Лабораторная работа №5
Программирование сокетов: создание сетевых приложений

Цель лабораторной работы: познакомиться с основами программирования сокетов на языке Python, а также доработаете код соответствующих задач для веб-сервера, UDP-соединений и прокси-сервера.

1. Введение

Типичное сетевое приложение состоит из двух частей – клиентской и серверной программ, работающих на двух различных конечных системах. Когда запускаются эти две программы, создаются два процесса: клиентский и серверный, которые взаимодействуют друг с другом, производя чтение и запись в сокеты. Таким образом, при создании сетевого приложения необходимо написать код как для клиентской, так и для серверной частей приложения.

Существуют два типа сетевых приложений. Для первого из них функционирование описывается стандартами выбранного протокола, такими как RFC или другие документы: такое приложение иногда называют «открытым», так как правила, определяющие операции в нем, известны всем. В такой реализации клиентская и серверная программы должны соответствовать правилам, продиктованным документами RFC.

Например, клиентская программа может быть реализацией клиентской части протокола FTP, определенного в RFC 959; аналогично, серверная программа может быть частью реализации протокола FTP сервера, также описанного в документе RFC 959. Если один разработчик пишет код для клиентской программы, а другой для серверной, и оба внимательно следуют правилам RFC, то эти две программы будут взаимодействовать без проблем. И действительно – многие из сегодняшних сетевых приложений предполагают взаимодействие между клиентскими и серверными программами, которые созданы независимыми разработчиками – например, браузер Firefox, который взаимодействует с веб-сервером Apache, или клиент BitTorrent, взаимодействующий с трекером BitTorrent.

Другой тип сетевых приложений представляют проприетарные (закрытые) приложения. В этом случае клиентские и серверные программы используют протокол прикладного уровня, который открыто не опубликован ни в RFC, ни в каких-либо других документах. Разработчик (либо команда разработчиков) создает клиентские и серверные программы и имеет полный контроль над всем, что происходит в коде. Но так как в кодах программы не реализован открытый протокол, другие независимые разработчики не смогут написать код, который бы взаимодействовал с этим приложением.

2. Программирование сокетов с использованием UDP

Процессы, запущенные на различных устройствах, взаимодействуют друг с другом с помощью отправки сообщений в сокеты. Мы говорили, что каждый процесс похож на дом, а сокет процесса – аналог двери. Приложение работает с одной стороны двери – в доме; протокол транспортного уровня находится по другую сторону двери – во внешнем мире. Разработчик приложения контролирует все, что находится со стороны прикладного уровня, однако со стороны транспортного уровня он мало на что влияет.

Теперь посмотрим детально на взаимодействие между двумя обменивающимися процессами, которые используют UDP-сокеты. Перед тем как отправляющий процесс сможет протолкнуть пакет данных через свой сокет, он должен сначала прикрепить к нему адрес назначения (IP-адрес назначения). После того, как пакет проходит через сокет отправителя, Интернет использует этот адрес назначения, чтобы направить пакет сокету принимающего процесса. Когда пакет прибывает в сокет получателя, принимающий процесс извлекает его через сокет, проверяет содержимое пакета и предпринимает соответствующие действия.

На хосте могут быть запущены несколько процессов сетевых приложений, причем каждый может использовать один или более сокетов, то необходимо идентифицировать определенный сокет на хосте назначения. При создании сокета ему присваивается идентификатор, называемый номером порта. Поэтому адрес назначения включает также и номер порта сокета. Отправляющий процесс прикрепляет к пакету адрес назначения, который состоит из IP-адреса хоста-приемника и номера порта принимающего сокета. Также адрес отправителя, состоящий из IP-адреса хоста-источника и номера порта исходящего сокета, также прикрепляются к пакету. Однако это обычно делается не в коде UDP-приложения, а автоматически операционной системой отправляющего хоста.

Для демонстрации приемов программирования сокетов с использованием UDP и TCP мы будем использовать простое клиент-серверное приложение, суть которого заключается в следующем:

1. Клиент читает строку символов (данные) с клавиатуры и отправляет данные серверу.

