RFC 1928) предназначен для установления прозрачного анонимного прокси-туннеля между клиентским и серверным приложениями. Основные возможности: указание адресов, которые должны быть скрыты от удаленного сервера; поддержка маршрутизации для клиентских приложений; поддержка аутентификации при использовании имени пользователя и пароля. Для реализации протокола обратитесь к своим знанием по работе с сокетами и соответствующему RFC.

7.4 HTTP Connect Proxy

Прокси-сервер HTTP CONNECT позволяет клиенту отправить HTTP-запрос к исходному серверу, такому как веб-сервер, чтобы установить соединение с удаленным хостом. Для этого клиент посылает запрос HTTP CONNECT прокси-серверу, который в свою очередь пересылает запрос исходному серверу. После установки соединения клиент может отправлять данные через прокси, словно туннель. Соответствующую команду HTTP Connect найдите или в описании RFC протокола HTTP или других ресурсах.

7.5 Задания по вариантам

1) Реализовать HTTP прокси
2) Реализовать SOCKS5 connect прокси
3) Реализовать SOCKS4 прокси
4) Реализовать SOCKS5 UDP Associate
5) Реализовать цепочку прокси (используя любой протокол)
6) Реализовать кэширующий HTTP прокси
7) Реализовать любой многопоточный прокси

8) Реализовать кэширующий HTTP прокси с использованием last_modified
9) Реализовать кэширующий HTTP прокси с использованием etag
10) Реализовать HTTP connect proxy

8. Лабораторная работа 8. Изучение json, Node.js и websocket, простейший пример парсинга.

8.1 Задание

В данной лабораторной работе необходимо создать веб-приложение, в соответствии с заданием и содержащим в себе асинхронную и синхронную части. В качестве технологии можно использовать любые технологии, языки программирования и фреймворки (PHP, Node js, ASP .Net, CGI, Fast CGI, WSGI, Fast API, Django, Flask), JSP, Spring, JavaScript, Dart, Haxe, WebAssembly), либо воспользоваться примерами и пояснениями ниже. Допускается развернуть приложение как локально, так и с использованием систем PaaS, IaaS, использовать для развертывания Docker, использовать любые веб-сервера и балансировщики нагрузки, включая Nginx. Можно воспользоваться указанными веб-серверами (Denwer - http://www.denwer.ru/, более современную Open Server - https://ospanel.io/, или XAMPP - https://www.apachefriends.org/ru/index.html, или WAMPServer http://www.wampserver.com/ru/).

Целью работы является научиться использовать технологии HTML, JavaScript на стороне клиента (фронт приложения) и научиться использовать любую выбранную технологию для создания, так называемой back-части приложения.

Варианты:

1. Вывести часы, показывающие текущее время и дату в формате чч:мм:сс, дд-мм-гггг. Добавить возможность сохранения по нажатию на кнопку, текущей отображаемой даты и времени, вывести время сервера и время клиента, сохранить имя пользователя и текущие даты и время. Использовать куки для изменения цвета и шрифта выводимого времени. Сделать отображение времени с элементами графики, например мелькание разделителей или флажков при изменении времени на 1 с. Сервер периодически получает и отображает данные и номера подключенных клиентов на каждом клиенте (использовать web socket), данные должны быть переданы и в json и xml преобразованным с использованием xslt.

2. При движении мыши по странице выводить текущие координаты указателя мыши по вертикали и горизонтали, отображать так же эти координаты в окне браузера на каком-либо элементе документа. Сделать возможным изменение какого-либо элемента при наведении на него указателя мыши. Сохранять координаты мыши на стороне сервера, вместе с введенным именем пользователя. По нажатию на кнопку формировать на стороне

сервера страницу с координатами какого-либо объекта, совпадающими с текущими координатами указателя мыши. Сервер периодически получает и отображает данные о координатах (например, при нажатии пользователем) и номера подключенных клиентов на каждом клиенте (использовать web socket), данные должны быть переданы и в json и xml преобразованным с использованием xslt.

3. Калькулятор. Создать форму, позволяющую осуществлять основные четыре арифметических действия. Должны быть поля для ввода аргументов и поле для вывода результата. Между полями аргументов должен быть выпадающий список, предоставляющий выбор одного из четырех арифметических действий. Также должно быть четвертое поле, в которое вводится предполагаемое значение результата. Если поле заполнено, то скрипт должен сверить полученный результат и вывести сообщение «Верно!» или «Неверно!» в зависимости от правильности результата. Сохранять вычисления пользователя на стороне сервера. Потом выводить их на странице в виде списка или таблицы. Сохранять в виде xml документа, при нажатии отдельной кнопки, все результаты приходят каждому клиенту. Использовать xslt и json.

4. Конвертер валют и физических величин. Создать форму позволяющую переводить километры в мили, метры в футы, килограммы в фунты и наоборот. Также должен осуществляться пересчет различных валют по курсу. Например, американские доллары в японские иены, рубли в индийские рупии и т.д. Должно быть не менее десяти различных величин или валют. Выбор величин должен сопровождаться выпадающим списком. Результаты сохранять на стороне сервера и потом выводить их в виде списка или таблицы. Таблица должна приходить каждому клиенту, если хотя бы один нажмет свою кнопку, использовать xml и xslt и json.

5. «Знаете ли Вы HTML». Создайте скрипт, задающий 5-10 вопросов о тегах и атрибутах HTML. Вопросы должны делиться на две группы: выбор из существующих вариантов (выпадающий список) и открытые вопросы, допускающие ответ в открытом виде (просто текст). Затем скрипт должен проверить правильность и выдать процент успешных ответов. Результаты пользователя должны размещаться на стороне сервера и потом доступны в виде списка или таблицы. Все результаты и выборы ответов, приходят другим пользователем. Например, пользователь1 на вопрос об HTML выбрал ответ 2 и указать вариант ответа. Использовать xslt и json.

6. Фото-галерея. Создать страницу со скриптом, который отображает в зависимости от действий пользователя (нажатие на кнопку) несколько (не более 5-10) различных рисунков попеременно в одной и той же области HTML- страницы. Если пользователь выбирает кнопку «Все на одной странице», то скрипт должен открыть новую страницу, на

которой должны быть размещены все рисунки в компактном виде. Сделать возможность сохранения рисунка на стороне сервера и затем его отображения в общем списке. Хранить названия рисунков в виде xml документа по рубрикам. При нажатии кнопки, у всех пользователей показывается информация номера выбранной картинки и сама картинка.

7. Сделать игру, в которой кружок (или изображение) двигается по случайным траекториям (какое-то время в одном направлении, потом в другом, пока не выйдет за пределы игрового поля и начинает полет с любого места внутри игрового поля, либо когда дойдет до границы, выбирает направление в соответствии с положением границы). Пользователь должен указателем мышки попасть по изображению и нажать кнопку, за что ему засчитываются баллы, результат пользователя должен быть сохранен на сервера и доступен для последующего отображения. Баллы каждого игрока при нажатии кнопки сразу приходят другим игрокам. Использовать json.

8. Сделать игру на основе таблицы или Canvas в которой растет некое дерево из квадратов, у веток дерева появляются листы и соответственно листы превращаются в ветви и так далее, пользователь должен успевать нажатием мышки на появившийся лист успевать обрезать дерево, за что ему начисляются очки, дерево «дошедшее» в своем росте до границы игрового поля «побеждает» пользователя. Только первый пользователь начинает игру. Остальные видят его результаты.

9. Игра «построй граф». Строится граф из вершин. Пользователь размещает вершины, после чего соединяет их ребрами с помощью мышки. Новые координаты приходят сразу другому клиенту, который тоже строит граф. Рисуются новые пришедшие линии графа. Граф должен быть сохранен на стороне сервера (в виде матрицы смежности). Потом сохраненный граф может быть отображен на стороне клиента.