2. Сервер получает данные и преобразует символы в верхний регистр.

3. Сервер отправляет измененные данные клиенту.

4. Клиент получает измененные данные и отображает строку на своем экране.

Рисунок 1 отображает основные действия клиента и сервера с сокетами, взаимодействующими через транспортную службу протокола UDP.

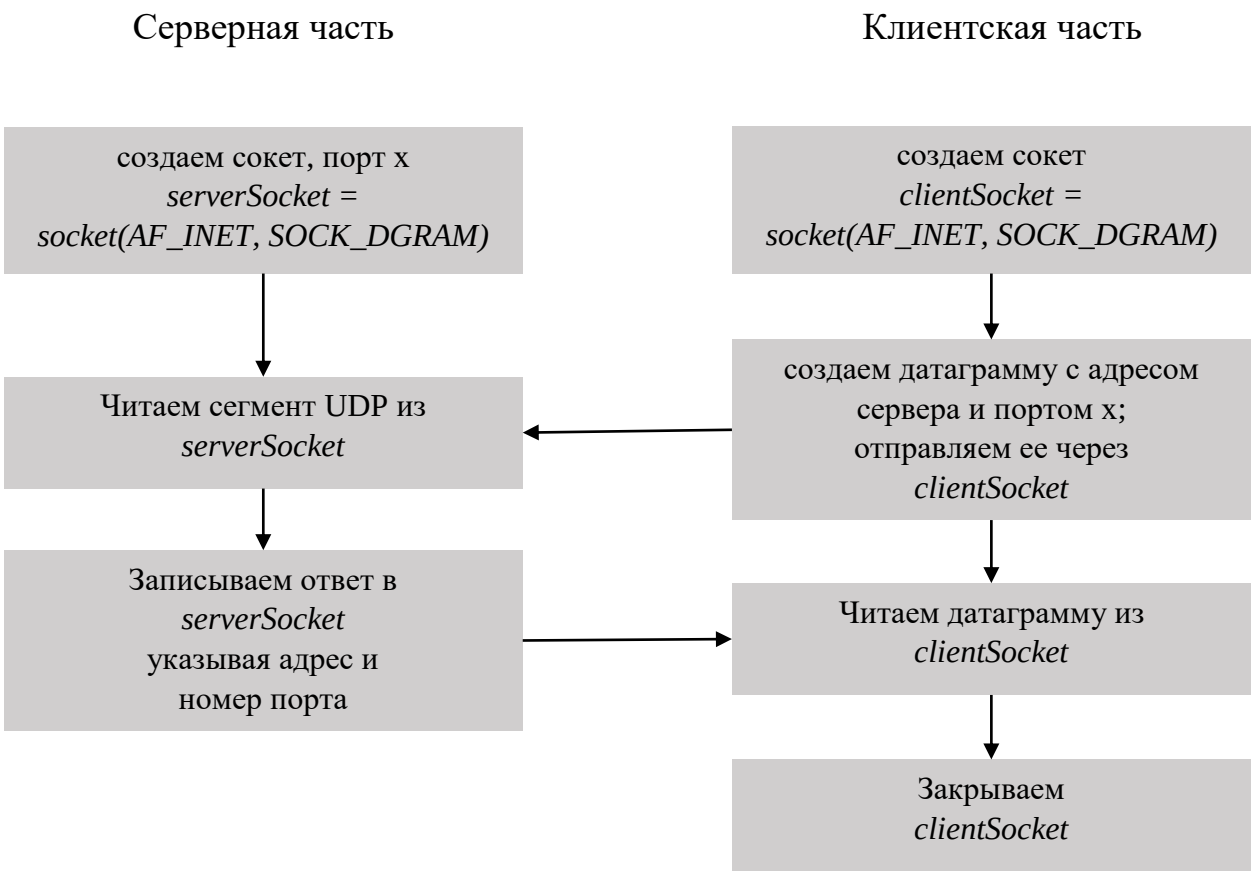

Рисунок 1. – Клиент-серверное приложение, использующее протокол UDP.