10. Сделать свое приложение или игру, в котором используются веб-сокеты, json, анимация, xml и xslt.

11. Сделать пользовательский чат, в строке чата отображается имя пользователя, время прихода сообщения, номер сообщения.

8.2. Примеры простых приложений с использованием технологий JavaSript, Node js HTML и Javascript

Структура HTML-документа HTML — это теговый язык разметки документов, то есть любой документ на языке HTML представляет собой набор элементов, причем начало и конец каждого элемента обозначается специальными пометками, называемыми тегами. Регистр, в котором набрано имя тега, в HTML значения не имеет. Элементы могут быть пустыми, то есть не содержащими никакого текста и других данных (например, тег

перевода строки
). В этом случае обычно не указывается закрывающий тег.

Кроме того, элементы могут иметь атрибуты, определяющие какие-либо их свойства (например, размер шрифта для тега <font>). Атрибуты указываются в открывающем теге. Вот пример части разметки HTML- документа:

Текст между двумя тегами - открывающим и закрывающим. <a href="http:

//www.example.com">Здесь элемент содержит атрибут href. А вот пример пустого элемента:

 Каждый HTML-документ, отвечающий спецификации HTML какой- либо версии, обязан начинаться со строки декларации версии HTML <! DOCTYPE>, которая обычно выглядит примерно так:

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01// EN" "http:

//www.w3.org/TR/html4/strict.dtd"> Если эта строка не указана, то добиться корректного отображения документа в браузере становится труднее.

Далее обозначается начало и конец документа тегами <html> и </html> соответственно. Внутри этих тегов должны находиться теги заголовка (<head></head>) и тела (<body></body>) документа.

Основные элементы HTML – документа Теги и их параметры нечувствительны к регистру. То есть <A HREF = http:

//yahoo.com> и <a href = http:

//yahoo.com> означают одно и то же.

В последних версиях HTML практически у каждого тега огромное число необязательных параметров — обычно не меньше 15. Мы приводим только основные параметры тегов.

Гиперссылки <A HREF=url > гиперссылка </A> — гиперссылка.

Текстовые блоки <H1> … </H1>, <H2> … </H2>, … ,<H6> … </H6> — заголовки 1, 2, … 6 уровня:

<P> — новый параграф. Можно в конце параграфа поставить </P>, но это не обязательно;

 — новая строка. Этот тег не закрывается (то есть не существует тега </BR>;

<HR> — горизонтальная линия;

<BLOCKQUOTE> … </BLOCKQUOTE> — цитата.

Обычно текст сдвигается вправо;

<PRE … </PRE> — режим preview.

В этом режиме текст заключается в рамку и никак не форматируется (то есть теги, кроме </PRE>, игнорируются, и переводы строки ставятся там, и только там, где они есть

в оригинальном документе);

<DIV> … </DIV> — блок (обычно используется для применения стилей CSS);

<SPAN> … </SPAN> — строка (обычно используется для применения стилей CSS).

Форматирование текста <EM> … </EM> — логическое ударение (обычно отображается курсивным шрифтом);

… </STRONG> — усиленное логическое ударение (обычно отображается жирным шрифтом);

<I> … </I> — выделение текста курсивом;

<B> … </B> — выделение текста жирным шрифтом;

<U> … </U> — подчёркивание текста;

<S> … </S> — зачёркивание текста;

<BIG> … </BIG> — увеличение шрифта;

<SMALL> … </SMALL> — уменьшение шрифта;

<BLINK> … </BLINK> — мигающий текст. Это один из самых ненавидимых тегов, потому что мигающий текст неприятен для глаз;

<MARQUEE> … </MARQUEEK> — сдвигающийся по экрану текст. Степень народной любви к этому тэгу примерно такая же, как и к мигающему;

_… — подстрочный текст. Например, H₂O создаст текст H2O;

[…] — надстрочный текст. Например, E=mc² создаст текст E=mc2;

<FONT параметры> … </FONT> — задание параметров шрифта.

<STRONG> Списки <UL> <LI> первый элемент </LI> <LI> второй элемент </LI> <LI> третий элемент </LI> </UL> создаёт список:

первый элемент второй элемент третий элемент

Если вместо <UL> (Unordered List, что означает ненумерованный список) поставить <OL> (Ordered List, нумерованный список), список получится нумерованным:

первый элемент второй элемент третий элемент

Объекты <IMG SRC=имя или URL> — вставка изображения. Этот тег не закрывается;

<APPLET>…</APPLET> — вставка Java-апплетов;

<SCRIPT>…</SCRIPT> — вставка скриптов;

Таблицы — создание таблицы. Параметры тега:

BORDER — толщина разделительных линий в таблице;

CELLSPACING — расстояние между клетками;

CAPTION — заголовок таблицы (этот тег необязателен);

TR — строка таблицы;

TH — заголовок столбца таблицы (этот тег необязателен);

TD — ячейка таблицы;

height - высота таблицы;

<TABLE>…</TABLE> Так, например, код:

<TABLE BORDER="1" CELLSPACING="0">

<CAPTION> Улов крокодилов </CAPTION>

<TH> Год </TH>

<TH> Улов </TH>

<TR> <TD> 1997 </TD> <TD> 214 </TD> </TR>

<TR> <TD> 1998 </TD> <TD> 216 </TD> </TR>

<TR> <TD> 1999 </TD> <TD> 207 </TD> </TR>

</TABLE>

<FORM> — создание формы

<INPUT> — элемент ввода (может иметь разные функции — от ввода текста до отправки формы)

<TEXTAREA> — текстовая область (многострочное поле для ввода текста)

<SELECT> — список (обычно в виде выпадающего меню)

<OPTION> — пункт списка JavaScript JavaScript — интерпретируемый язык программирования, являющийся одной из реализаций языка ECMAScript и основанный на концепции прототипов, пришедшей из языка Self.

В настоящее время JavaScript используется в основном для создания сценариев поведения браузера, встраиваемых в Web-страницы, но также находит применение в качестве скриптового языка доступа к объектам приложений. Некоторые приложения имеют встроенный интерпретатор JavaScript, позволяющий расширять их возможности без изменения самого приложения.

JavaScript обладает рядом свойств объектно-ориентированного языка, но благодаря прототипированию поддержка объектов в нём отличается от традиционных объектно-ориентированных языков. Так же, JavaScript имеет ряд свойств, присущих функциональным языкам — функции как объекты первого уровня, объекты как списки, карринг (currying), анонимные функции, замыкания (closures) — что придаёт языку дополнительную гибкость.

Название JavaScript является зарегистрированной торговой маркой компании Sun Microsystems, Inc.

О языке JavaScript Синтаксис JavaScript хотя и похож на язык Си, концептуально имеет коренные отличия. Основными чертами JavaScript являются:

функции как объекты первого уровня обработка исключений автоматическое приведение типов автоматическая сборка мусора анонимные функции Изначально JavaScript был разработан как скриптовый язык для расширения возможностей существующих приложений. В первую очередь он стал хорошо известным и популярным благодаря использованию в браузерах.

При использовании в рамках технологии DHTML, код JavaScript включается в HTML-код страницы и исполняется интерпретатором, встроенным в браузер. JavaScript заключается в теги <script></script>, по спецификации HTML 4.01 у тега <script> обязателен атрибут type="text/javascript", хотя в большинстве браузеров язык сценариев по умолчанию именно JavaScript.

При этом атрибут language (language="JavaScript"), несмотря на его активное использование, не входит в стандарт и поэтому считается некорректным.