В качестве примера предлагается простая программа. Клиентская программа называется UDPClient.py, а серверная – UDPServer.py. На самом деле «правильный код» будет включать гораздо больше строк, например, для обработки ошибок, но мы рассмотрим только основные ключевые моменты. Для нашего приложения мы взяли произвольный номер порта сервера 12 000.

Листинг кода UDPClient.py

```
from socket import *
serverName = 'hostname'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message,(serverName,
serverPort))
modifiedMessage, serverAddress = clientSocket.
recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

*from socket import **

Модуль socket является базовым для всех сетевых взаимодействий, описываемых на языке Python. Включив эту строку, мы сможем создавать сокеты внутри нашей программы.

serverName = 'hostname'

serverPort = 12000

В первой строке мы присваиваем переменной serverName строковое значение 'hostname'. Здесь мы должны поставить строку, содержащую либо IP-адрес сервера (например, «128.138.32.126»), либо имя хоста сервера (например, «cis.poly.edu»). При использовании имени хоста произойдет автоматическое преобразование его в IP-адрес. Во второй строке мы устанавливаем значение целочисленной переменной *serverPort* равным *12000*.

clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)

В этой строке мы создаем сокет клиента, называя его *clientSocket*. Первый параметр определяет семейство адресов; в частности, *AF_INET* указывает, что мы будем использовать протокол IPv4. Второй параметр указывает на тип сокета – *SOCK_DGRAM*, что означает, что это UDP-сокет (а не TCP). Отметим, что мы не задаем номер порта клиентского сокета при его создании – позволяем это сделать за нас операционной системе. Теперь, когда создана «дверь» клиентского процесса, мы можем создавать сообщения и отправлять их через нее.

message = raw_input('Input lowercase sentence:')

raw_input() – это встроенная в Python функция. При ее выполнении пользователю на клиентской стороне предлагается подсказка со словами «Input lowercase sentence» («Введите предложение в нижнем регистре»). После этого пользователь может использовать клавиатуру для ввода строки, которая будет занесена в переменную *message*. Теперь, когда у нас есть сокет и сообщение, мы можем отправлять это сообщение через сокет хосту назначения.

clientSocket.sendto(message,(serverName, serverPort))

В этой строке с помощью метода *sendto()* к сообщению добавляется адрес назначения *(serverName, serverPort)*, и весь результирующий пакет отправляет в сокет процесса – *clientSocket*. (Как указывалось ранее, адрес источника также добавляется к пакету, хотя это делается автоматически, а не явно через строки кода.) На этом отправка сообщения от клиента серверу через сокет UDP закончена. После отправки пакета клиент ожидает получения данных с сервера.

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)

С помощью этой строки данные пакета, прибывающего из Интернета на сокет клиента, помещаются в переменную *modifiedMessage*, а адрес источника пакетов – в переменную *serverAddress*. Последняя переменная содержит как IP-адрес, так и номер порта сервера. В действительности программе UDPClient не

нужна эта информация, так как она уже знает адрес сервера с самого начала, но, тем не менее, данная строка в программе присутствует. Метод *recvfrom()* генерирует входной буфер размером 2048 байт, который используется для различных целей.

print modifiedMessage

Данная строка выводит измененное сообщение на дисплей пользователя. Это первоначальная строка, в которой все символы стали заглавными.

clientSocket.close()

Здесь мы закрываем сокет, и процесс завершается.

Теперь рассмотрим программу серверной части приложения:

Листинг кода UDPServer.py

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print "The server is ready to receive"
while 1:
message, clientAddress = serverSocket.recvfrom(2048)
modifiedMessage = message.upper()
serverSocket.sendto(modifiedMessage, clientAddress)
```

Заметим, что начало программы UDPServer очень похоже на UDPClient. Она также импортирует модуль *socket*, также устанавливает значение целочисленной переменной *serverPort в 12000*, также создает сокет типа *SOCK_DGRAM (UDP)*. Первая, значительно отличающаяся строка, это:

serverSocket.bind(('', serverPort))