Примеры программ на JavaScript Пример объявления и использования класса в JavaScript (класс является одновременно функцией, так как фунции - это объекты первого уровня):

```
function MyClass() { this.myValue1 = 1;
this.myValue2 = 2;
}
var mc = new MyClass();
mc.myValue1 = mc.myValue2*2;
```

Скрипт, выводящий модальное окно с классической надписью «Hello, World!» внутри браузера:

```
<script type="text/javascript"> alert('Hello, World!');
</script>
```

Следуя концепции интеграции JavaScript в существующие системы, браузеры поддерживают включение скрипта, например, в значение атрибута события:

Удалить Здесь при нажатии на ссылку функция alert('Вы уверены?');

вызвает модальное окно с надписью «Вы уверены?», a return false;

блокирует переход по ссылке.

Разумеется, этот код будет работать только если в браузере есть и включена поддержка JavaScript, иначе переход по ссылке произойдет без предупреждения.

Есть и третья возможность подключения JavaScript — написать скрипт в отдельном

файле, а потом подключить его с помощью конструкции <script type="text/javascript" src="http://Путь_до_файла_со_скриптом"></script>

8.3 Технология PHP

Ниже приведен простой пример программирования на PHP:

<html>

<head>

<title>Пример</title> </head>

<body> <?php echo "Привет, я - скрипт PHP!";?> </body>

</html>

Обратите внимание на отличие этого скрипта от скриптов, написанных на других языках, например, на Perl или C - вместо того, чтобы создавать программу, которая занимается формированием HTML-кода и содержит бесчисленное множество предназначенных для этого команд, вы создаете HTML-код с несколькими внедренными командами PHP (в приведенном случае, предназначенными для вывода текста). Код PHP отделяется специальными начальным и конечным тегами, которые позволяют процессору PHP определять начало и конец участка HTML-кода, содержащего PHP-скрипт.

Существует четыре набора тегов, которые могут быть использованы для обозначения PHP-кода. Из них только два (<?php. . .?> и <script language = "php">…… </script>) всегда доступны;

другие могут быть включены или выключены в конфигурационном файле php.ini.

Теги, поддерживаемые PHP:

1. <?php echo("если вы хотите работать с документами XHTML делайте так\n");?>

2. <? echo ("это простейшая инструкция обработки SGML\n");?>

<?= выражение ?>

Это синоним для "<? echo выражение ?>" или XML, language="php"> редакторы (например, FronPage) обработки");

</script>

3. <script echo ("некоторые не любят инструкции

4. <% echo ("Вы можете по выбору использовать теги в стиле ASP");%>

<%= $variable;# Это синоним для "<% echo . . ." %>

Первый способ, <?php. . .?>, наиболее предпочтительный, так как он позволяет использовать PHP в коде, соответствующем правилам XML, таком как XHTML.

Работа с формами

Одно из главнейших достоинств PHP - то, как он работает с формами HTML. Здесь

основным является то, что каждый элемент формы автоматически становится доступен вашим программам на PHP. Для подробной информации об использовании форм в PHP читайте раздел " Переменные из внешних источников" [Руководства по PHP].

Ниже приведен пример формы HTML:

```
<form action="action.php" method="POST">
Ваше имя:
<input type="text" name="name" />
Ваш возраст:
<input type="text" name="age" />
Ваш пол:
<input type = “radio” name = “gender” CHEKED VALUE = “1”>Мужчина <br>
<input type = “radio” name = “gender” VALUE = “2”>Женщина <input type="submit">
 </form>
```

В этой форме нет ничего особенного. Это обычная форма HTML без каких-либо специальных тегов. Когда пользователь заполнит форму и нажмет кнопку отправки, будет вызвана страница action.php. В этом файле может быть что-то вроде:

```
Здравствуйте,
<?php echo $_POST["name"];?>.
<br> Вам
<?php echo $_POST["age"];?> лет. <br> Вы <?php if ($_POST["gender"] = 1) {?> Мужчина. <?php }
else {?> Женщина. <?php } ?>
```

Пример вывода данной программы:

Здравствуйте, Владимир.

Вам 30 лет.

Вы мужчина.

работа данного кода проста и понятна. Переменные $_POST["name"] и $_POST["age"] автоматически установлены для вас средствами PHP. В переменной $_POST["gender"] находится значение VALUE, в зависимости от выбранного переключателя (аналогично можно передавать значения элементов флажков, но при этом значения их атрибута «name» должны отличаться). Заметим, что метод отправки нашей формы - POST. Если бы мы использовали метод GET, то информация нашей формы была бы в суперглобальной переменной $_GET. Также можно использовать переменную $_REQUEST, если источник данных не имеет значения. Эта переменная содержит смесь

данных GET, POST, COOKIE и FILE.

Ниже вам предлагается ознакомиться с кодом простейшего приложения, некоторые строки которого будут представлены с пояснением.

Содержание файла index.html

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta charset="UTF-8">
<title>Lab5</title>
<link rel="stylesheet" type="text/css" href="style.css">// подключаем css файл со стилями
<script src="//ajax.googleapis.com/ajax/libs/jquery/3.1.0/jquery.min.js"></script>// подключаем библиотеку jquery
</head>
<body>
<div class="question_box">
 <form>
  <label>Выберите исходную валюту:</label>
<select name="from" id="from">
<option value="0" selected="selected">Доллар США</option>
<option value="1">Евро</option>
<option value="2">Рубль</option>
</select>
<label>Введите сумму: </label>
<input name="cash" id="cash" class="is" type="text"/>//создаем поля ввода для конвертируемой суммы
  <br/> <br/>
  <label>Выберите расчетную валюту: </label>
 <select name="to" id="to" >
<option value="0">Доллар США</option>
<option value="1" selected="selected">Евро</option>
<option value="2">Рубль</option>
 </select>
<input class="button" type=button value="Вычислить" onclick='GiveResult();'>
```

```
<div class="res">//поля для вывода результата вычисления
  <label><span id="result" class="result"></span></label>
 </div>
 </form>
<img src="money.svg" class="bgimg">
 </div>
      <script type="text/javascript">//указываем, что далее будет идти js-скрипт
        c = new Array(); //массив констант
        n = new Array(); //массив для хранения обозначений валюты
        c[0] = 1;
        n[0] = "USD";
        c[1] = 0.844;
        n[1] = "EUR";
        c[2] = 58.01;
        n[2] = "RUB";

        function GiveResult() //функция вычисления результата
        {
          var res, vfrom, vto, vcash; //объявление переменных результата, из какой валюты конвертируем, в какую конвертируем, введенная пользователем сумма соответственно
    vcash = document.getElementById("cash").value; //функ-я позволяет получить элемент по идентификатору, далее аналогично
          vfrom = document.getElementById("from").value;
          vcash = vcash.replace(',', '\.'); //
      vcash = vcash.replace(' ', '');  // замена символов, необходимо для осуществления вычислений
          vcash = vcash.replace(' ', '');//
          vto = document.getElementById("to").value; //получаем символьное обозначение валюты, в которую будем конвертировать
          res = c[vto] * vcash / c[vfrom]; //осуществляем вычисление результата

          res = res.toFixed(3); //указываем сколько знаков после запятой будет указано в результате вычислений
          res = res.toString(); //переводим результат в строковое представление
```

res = res.replace('\.', ','); //заменяем точку, необходимую для производства вычислений, обратно на запятую

// Ниже помощью метода post библиотеки jquery, отправляем значение вычисленного результата в форму main.php, который осуществляет запись его в txt - файл

$.post("main.php",{res:res});

res = " Результат перевода: " + res + " " + n[vto];

document.getElementById("result").innerHTML = res; //вывод результата на форму в html

}

</script>

</body>

</html>

Опустим содержание файла style.css в виду того, что там представлены только стили, применяемые к элементам разметки html и данная информация не несет за собой никакой интеллектуальной нагрузки.

Содержание файла main.php

<?php

$ress=$_POST['res']; // создаём переменную и сразу же инициализируем её с помощью метода _POST, параметром которого является атрибут, отправленной переменной из html-файла

$fp = fopen('textfile.txt', 'a+'); //Открыем файл только для записи, причем указатель помещается в конец файла. Если файл не существует - пытаемся его создать.

$file = fwrite($fp, $ress); // запишем в открытый файл переменную, переданную ajax-запросом в html-файле

$file = fwrite($fp, "\n"); // добавим перенос строки

fclose($fp); //закроем файл

?>

Еще пример одного приложения.
Листинг приложения rcr-example:
Файл Index.php:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"> //Кодировка
    <title>Best game ever</title>
  </head>
  <body>
<form style="margin-left: 35%; margin-top: 1%;"  action="game.php" method="POST">
//Вызов страницы game.php с передачей данных методом  POST
Ваше имя:
<input type="text" name="name" /> // Поле для ввода имени
<input type="submit"> //Кнопка отправки формы.
</form>
  </body>
</html>
```