Данная строка связывает (то есть назначает) порт под номером 12 000 с сокетом сервера. Таким образом, в программе UDPServer строки кода (написанные разработчиком приложения) явным образом присваивают номер порта сокету. Связывание порта означает, что теперь, если кто-то отправляет пакет на порт 12 000 нашего сервера, то этот пакет будет направлен в данный сокет. Дальше программа UDPServer переходит в бесконечный цикл while, который позволяет получать и обрабатывать пакеты от клиентов в неограниченном количестве.

message, clientAddress = serverSocket.recvfrom(2048)

Приведенная строка очень похожа на то, что мы уже видели в UDPClient. Когда пакет прибывает на сокет сервера, данные пакета помещаются в переменную *message*, а данные источника пакетов – в переменную *clientAddress*.

Переменная *clientAddress* содержит IP-адрес и номер порта клиента. Эта информация будет использоваться программой, так как в ней передается адрес возврата и сервер теперь знает, куда направлять свой ответ.

modifiedMessage = message.upper()

Данная строка является основной в нашем простом приложении. Здесь мы берем введенную клиентом строку и, используя метод *upper()*, меняем ее символы на заглавные.

serverSocket.sendto(modifiedMessage, clientAddress)

Данная последняя строка кода присоединяет адрес клиента (IP-адрес и номер порта) к измененному сообщению и отправляет результирующий пакет в сокет сервера (как уже упоминалось, адрес сервера также присоединяется к пакету, но это делается не в коде, а автоматически). Затем Интернет доставляет пакет на этот клиентский адрес.

Сервер после отправки пакета остается в бесконечном цикле, ожидая прибытия другого UDP-пакета (от любого клиента, запущенного на произвольном хосте).

Чтобы протестировать ваши части программ, просто запустите UDPClient.py на одном хосте и UDPServer.py – на другом. Не забудьте включить правильное имя или IP-адрес сервера в клиентскую часть.

Затем вы запускаете UDPServer.py на серверном хосте, что создает процесс на сервере, который будет ожидать контакта клиента. После этого вы запускаете UDPClient.py на клиенте, тем самым создаете процесс на клиентской машине. Наконец, вы просто вводите предложение, завершая его возвратом каретки.

Вы можете разработать свое собственное клиент-серверное приложение UDP, просто модифицировав клиентскую или серверную части программы.

3. Программирование сокетов с использованием протокола TCP

В отличие от UDP, протокол TCP является протоколом с установлением соединения. Это означает, что клиент и сервер, перед тем как отправлять данные друг другу, сначала обмениваются рукопожатиями и устанавливают TCP-соединение. Одна сторона этого соединения связана с клиентским сокетом, а другая – с серверным. При создании TCP-соединения с ним связывается адрес клиентского сокета (IP-адрес и номер порта) и адрес серверного сокета (IP-адрес и номер порта). Когда одна сторона пытается отправить данные другой стороне с помощью установленного TCP-соединения, она просто передает их в свой сокет. В этом заключается отличие от UDP, в котором сервер вначале должен прикрепить адрес назначения к пакету и лишь затем отправить пакет в сокет.

Теперь рассмотрим подробнее, как взаимодействуют клиентская и серверная программы при работе через TCP. Задача клиента заключается в

инициировании контакта с сервером. Для того чтобы реагировать на начальный контакт клиента, сервер должен быть «готов». Готовность подразумевает под собой две вещи: во-первых, как и в случае с протоколом UDP, TCP-сервер должен быть запущен как процесс, перед тем как клиент попытается инициировать контакт; во-вторых, у серверной программы должна существовать специальная «дверь», а именно специальный сокет, который принимает начальный контакт от клиентского процесса, запущенного на произвольном хосте. Используя нашу аналогию с домом и дверью, мы иногда будем называть начальный контакт клиента как «стук во входную дверь».

Когда серверный процесс запущен, клиентский процесс может инициировать TCP-соединение с сервером. Это осуществляется в клиентской программе с помощью создания TCP-сокета. При этом клиент указывает адрес входного сокета сервера, а именно IP-адрес серверного хоста и номер порта сокета. После создания своего сокета клиент инициирует тройное рукопожатие и устанавливает TCP-соединение с сервером. Данное рукопожатие полностью невидимо для клиентской и серверной программ и происходит на транспортном уровне.