Файл game.php:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"> //Кодировка
    <title>Best game ever</title>
    <link rel="stylesheet" href="styles/style.css"> //Подключения файла стилей

  </head>
  <body>

    <p style="margin-left: 38%; margin-top: 1%"> Добро пожаловать, <? echo $_POST["name"]; ?>!</p> //Приветствие игрока. Использование переданного имени в PHP коде. Отступ слева – 38% экрана, сверху – 1%.

    <button id="start-button" onclick="gameStart()">Начать игру!</button>
// Кнопка начало игры. Нажатие вызывает js-функцию gameStart().
    <div id="score">Счет: 0</div> // Отображение счета
        <div id="timer" > Времени прошло: 0:00</div> // Таймер
    <div id="square" onclick="move() "> </div> // Круг, который нужно нажимать, нажатие вызывает функцию move().
    <div id="size"> </div> //Границы поля

  <script type="text/javascript" src="scripts/myscript.js"></script> // Вызов javascript
  </body>
</html>
```

Файл myscript.js:

```
    var on;
    var sumScore = 0; // Переменная для изменения счета игрока.
    var square = document.getElementById('square'); //Красный круг
    var score = document.getElementById('score'); //Счет
    var time = document.getElementById('timer'); //Таймер
```

```
        function move() //Смещение красного круга
            {

            var fromLeft = Math.random() * (58 - 32) + 32; //Случайные координаты красного круга с левой стороны от 32 до 58 % экрана.
            var fromTop = Math.random() * (540 - 100) + 100; //Случайные координаты красного круга сверху от 100 до 540 пикселей.
            var click = document.getElementById('square'); //Красный круг
            click.style.left = fromLeft + '%'; //Изменение положения круга на экране
            click.style.top = fromTop + 'px'; // Изменение положения круга на экране
            sumScore++; //Увеличение счета
            score.innerHTML = 'Счет: ' + sumScore; //Визуальное изменение счета на экране
              }

          function gameStart() { //Начало игры
            sumScore = 0; //Обнуление счета
            score.innerHTML = 'Счет: ' + sumScore;
            square.style.width = '40px'; //Создание красного круга, изменение ширины с 0 до 40 пикселей
            square.style.height = '40px'; //Создание красного круга, изменение длинны с 0 до 40 пикселей
            time.innerHTML = 'Времени прошло: 0:00'; //Сброс времени
            clearInterval(on); //Запуск интервала.
            timerStart(); //Запуск таймера
          }
          function gameEnd() { //Конец игры
            square.style.width = '0px'; //Удаление красного круга изменением длины до 0 пикселей
            square.style.height = '0px'; //Удаление красного круга изменением ширины до 0 пикселей
            alert('Ваш счет: ' + sumScore); //alert-сообщение с итоговым счетов
            score.innerHTML = 'Счет: ' + sumScore;
            }
          function timerStart() { //Функция запуска таймера.

            var second = 0; //Сброс таймера
            var minute = 0; //Сброс таймера

            on = setInterval(function () { //Интервал. Действия выполняются раз в секунду
              second++; //Увеличение поля секунд таймера
              if (second == 60) {minute++; second = 0;} //Сброс секунд, прибавление минут.
              if (second <= 9) time.innerHTML = 'Времени прошло: ' + minute + ':0'+second; //Шаблон отображения таймера, если прошло меньше десяти секунд
              else time.innerHTML = 'Времени прошло: ' + minute + ':' + second;
      // Шаблон отображения таймера, если прошло больше десяти секунд
              if (minute == 1) {clearInterval(on); gameEnd();} //Условие прекращения игры, остановка таймера, вызов функции gameEnd()
            }, 1000);

          }
```

Файл style.css:

```
#square { //Красный круг
    width: 0px; //Ширина
    height: 0px; //Высота
    background-color: red; //Цвет
    z-index: 2; //Расположение слоя
    position: absolute; //Расположение
    left: 33%; //Отступ слева
    top: 100px; //Отступ сверху
    border-radius: 50%; //Приведение к круглому виду
  }

#size { //Поле
    width: 543px; //Ширина
    height: 500px; //Высота
    border: 3px solid black; //Цвет
    position: absolute; //Позиция
    left: 32%; //Отступ слева
    top: 90px; //Отступ сверху
    z-index: 1; //Расположение слоя
}

#timer { //Таймер
    position: absolute; //Позиция
    left: 38%; //Отступ слева
    top: 53px; //Отступ сверху
    border: 2px solid rebeccapurple; //Цвет рамки
    border-radius: 50px; //Закругление краев
    width: 160px; //Ширина
    height: 20px; //Высота
    text-align: left; //Расположение текста слева
    padding-left: 5px; //Отступ слева от границы элемента
}
#start-button{ //Кнопка начала игры
    position: absolute; //Расположение
    top: 55px;
    left: 32%;
}

#score { //Счет
    position: absolute;
    left: 48%;
    top: 53px;
    border: 2px solid rebeccapurple; //Цвет рамки
    border-radius: 50px; //Закругление краев
    width: 70px;
    height: 20px;
    text-align: left;
    padding-left: 5px;
}
```

Коды сайта: Онлайн-Чат

index.php

```
<?php
   if(isset($_POST['message'])){
      $connect = mysqli_connect('localhost', 'sit5labuser', 'sit5labpassword', 'sit_db');
      $name = $_POST['name'];
      $message = $_POST['message'];
      mysqli_query($connect, "INSERT INTO `messages` (`id`, `name`, `message`) VALUES (NULL, '$name', '$message')");
      }

?><!DOCTYPE html>
<html>
   <head>
      <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
      <title>Онлайн-Чат для СиТ!</title>
      <style>
         body{
               background-color: green;
            }
         iframe{
            background-color: lightyellow;
            margin-bottom: 20px;
            border-radius: 40px;
            border: 5px groove green;
            padding: 10px;
         }
            h1,h3,h6{
               color: white;
            }
      </style>
   </head>

   <body>
      <center>
```

```
            <h1>Онлайн-Чат для СиТ!</h1>
              <table width="1000px"><tr><td width="800px" height="500px" valign="top">
            <iframe name="get_i_name" id="get_i" width="800px" height="500px"
src="messages.php"></iframe>
              </td><td width="200px">
            <form id="set_f" action="/" method="post">
              <h3>Имя (никнейм): </h3>
              <input name="name" type="text" size="30" maxlength="30"
placeholder="Имя..." value="<?php echo $_POST['name']; ?>">
              <h3>Сообщение: </h3>
              <textarea name="message" form="set_f" cols="50" rows="4"
placeholder="Сообщение..."></textarea>
              <br/>
              <button onclick="send();">Отправить!</button>
            </form>
              </td></tr></table>

            <script>
              setInterval(function () {
                document.getElementById('get_i').src =
document.getElementById('get_i').src;
              }, 3000);
              function send() {
                document.getElementById('set_f').submit();
              }
            </script>
          </center>
        </body>
      </html>
```

messages.php

```
<?php
$connect = mysqli_connect('localhost', 'sit5labuser', 'sit5labpassword', 'sit_db');
$result = mysqli_query($connect, "SELECT * FROM messages ORDER BY id DESC
LIMIT 12");
```

```
while($res = mysqli_fetch_assoc($result)){
    echo '<strong>'.$res['name'].'</strong>: <em>'.$res['message'].'</em><br/><hr/>';
}
```

8.4. Примеры работы с web-сокетами и XML (node js, python Tornado).

8.4.1 Пример работы с HTTP сервером Node.js.

Для установки node.js в Linux (Debian, Ubuntu, Mint) достаточно вызвать команду sudo apt-get install nodejs npm.

Некоторые модули можно установить используя Npm, либо непосредственно в установщике выбрать модуль node, например, node-websocket.

Попробуем создать простейший сервер (файл server1.js).