Во время тройного рукопожатия клиентский процесс стучится во входную дверь серверного процесса. Когда сервер «слышит» стук, он открывает новую дверь, а точнее говоря, создает новый сокет, который предназначен этому конкретному стучащемуся клиенту. В нашем примере ниже входная дверь – это TCP-сокет, который мы назвали serverSocket; вновь создаваемый сокет, предназначенный для клиента, который организует соединение, мы назвали connectionSocket. Те, кто рассматривает TCP-сокеты впервые, иногда путает понятия входного сокета (который является начальной точкой контакта для всех клиентов, ожидающих связи с сервером) и вновь создаваемого сокета соединения серверной стороны, который последовательно создается для связи с каждым клиентом.

С точки зрения приложений, клиентский сокет и серверный сокет соединения связаны напрямую. Как показано на рисунке 2, клиентский процесс может отправлять произвольное количество байт в свой сокет и протокол TCP гарантирует, что серверный процесс получит (через сокет соединения) каждый отправленный байт в определенном порядке. Таким образом, TCP предоставляет надежную службу доставки между клиентским и серверным процессами. Так же как люди могут ходить через дверь туда и обратно, клиентский процесс может не только отправлять данные, но и принимать их через свой сокет. Аналогично, серверный процесс не только принимает, но и отправляет данные через свой сокет соединения.

Рисунок 2. – Два сокета серверного процесса.

Мы будем использовать то же самое простое клиент-серверное приложение для демонстрации приемов программирования сокетов с использованием TCP: клиент отправляет на сервер одну строку данных, сервер делает преобразование символов строки в заглавные и отправляет строку обратно клиенту. На рисунке 3 представлена основная схема взаимодействия клиента и сервера, связанная с сокетами, через транспортную службу протокола TCP.

Листинг кода TCPClient.py

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

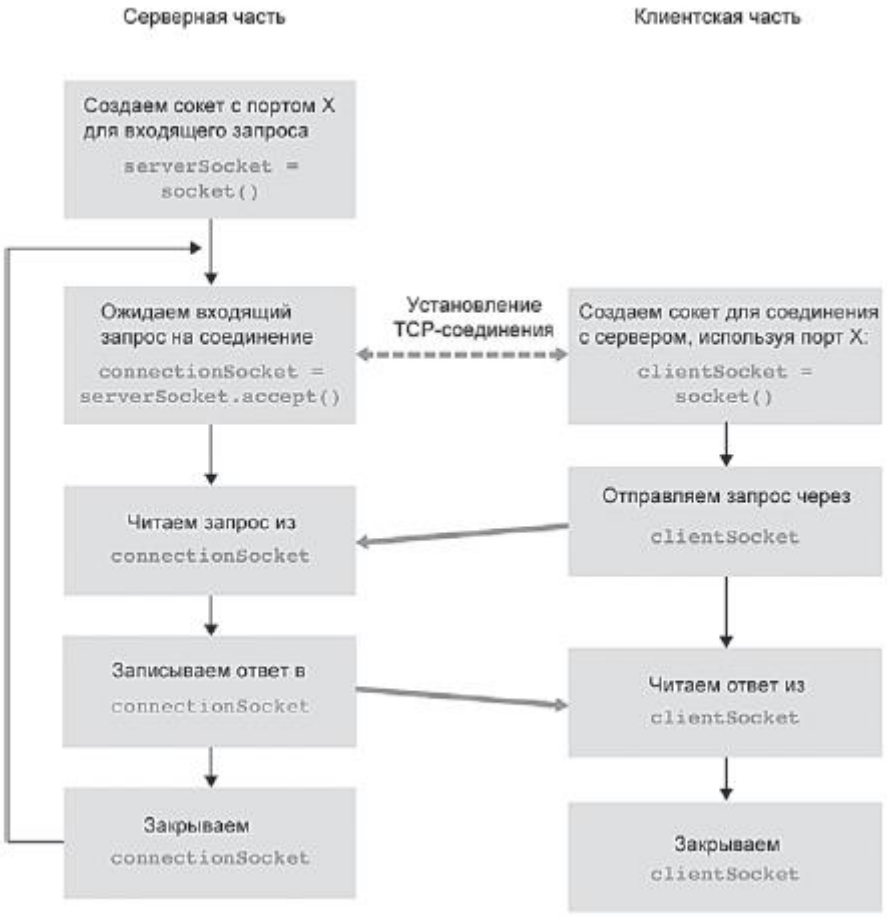

Рисунок 3. – Клиент-серверное приложение, использующее протокол TCP.

Теперь давайте рассмотрим некоторые строки кода, которые значительно отличаются от реализации в случае с протоколом UDP. Первая отличающаяся строка – это создание клиентского сокета.

clientSocket = socket(AF_INET, SOCK_STREAM)

В данной строке создается клиентский сокет, называемый *clientSocket*. Первый параметр, опять же, указывает нам, что мы используем сетевой протокол IPv4. Второй параметр определяет тип сокета (*SOCK_STREAM*), другими словами, это и есть TCP-сокет (а не UDP). Заметим, что мы не указываем номер порта клиентского сокета при его создании, а предоставляем сделать это за нас операционной системе. Следующая строка кода, которая очень отличается от того, что мы видели в UDPClient, это:

clientSocket.connect((serverName,serverPort))

Вспомним, что перед тем, как клиент и сервер смогут посылать данные друг другу, используя TCP-сокет, между ними должно быть установлено TCP-соединение. Строка выше как раз инициирует соединение между клиентом и сервером. Параметром метода *connect()* является адрес серверной части соединения. После исполнения этой строки кода осуществляется тройное рукопожатие, и между клиентом и сервером устанавливается TCP-соединение.

sentence = raw_input('Input lowercase sentence:')

Как и в программе UDPClient, здесь программа получает предложение, введенное пользователем. В строковую переменную *sentence* записываются все символы, вводимые пользователем до тех пор, пока он не введет символ возврата каретки. Следующая строка также сильно отличается от того, что мы видели в программе UDPClient:

clientSocket.send(sentence)

Данная строка отправляет строковую переменную *sentence* через клиентский сокет в TCP-соединение. Заметим, что программа явно не создает пакет и не прикрепляет адрес назначения к нему, как это было в случае с UDP-сокетами.

Вместо этого она просто сбрасывает данные строки *sentence* в TCP-соединение. После этого клиент ожидает получения данных от сервера.

modifiedSentence = clientSocket.recv(2048)

Прибывающие с сервера символы помещаются в строковую переменную *modifiedSentence*. Они продолжают накапливаться в переменной до тех пор, пока в строке не будет обнаружен символ возврата каретки. После печати строки с заглавными символами мы закрываем клиентский сокет:

clientSocket.close()

Последняя строка закрывает сокет, а, следовательно, закрывает TCP-соединение между клиентом и сервером. На самом деле она приводит к тому, что TCP-сообщение с клиента отправляется на сервер.