```
//вывод версии node.js
console.log(process.version);
//загрузка модуля http
var http = require('http');
//создаем сервер http с прослушиванием на порту 10555
//req – request  - запрос
//res – response - ответ
http.createServer(function (req, res) {
//формируем заголовки ответа и код ответа
   res.writeHead(200, {'Content-Type': 'text/plain'});
//формируем тело ответа
   res.end('Hello World\n');
}).listen(10555, '127.0.0.1');
console.log('Server running at http://127.0.0.1:10555/');
```

Запустим наш сервер:

```
node server1.js
```

Загрузим в Браузере:

http://127.0.0.1:10555/

8.4.2 Пример работы с Json и http сервером Node.js.

Создадим еще один простой пример, в котором JSON структура преобразуется в

текст HTML, который передается клиенту.

```
console.log(process.version);
var http = require('http');
var fs = require('fs');
//пример синхронного чтения файла для передачи пользователю
var contents = fs.readFileSync('index.html', 'utf8');
//получение текущей даты и вывода его на экран
dt = new Date()
dt.toLocaleDateString("ru-RU")
console.log(dt.toString());
//пример   структуры JSON
var mJSON = { "name": "Server", "type": "simple" , "time" : dt.toString(),
"val" : {"t" :"t1","struct": {"par1":"value"}, "t2":"t3" },
"mas":["1","2", {"val1":{"1":"2","3":"4"},"val2":"2"},"3"]};
//преобразование структуры в строку
var mString = JSON.stringify(mJSON);
console.log(mString);
//функция для построения документа со вложенными списками по JSON
function gethtmltree(mJSON)
{
S= ""
function gettree(mJSON1,i)
{
S = S+"<ul>\n";
for(key in mJSON1)
        {
        if(typeof(mJSON1[key])==="object")
        {
        S = S+"<li> "+key +"\n";
        gettree(mJSON1[key],i+1);
        S = S+"</li>\n";
        }   else {
        S=S+"<li>"+key+": "+mJSON1[key]+"</li>\n";
        }
```

```
        }
S = S+"</ul>\n";
};
gettree(mJSON,1);
S1 = "<html><title> file </title> <body> \n";
S2 = " </body> </html> ";
return S1+S+S2;
}
//получаем текст страницы
S = gethtmltree(mJSON);

console.log(S);
http.createServer(function (req, res) {
   res.writeHead(200, {
      'Content-Type': 'text/html'
   });
   res.write(S)
   res.end();
}).listen(10555, '127.0.0.1');

console.log('Server running at http://127.0.0.1:10555/');
```

Приведенный пример очень прост, но позволяет продемонстрировать динамическое формирование страницы на базе JSON данных. Можно модифицировать пример, когда получая JSON сообщение формировать на его базе визуальное отображение на странице. Здесь не принципиально, что содержится в Index.html, так как здесь этот файл вообще не используется, но вы можете поэкспериментировать и отобразить contents, а не строку S. Можете для эксперимента модифицировать функцию gethtmltree с использованием выпадающих списков. Также попробуйте использовать jQuery.

8.4.3 Пример работы с websocket на Node.js.

Отличие технологии Websocket от HTTP заключается в возможности использования асинхронных запросов, как со стороны клиента, так и со стороны сервера. Использование HTTP предполагает запрос от клиента и ответ, для реализации интерактивного приложения взаимодействующего с сервером необходимо постоянно обращаться к серверу

с запросами, чтобы узнать его состояние. Далее рассмотрим простейший пример, в котором, реализуется идея асинхронной отправки данных от сервера к клиенту и обратно, по событию от таймера на стороне сервера. Таким образом, сервер сам инициирует событие, на которое реагирует клиент. В том, числе веб-соккеты позволяют инициировать обработку событий подключения клиента, отключения, отправки сообщения. Для работы с websocket необходимо установить также компонент Node-websocket. В операционной системе линукс данный компонент можно найти в репозитарии. Наш проект будет состоять из файла с кодом сервера, файла с кодом html страницы и скриптом создающим клиентское веб-соккет соединение. В данной реализации присутствуют ошибки, которые позволяют понять принцип использования веб-соккетов в дальнейшем при сравнении с правильным кодом.

Файл webserver1.js.

```
console.log("WebSocket Server ")
//библиотека работы с файлами
var fs = require('fs');
//синхронно читаем клиентский html файл
var contents = fs.readFileSync('webclient.html', 'utf8');
//библиотека работы с вебсоккетом сервера
var WebSocketServer = require('websocket').server;
//библиотека http
var http = require('http');
//создаем http сервер
var server = http.createServer(function(request, response) {
//отдаем считанный документ клиенту (браузеру)
  response.writeHead(200, {'Content-Type': 'text/html' });
  response.write(contents)
  response.end()
});
//прослушивание сокета сервера
server.listen(10556, function() { });
// create the веб сокет server ассоциированный с http сервером
wsServer = new WebSocketServer({
  httpServer: server
});
//так делать неправильно, так как индекс глобальная переменная и будет изменяться
//одновременно обработчикам двух клиентов, далее будет показан правильный код
```

```
var fl = true;
var index = 0;
// WebSocket server - обработка события запрос соединения
wsServer.on('request', function(request) {
  var connection = request.accept(null, request.origin);
  fl = true;

  // обработчик собатия от таймера
  function timecounter(arg) {
  console.log('arg was => %s', arg);
  //отправляем json строку клиенту
//так делать неправильно так как, в случае отключения будет попытка отправки на уже
//закрытый сокет, необходимо сначала сделать проверку флага
  connection.sendUTF(
  JSON.stringify({ 'type': 'chat', 'data': String(index)} ));
  index=index+1;
//так делать неправильно так как, в случае отключения будет попытка отправки на уже
//закрытый сокет
  if(fl)
  setTimeout(timecounter, 3000, index);
  }
  setTimeout(timecounter, 3000, 'start');

  // This is the most important callback for us, we'll handle
  // all messages from users here.
  connection.on('message', function(message) {

    // process WebSocket message  , если это текст
    if(message.type === 'utf8') {
    console.log("client send json: "+message.utf8Data);
    msg = JSON.parse(message.utf8Data);
    console.log("client send data: "+msg.data);
    }
  });

  connection.on('close', function(connection) {
   // close user connection
    fl = false;
    console.log("User close connection")
  });
```

```
});
```

Файл webclient.html.