Листинг кода TCPServer.py

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while 1:
connectionSocket, addr = serverSocket.accept()
sentence = connectionSocket.recv(1024)
capitalizedSentence = sentence.upper()
connectionSocket.send(capitalizedSentence)
connectionSocket.close()
```

Опять же, рассмотрим строки, которые значительно отличаются от представленных в программах UDPServer и TCPClient. Так же, как и в программе TCPClient, сервер создает TCP-сокет с помощью строки :

serverSocket=socket(AF_INET,SOCK_STREAM)

Затем мы связываем номер порта сервера (переменная *serverPort*) с нашим сокетом:

serverSocket.bind(('',serverPort))

Но в случае с TCP переменная *serverSocket* будет являться нашим входным сокетом. После подготовки входной «двери» мы будем ожидать клиентов, стучащихся в нее:

serverSocket.listen(1)

Эта строка означает, что сервер «слушает» запросы от клиентов по TCP-соединению. Параметр в скобках определяет максимальное число поставленных в очередь соединений.

connectionSocket, addr = serverSocket.accept()

Когда клиент стучится в дверь, программа запускает для серверного сокета метод *accept()*, и тот создает новый сокет на сервере – сокет соединения, который предназначен для этого конкретного клиента. Затем клиент и сокет завершают рукопожатие, создавая тем самым TCP-соединение между *clientSocket* и *connectionSocket*, после установления которого клиент и сервер могут обмениваться данными друг с другом, причем данные, с одной стороны, к другой доставляются с гарантией, а также гарантируется порядок их доставки.

connectionSocket.close()

После отправки измененной строки клиенту мы закрываем сокет соединения. Но так как серверный сокет остается открытым, любой другой клиент может постучаться в дверь и отправить серверу новую строку для изменения.

4. Практическое задание

Практическое задание выполняется в группах по два человека либо выполняет один человек. Вариант назначается преподавателем.

Вариант 1

Создать веб-сервер, который может обработать 3 HTTP-запрос. Ваш веб-сервер должен будет принять и проанализировать HTTP-запрос, получить требуемый файл из файловой системы сервера, создать ответное HTTP-сообщение, состоящее из запрошенного файла и предваряющих его строк заголовка, а затем отправить ответ непосредственно клиенту. Если требуемый

файл на сервере отсутствует, клиенту должно вернуться HTTP-сообщение 404 Not Found.

Вариант 2

Реализовать многопоточный сервер, который мог бы обслуживать несколько запросов одновременно. Сначала создайте основной поток (процесс), в котором ваш модифицированный сервер ожидает клиентов на определенном фиксированном порту. При получении запроса на TCP-соединение от клиента он будет устанавливать это соединение через другой порт и обслуживать запрос клиента в отдельном процессе. Таким образом, для каждой пары запрос-ответ будет создаваться отдельное TCP-соединение в отдельном процессе.

Вариант 3

Написать прокси-сервер с обработкой ошибок.

Вариант 4

Написать простой прокси-сервер, который поддерживает только методы GET и POST протокола HTTP. Добавьте поддержку метода POST путем добавления в запрос тела запроса.

Вариант 5

Написать прокси-сервер с кэшированием HTTP страниц.

Вариант 6

Вы должны реализовать следующую программу клиентской части.

Клиент должен отправить 10 эхо-запросов серверу. Поскольку UDP является ненадежным с точки зрения доставки протоколом, пакет, отправленный от клиента к серверу или наоборот, может быть потерян в сети. Так как клиент не может бесконечно ждать ответа на запрос, нужно задать период ожидания ответа (тайм-аут), равный одной секунде – если ответ не будет получен в течение одной секунды, клиентская программа должна предполагать, что пакет потерян. Чтобы узнать, как устанавливать значение тайм-аута сокета, вам следует обратиться к документации по языку Python.

В частности, ваша клиентская программа должна

(1) отправить эхо-запрос, используя UDP (Примечание: в отличие от TCP, вам не нужно сначала устанавливать соединение, так как UDP является протоколом без установления соединения.)

(2) распечатать ответное сообщение от сервера, если таковое имеется

(3) вычислить и вывести на печать время оборота (RTT) в секундах для каждого пакета при ответе сервера

(4) в противном случае, вывести сообщение «Request timed out» (Время запроса истекло)

В процессе разработки вы будете запускать UDPPingerServer.py на вашем компьютере и тестировать работу клиента, отправляя пакеты на localhost (или 127.0.0.1). После того как вы полностью отладите свой код, вы должны увидеть,

как ваши клиентская и серверная части приложения, установленные на разных компьютерах, взаимодействуют по сети.

Вариант 7

Написать клиент-серверное приложение чат.

Вариант 8

Клиент при обращении к серверу получает случайно выбранный сонет Шекспира из файла.

Вариант 9

Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.

5. Контрольные вопросы

Контрольные вопросы по данной лабораторной зависят от задания. Однако это не освобождает Вас от того факта, что надо знать и другое.

6. Приложение

6.1. Шаблон кода веб-сервера на языке Python

```
#импортируем модуль для работы с сокетами
from socket import *
serverSocket = socket(AF_INET, SOCK_STREAM)
#Подготавливаем сокет сервера
#надо дописать
while True:
#Устанавливаем соединение
print 'Готов к обслуживанию...'
connectionSocket, addr = #надо дописать
try:
message = #надо дописать
filename = message.split()[1]
f = open(filename[1:])
outputdata = #надо дописать
#Отправляем в сокет одну строку HTTP-заголовка
#надо дописать
#Отправляем содержимое запрошенного файла клиенту
for i in range(0, len(outputdata)):
connectionSocket.send(outputdata[i])
connectionSocket.close()
except IOError:
#Отправляем ответ об отсутствии файла на сервере
#надо дописать
#Закрываем клиентский сокет
#надо дописать
serverSocket.close()
```

6.2. Шаблон кода прокси-сервера на языке Python

```
from socket import *
import sys
if len(sys.argv) <= 1:
print 'Используйте : "python ProxyServer.py server_ip"\n[server_ip – IP-адрес
прокси-сервера]'
sys.exit(2)
# Создаем серверный сокет, привязываем его к порту и начинаем слушать
tcpSerSock = socket(AF_INET, SOCK_STREAM)
#надо дописать
```

```
while 1:
# Начинаем получать данные от клиента
print 'Готов к обслуживанию...'
tcpCliSock, addr = tcpSerSock.accept()
print 'Установлено соединение с:', addr
message = #надо дописать
print message
# Извлекаем имя файла из сообщения
print message.split()[1]
filename = message.split()[1].partition("/")[2]
print filename
fileExist = "false"
filetouse = "/" + filename
print filetouse
try:
# Проверяем, есть ли файл в кэше
f = open(filetouse[1:], "r")
outputdata = f.readlines()
fileExist = "true"
# Прокси-сервер определяет попадание в кэш и генерирует ответное сообщение
tcpCliSock.send("HTTP/1.0 200 OK\r\n")
tcpCliSock.send("Content-Type:text/html\r\n")
#надо дописать
print 'Читаем из кэша'
# Обработка ошибки, когда файл в кэше не найден
except IOError:
if fileExist == "false":
# Создаем сокет на прокси-сервере
c = # Начало вставки. # Конец вставки.
hostn = filename.replace("www.","",1)
print hostn
try:
# Соединяемся с сокетом по порту 80
#надо дописать
# Создаем временный файл на этом сокете и запрашиваем порт 80 файл, который нужен клиенту
fileobj = c.makefile('r', 0)
fileobj.write("GET "+"http://" + filename + "
HTTP/1.0\n\n")
```

```
# Читаем ответ в буфер
#надо дописать
# Создаем новый файл в кэше для запрашиваемого файла
# А также отправляем ответ из буфера и соответствующий файл на сокет клиента
tmpFile = open("./" + filename,"wb")
#надо дописать
except:
print "Неверный запрос"
else:
# HTTP-ответ для файла не найден
#надо дописать
# Закрываем сокеты клиента и сервера
tcpCliSock.close()
#надо дописать
```

6.3. Эхо запросы через UDP. Код серверной части

```
# UDPPingerServer.py
# Следующий модуль используется для генерирования случайных потерь пакетов
import random
from socket import *
# Создаем UDP-сокет
# Для UDP используем SOCK_DGRAM
serverSocket = socket(AF_INET, SOCK_DGRAM)
# Связываем порт 12000 с сокетом сервера
serverSocket.bind(('', 12000))
while True:
# Генерируем случайное число от 0 до 10
rand = random.randint(0, 10)
# Получаем пакеты от клиента с адресом address
message, address = serverSocket.recvfrom(1024)
# Делаем символы клиентского сообщения заглавными
message = message.upper()
# Если rand меньше 4, считаем пакет потерянным и не выдаем ответ
if rand < 4:
continue
# В противном случае сервер дает ответ
serverSocket.sendto(message, address)
```