```
<!DOCTYPE html>
<html>
<head>
<!--//задаем кодировку документа -->
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
 <title>Пример работы с веб соккетом</title>
</head>
<body>
 Hello World!!! Iam webSocket
 <div id="elem1"></div>
 <script>
  //создаем клиентский веб соккет
  let ws = new WebSocket("ws://localhost:10556");
  // обработка асинхронного события появления сообщения от сервера
  ws.onmessage = ({data}) => {
      //парсим строку посланную вебсокетом сервера из таймера
      jsmsg = JSON.parse(data)
      //выводим счетчик в элемент div (поле data)
      document.getElementById('elem1').innerHTML = jsmsg.data;
  }
  // отправляем сообщение при открытии документа
  msg = JSON.stringify({data: "hello"} );
  ws.onopen = () => ws.send(msg);
 </script>
</body>
</html>
```

Очевидно, что данный пример обладает недостатками, один из них связан с тем, что в сервере используются глобальные переменные, и если, например, открыть подключение еще из одного окна браузера, то данный веб-браузер будет реагировать

неправильно, на клиентах будет отображаться один и тот же счетчик и соответственно при этом происходит его «ускорение». Так же при закрытии клиента по таймеру все равно будет сделана попытка отправить ему сообщение, и произойдет «зависание». Более правильное решение для сервера приводится снизу. Клиент достаточно прост, он отправляет сообщение в формате JSON, а также отображает сообщение о номере тика от сервера.

Файл webserver1.js.

```
console.log("WebSocket Server ")
//библиотека работы с файлами
var fs = require('fs');
//синхронно читаем клиентский html файл
var contents = fs.readFileSync('webclient.html', 'utf8');
//библиотека работы с вебсоккетом сервера
var WebSocketServer = require('websocket').server;
//библиотека http
var http = require('http');
//создаем http сервер
var server = http.createServer(function(request, response) {
//отдаем считанный документ клиенту (браузеру)
  response.writeHead(200, {'Content-Type': 'text/html' });
  response.write(contents)
  response.end()
});
//прослушивание соккета сервера
server.listen(10556, function() { });

// create the веб соккет server ассоцированный с http сервером
wsServer = new WebSocketServer({
  httpServer: server
});
// WebSocket server - обработка события запрос соединения
wsServer.on('request', function(request) {
//переменные для каждого клиента свои
```

```
var fl = true;
var index = 0;
var connection = request.accept(null, request.origin);
console.log(connection.toString());
// обработчик собатия от таймера
function timecounter(arg) {
console.log('arg was => %s', arg);
console.log(fl);
//проверка закрыт ли соккет до отправки ему сообщения
if(fl) {
//отправляем json строку клиенту
connection.sendUTF(
JSON.stringify({ 'type': 'chat', 'data': String(index)} ));
index=index+1;
setTimeout(timecounter, 3000, index);
}
}
setTimeout(timecounter, 3000, 'start');

// This is the most important callback for us, we'll handle
// all messages from users here.
connection.on('message', function(message) {

   // process WebSocket message  , если это текст
   if(message.type === 'utf8') {
   console.log("client send json: "+message.utf8Data);
   msg = JSON.parse(message.utf8Data);
   console.log("client send data: "+msg.data);
   }
});
connection.on('close', function(connection) {
  // close user connection
   fl = false;
   console.log(connection.toString());
   console.log("User close connection")
```

```
  });
});
```

8.4.4 Реализация обработки XML, использование XSLT и websocket на Python.

Для работы с websocket на Python будем использовать python 3 и tornado. В linux на базе debian (mint, ubuntu) можно использовать менеджер программ, пакет python-tornado, либо установить Pip3 и далее pip3 install tornado. Вместо tornado можете использовать любой другой веб-сервер.

Сделаем для начала такой же сервер обработчик, что мы делали на node.js.

Файл: webserv.py.

```
import threading
import tornado.websocket
import tornado.web
import tornado.ioloop

#обработчик запросов к http серверу
class MainHandler(tornado.web.RequestHandler):
        #обработка запроса get
        def get(self):
                print('Get')
                #передаем нашу страницу в которой реализована работа с клиентом
                self.render('webclient.html')
#обработчик событий  вебсоккетов
class EchoWebSocket(tornado.websocket.WebSocketHandler):
        clients = []#массив клиентов
        fl = True
        index = 0
        #процедура отправки клиенту текущего индекса через 3 сек
        def go(self,client):
                print('ok')
                if(self.fl):
```

```
                    self.index=self.index+1
                    s = u'{"type": "chat", "data": "' +str(self.index) +'"}'
                    print("send message : "+s)
                    #посылаем сообщение клиенту
                    client.write_message(s)
                    #запускаем таймер который будет вызывать функцию go с
аргументом client каждые 3 сек
                    t = threading.Timer(3.0, self.go,[client])
                    t.start()


#проверяются и даются права на действия с сокетом, здесь права даются всем
        def check_origin(self, origin):
                return True

        #обработка события открытия соединения
        def open(self):
                print("Client open")
                #добавляем клиента в список
                self.clients.append(self)
                self.fl = True
                #запускаем поток отправки сообщение клиенту
                self.go(self)
        #обработка прихода события от сервера
        def on_message(self, message):
                        print("Client message "+message)
        #обработка события закрытия сокета клиента
        def on_close(self):
                self.fl = False
                #удаляем клинта из списка
                self.clients.remove(self)
                print("WebSocket closed")

#создаем приложение tornado с обработчиком вебсокетов и http сервером
app = tornado.web.Application([(r"/ws", EchoWebSocket),(r'/', MainHandler)])
```

```
#прослушиваем на порту
app.listen(10556)
print("Start Server")
#запускаем цикл прослушивания и обработки событий
tornado.ioloop.IOLoop.instance().start()
```

Файл запускается в среде разработки, если запускается из командной строки, то желательно ввести функцию main.

В файле клиента была заменена одна строчка, чтобы отделить доступ http от доступа к вебсоккету:

```
//создаем клиентский веб-сокет
    let ws = new WebSocket("ws://localhost:10556/ws");
```

добавлено ws.

8.4.5 Возможности динамического использования XML и XSLT для формирования документа.

Введение в XML

XML (Extensible Markup Language — расширяемый язык разметки) — рекомендованный Консорциумом Всемирной паутины язык разметки, фактически представляющий собой свод общих синтаксических правил.

XML предназначен для хранения структурированных данных для обмена информацией между программами, а также для создания на его основе более специализированных языков разметки (например, XHTML), иногда называемых словарями. XML является упрощённым подмножеством языка SGML.

Целью создания XML было обеспечение совместимости при передаче структурированных данных между разными системами обработки информации, особенно при передаче таких данных через Интернет.

Словари, основанные на XML (например, RDF, RSS, MathML, XHTML, SVG), сами по себе формально описаны, что позволяет программно изменять и проверять документы на основе этих словарей, не зная их семантики, то есть не зная смыслового значения элементов. Важной особенностью XML также является применение так называемых пространств имён (англ. namespace).

Краткий обзор синтаксиса XML Ниже приведён пример простого кулинарного рецепта, размеченного с помощью XML:

<?xml version="1.0" encoding="UTF-8"?>

<Recipe name="хлеб" prep_time="5 мин" cook_time="3 час">

<title>Простой хлеб</title>

<ingredient amount="3" unit = "стакан"> Мука </ingredient>

<ingredient amount="0.25" unit = "грамм"> Дрожжи </ingredient>

<ingredient amount = "1.5" unit = "стакан"> Тёплая вода </ingredient>

<ingredient amount="1" unit="чайная ложка"> Соль</ingredient>

<Instructions> <step>Смешать все ингредиенты и тщательно замесить.</step>

<step>Закрыть тканью и оставить на один час в тёплом помещении.</step>

<step>Замесить ещё раз, положить на противень и поставить в духовку.</step>

</Instructions>

</Recipe>

Обратите внимание, что названия и значения элементов и атрибутов могут состоять не только из букв латинского алфавита, но десятичным разделителем может быть только точка.

Первая строка XML-документа называется объявлением XML (англ. XML declaration) — это необязательная строка, указывающая версию стандарта XML (обычно это 1.0), также здесь может быть указана кодировка символов и внешние зависимости.

Остальная часть этого XML-документа состоит из вложенных элементов, некоторые из которых имеют атрибуты и содержимое.

Элемент обычно состоит из открывающего и закрывающего тегов (меток), обрамляющих текст и другие элементы. Содержимым элемента (англ. content) называется всё, что расположено между открывающим и закрывающим тегами, включая текст и другие (вложенные) элементы.

Ниже приведён пример XML-элемента, который содержит открывающий тег, закрывающий тег и содержимое элемента: <step>Замесить ещё раз, положить на противень и поставить в духовку.</step>

Кроме содержания у элемента могут быть атрибуты — пары имя- значение, добавляемые в открывающий тег после названия элемента.

Значения атрибутов всегда заключаются в кавычки (одинарные или двойные), одно и то же имя атрибута не может встречаться дважды в одном элементе. Не рекомендуется использовать разные типы кавычек для значений атрибутов одного тега.

<ingredient amount="3" unit = "стакан">Мука </ingredient>

В приведённом примере у элемента «ingredient» есть два атрибута:

«amount», имеющий значение «3», и «unit», имеющий значение «стакан». С точки зрения XML-разметки, приведённые атрибуты не несут никакого смысла, а являются просто набором символов.

Кроме текста элемент может содержать другие элементы:

<Instructions> <step>Смешать все ингредиенты и тщательно замесить.</step> <step>Закрыть тканью и оставить на один час в тёплом помещении.</step> <step>Замесить ещё раз, положить на противень и поставить в духовку.</step> </Instructions> В данном случае элемент «Instructions» содержит три элемента «step».

XML не допускает перекрывающихся элементов. Например, приведённый ниже фрагмент некорректен, так как элементы «em» и «strong» перекрываются.

<!-- ВНИМАНИЕ! Некорректный XML! -->

Обычный акцентированный

выделенный и акцентированный выделенный

Каждый XML-документ должен содержать в точности один корневой элемент (англ. root element или document element), таким образом, следующий фрагмент не может считаться корректным XML-документом.

<!-- ВНИМАНИЕ! Некорректный XML! -->

<thing>Сущность №1</thing>

<thing>Сущность №2</thing>

Для обозначения элемента без содержания, называемого пустым элементом, допускается применять особую форму записи, состоящую из одного тега, в котором после имени элемента ставится косая черта.

Следующие фрагменты полностью равнозначны:

<foo></foo> <foo/>

В XML определены два метода записи специальных символов:

ссылка на сущность и ссылка по номеру символа. Сущностью (англ. entity) в XML называются именованные данные, обычно текстовые, в частности спецсимволы. Ссылка на сущность (англ. entity references) указывается в том месте, где должна быть сущность и состоит из амперсанда («&»), имени сущности и точки с запятой («;»).

В XML есть несколько предопределённых сущностей, таких как «lt» (ссылаться на неё можно написав «<») для левой угловой скобки и «amp» (ссылка — «&») для амперсанда, возможно также определять собственные сущности. Помимо записи с помощью сущностей отдельных символов, их можно использовать для записи часто

встречающихся текстовых блоков. Ниже приведён пример использования предопределённой сущности для избегания использования знака амперсанда в названии:

<company-name>AT& T</company-name>

Полный список предопределённых сущностей состоит из &

(«&»), <

(«<»), >

(«>»), '

(«'»), и "

(«"») — последние две полезны для записи разделителей внутри значений атрибутов. Определить свои сущности можно в DTD-документе. Иногда бывает необходимо определить неразрывный пробел, который в HTML обозначается как

в XML его записывают

Ссылка по номеру символа (англ. numeric character reference) выглядит как ссылка на сущность, но вместо имени сущности указывается символ # и число (в десятичной или шестнадцатеричной записи), являющееся номером символа в кодовой таблице Юникод. Амперсанд может быть представлен следующим образом:

<company-name>AT&

T</company-name>

Существует ещё множество правил, касающихся составления корректного XML-документа, но целью данного краткого обзора было лишь показать основы, необходимые для понимания структуры XML- документа.

Введение в XSLT.

Для изучения можно обратиться к

http://citforum.ru/internet/xslt/xslt.shtml.

https://htmlweb.ru/xml/xslt1.php.

XSLT (eXtensible Stylesheet Language Transformations) - расширяемый язык преобразования листов стилей.

Язык XSLT служит транслятором, с помощью которого можно свободно модифицировать исходный текст. XLST играет решающую роль в утверждении XML в качестве универсального языка хранения и передачи данных. Область применения XSLT широка - от электронной коммерции до беспроводного Web.

Фактическая сборка результирующего документа происходит, когда исходный документ и лист стилей XSLT передаются в синтаксический анализатор XSLT (XSLT-процессор).

При использовании XSLT в среде Web синтаксический анализ может происходить либо на стороне пользователя (т.е. в пользовательском браузере), либо на стороне сервера.

Анализ XSLT на стороне клиента похож на процедуру применения каскадных листов стилей. В исходный документ xml нужно добавить тег

```
<?xml-stylesheet type="text/xsl" href="transform.xsl" ?>
```

Здесь transform.xsl - имя файла листа стилей XSLT.

Рассмотрим пример файла Xml : file.xml, его нужно открыть в браузере.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="file.xslt" ?>
<people>
  <man id= "1">
    <name>John</name>
    <age>30</age>
    <work>Driver</work>
  </man>
  <man id = "2">
    <name>Lisa</name>
    <age>20</age>
    <work>Programmist</work>
  </man>
</people>
```

И файл преобразования: file.xslt.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version = "1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <html>
```

```
<head>
  <title>People</title>
</head>
<body>
  <table border = "1">
      <tbody>
        <xsl:for-each select="people/man">
          <tr>
          <th>
                <xsl:value-of select="@id"/>
            </th>
            <th>
                <xsl:value-of select="name"/>
            </th>
            <th>
           <xsl:value-of select="age"/>
            </th>
            <th>
           <xsl:value-of select="work"/>
            </th>

          </tr>
        </xsl:for-each>
      </tbody>
  </table>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

Если есть необходимость продвигаясь с помощью for-each обратиться в вышестоящему тэгу, можно использовать «../» (как при обращении к вышестоящей папке в операционных системах). Как можно догадаться тэги html встраиваются в шаблон преобразования как обычный текст, шаблоны позволяют выполнять определенные

действия, например, for-each двигается по заданному уровню вложения, select-of вставляет содержание тэга или атрибута. Для обращения к атрибуту указывается значок @.

Попробуем автоматически генерировать на базе xml готовый html файл и отправлять клиенту, используя Python. Для этого будем использовать парсер DOM, его отличие от SAX парсера в том, что парсер DOM загружает полностью документ в память, позволяет работать с документом как с деревом, добавлять новые элементы или удалять их, требует много памяти. SAX же парсер имеет событийный механизм доступа к документу XML и его элементам, требует меньше памяти.

```
# Импортируем библиотеки
import threading
import tornado.websocket
import tornado.web
import tornado.ioloop
#библиотека для преобразования xml с помощью xslt
import lxml.etree as ET

#парсим xml файл в dom
dom = ET.parse("file.xml")
#парсим шаблон в dom
xslt = ET.parse("file.xslt")
#получаем трансформер
transform = ET.XSLT(xslt)
#преобразуем xml с помощью трансформера xslt
newhtml = transform(dom)
#преобразуем из памяти dom в строку, возможно, понадобится указать кодировку
strfile = ET.tostring(newhtml)

#обработчик запросов к http серверу
class MainHandler(tornado.web.RequestHandler):
        #обработка запроса get
        def get(self):
                print('Get')
                #передаем нашу страницу в которой реализована работа с клиентом
```

```
                self.write(strfile)
#создаем приложение tornado с обработчиком вебсоккетов и http сервером
app = tornado.web.Application([(r'/', MainHandler)])
#прослушиваем на порту
app.listen(10556)
print("Start Server")
#запускаем цикл прослушивания и обработки событий
tornado.ioloop.IOLoop.instance().start()
```

Таким образом, имеется механизм автоматического формирования html страницы на базе Xml документа, кроме того, считав документ xml, можно в DOM документе добавлять и изменять сам документ, например, при реакции на события от пользователей.

Полезным может быть:

Работа со строкой содержащей xml, а не с файлом:

```
from io import BytesIO
some_file_or_file_like_object = BytesIO(b"<root>data</root>")
tree = ET.parse(some_file_or_file_like_object)
str1 = ET.tostring(tree)
print(str1)
```

Парсинг прямо из строки:

```
>>> root = ET.XML("<root>data</root>")
>>> print(root.tag)
root
>>> ET.tostring(root)
b'<root>data</root>'
>>> root = ET.HTML("<root>data</root>")
```

Более подробно можно обратиться к:

https://lxml.de/tutorial.html

9. Теоретические вопросы

1. Уровни OSI. Назначение уровней. Сетевые сервисы (сервис печати, сервис СУБД (файлсерверная, серверная архитектура, трехзвенная), сервис сообщений, файловый сервис, сервис приложений (iaas, saas, paas, для чего предназначены данные сервисы, виртуализация на уровне ядра), приведите примеры протоколов обеспечивающих данные сервисы. Понятие интерфейса и протокола. Источники стандартов в телекоммуникациях, сетях и интернете (iso, ieee (802.3, 802.11), isoc, iana (rir)), способы принятия стандартов. Команды ping, ipconfig (ifconfig), tracert (traceroute).

Записать примеры определений функций сервисов (описание их параметров и названия), реализуемые каждым из уровней (предоставляемые вышестоящему уровню). Пример, интерфейсных функций сокетов для транспортного уровня.

2. Основные характеристики протоколов электронной почты. Команды протоколов SMTP, POP3. Система DNS. Mx запись. (как применяется mx dns запрос при отправке почты, какие еще dns запросы бывают, рекурсивный и итеративный запросы, корневые сервера, локальные DNS сервера) Взаимодействие почтовых серверов. MIME формат.(Boundary) . Протокол IMAP. Команда nslookup. Схема взаимодействия почтовых агентов и серверов. Round robin DNS. Кодировка base64. Зачем нужна. Как сделать mx запрос командой nslookup.

3. Команды протокола HTTP. (отличие PATCH и PUT). Коды ответов HTTP (200, 304, 302, 401, 403, 402). Заголовки range, referer, host, expires, vary. Команды протокола FTP. Устройство NAT (Full cone NAT, restricted NAT). Как NAT меняет пакет и сегмент. Привести пример для подмены адресов устройством NAT для различных сетей (192.168.0.0-192.168.255.255, 172.16.0.0-172.32.255.255, 10.0.0.0, 100.0.0.0, какие префиксы у данных сетей), сколько компьютеров может поддерживать NAT с узлами открывшими сколько-то соединений. Проблемы клиента за устройством NAT в активном режиме FTP. Зачем нужен активный и пассивный режим. Почему реализован переход с HTTP1 на HTTP2 и далее на HTTP3. Проблема HOL для HTTP и TCP. В чем проблема текстового протокола HTTP1 и недостатки использования base64. Есть ли у FTP проблема с base64. Зачем FTP возвращает на команду PASV еще и IP, что это дает, зачем вообще FTP два и более соединений в отличие от HTTP.

4. Основные технологии на стороне сервера и клиента. CGI, FAST CGI (чем FAST CGI лучше CGI, через что передаются данные, привести примеры), PHP, ASP, JSP, Фреймворки Node JS, django, ruby on rails, Java script, Dart, SWF, ACTIVEX, Haxe, Wasm.

Зачем нужен балансировщик нагрузки типа NGINX, что такое медленные клиенты и в чем преимущество дополнительного использования NGINX, зачем над веб-сервером на fastapi или flask доп WSGI сервера типа uvicorn, проблема GIL python. JAVA spring, аннотации и декораторы в современных фреймворках для web.

CSRF – межсайтовая подделка запроса. Способы защиты. Xml. json отличие от xml

Xhtml, DOM sax парсер, RDF, OWL семантическая паутина, XQuery, чем foreach отличается от match в Xsl. Способы ускорения выполнения запросов. (отправка данных авторизации в самом запросе). Привести пример xml файла и шаблона преобразования в HTML.

5. Proxy. socks5, цепочка proxy (нарисовать схему цепочки), UDP Associate (нарисовать таблицы преобразования IP и портов), http proxy, http connect proxy, if-modified-since last-modified if-none-match tag, http connect proxy. VPN. IPSec. SSL, TLS. HTTPS. Принцип создания сертификата.

6. Физический уровень.

Методы кодирования. Модуляция (амплитудная, фазовая, частотная, QAM), манипуляция. Потенциальные коды. (AMI, MLT-3, PAM, почему в MLT-3 три уровня, а в AMI 2 на лог. единицу) Импульсные коды. Синхронизация (как достигается). Коды 4b-5b, 8b-6T. Скрэмблирование. Кодирование на четность, по паритету, код хэмминга. Сверточные коды. Мультиплексирование. (TDMA, FDMA, CDMA). Почему OFDMA на исходящее, а SC-FDMA на входящее. Зачем в CDMA ортогональность кодов псевдослучайной последовательности. Хаб, концентратор. Виды проводных физических сред передачи — витая пара, оптоволоконный кабель, коаксиальный кабель, как устроены. Классификация сетей. Локальные, муниципальные и глобальные сети. Сети отделов, кампусов, корпоративные.

7. Протоколы и технологии канального уровня.

Доступ к среде CSMA/CD. Технология Ethernet. Что такое двойной экспоненциальный откат. Что такое коллизия. Почему есть ограничение на минимальный размер кадра, как рассчитывается минимальный размер кадра на основе размера сети, пропускной способности сети. MAC адрес. Формат кадра. Что такое интеллектуальный коммутатор, зачем нужен STA. Что такое широковещательный шторм, зачем нужна сегментация сетей Ethernet, как изменяется интенсивность трафика при сегментации.

Технология Token Ring. Для чего MSAU. Что такое маркерный доступ. Основные скорости передачи. Приоритеты.

Технология FDDI. Зачем двойное кольцо. Размер сети. Скорость передачи.

Технология WiFi. Доступ к среде. CSMA/CA. Difs, sifs, pifs интервалы. ACK, RST,

CST.

8. Маршрутизация

Дистанционны векторные протоколы (DVA), RIP протокол, триггерное обновление, замораживание изменений и расщепление горизонта, route poisoning, poison reverse протокол BGP, IGRP, EIGRP внутренний протокол и внешний протокол маршрутизации IGP EGP. Маршрутные метрики, Алгоритм маршрутизации на основе состояний линий связи (LSA – link state). OSPF, автономные системы в интернете, групповые протоколы маршрутизации, CBT RPF PIM SM, DM, IGMP.

9. UDP. TCP алгоритм медленного пуска, тройного рукопожатия, обеспечение надежности, таймеры TCP, как рассчитывается таймер повторной передачи, SCTP, алгоритм четверного рукопожатия, многопоточность, мультихоуминг, TCP SYN Flood атака. Формат сегмента TCP, SCTP и UDP.

Формат пакета IPV4 и IPV6, отличие v4 от v6, формат адресов, фрагментация, протокол ARP, RARP, NDS, связь адреса IPv6 с MAC, контрольная сумма в 4 и 6 версии, недостатки, формат адреса 6,. Команды ICMP. ICMP флуд атака. Перенаправление шлюза. Эхо запрос, эхо ответ. Подавление трафика.

10. Digest-авторизация. CRAM-md5. AUTH 2.0. Authorization credentials flow + PKCE, Client Credentials Flow, implicit flow, device code flow. JWT токен, scope. Для чего может понадобиться фингер принт. Refresh token. Access token. Что возможно при их перехвате.