Программное обеспечение инфокоммуникационных технологий

Программирование сетевых задач

Колосовский А.В.

Введение

Компьютерный мир глобализируется на основе сетевых коммуникаций и протоколов. Интернет становится обязательным атрибутом повседневности.

Все больше появляется приложений, ориентированных на сеть: это серверы баз данных, сетевые игры, различные сетевые протоколы, Web-серверы, апплеты, сервлеты, CGI-скрипты и т.д. Более того, сеть – это уже компьютер в том случае, когда используется распределенная кластерная архитектура вычислений.

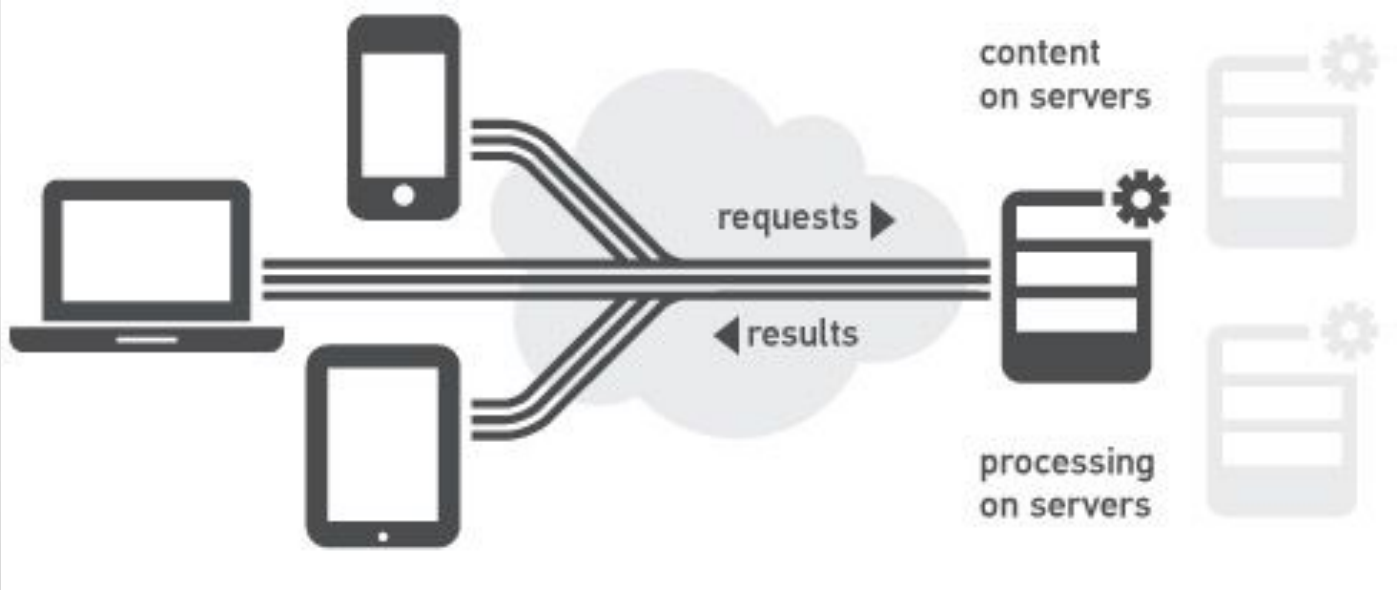
Введение

В области компьютеризации понятие программирования сетевых задач или иначе называемого сетевого программирования (англ. network programming), довольно сильно схожего с понятиями программирование сокетов и клиент-серверное программирование, включает в себя написание компьютерных программ, взаимодействующих с другими программами посредством компьютерной сети.

Введение

Программа или процесс, инициирующие установление связи, называются клиентским процессом, а программа, ожидающая инициации связи, называется серверным процессом. Клиентский и серверный процессы вместе образуют распределенную систему. Связь между клиентским и серверным процессами может быть или на основе соединений (как например, TCP-протокол, устанавливающий виртуальное соединение или сессию), или без соединений (на основе UDP-датаграмм).

Введение


content
on servers
requests
results
processing
on servers

Клиент-сервер

Для обеспечения сетевых коммуникаций используются **сокеты**.

- **Сокет** это конечная точка сетевых коммуникаций. Каждый использующийся сокет имеет тип и ассоциированный с ним процесс. Сокеты существуют внутри коммуникационных доменов. Домены это абстракции, которые подразумевают конкретную структуру адресации и множество протоколов, которое определяет различные типы сокетов внутри домена. Примерами коммуникационных доменов могут быть: UNIX домен, Internet домен, и т.д.

- И у клиента, и у сервера есть socket. Socket связан с определенным IP адресом и номером порта. Обе «стороны» соединения используют socket’ы, и они (socket’ы) платформенно- независимы. Это значит, что машина с Windows может «общаться» по сети с Unix машиной, используя сокеты. По socket’ам данные могут передаваться и приниматься.

Клиент-сервер

Выделяют два типа socket'ов: потоковый socket (**SOCK_STREAM**) и, так называемый, дейтаграммный socket (datagram socket, **SOCK_DGRAM**).

- Потоковый вариант разработан для приложений, нуждающихся в надежном соединении и часто использующем продолжительные потоки данных. Протокол, использующийся для данного типа socket'ов – **TCP**.

- Потоковый вариант чаще всего используется в хорошо известных протоколах, таких как SMTP, POP3, HTTP, TCP.

- Дейтаграммные socket'ы используют **UDP** протокол и имеют низкий сигнал соединения и большой размер буфера данных. Они применяются в приложениях, которые отправляют данные малых размеров и не нуждаются в идеальной надежности. В отличии от потоковых socket'ов, дейтаграммные socket'ы не гарантируют стопроцентной передачи данных получателю, как и не гарантируют передачи данных в нужном порядке.

- Дейтаграммный тип socket'ов полезнее для приложений, где надежность не является высоким приоритетом, таким как скорость (например аудио или видео трансляция). В приложениях, которые нуждаются в надежности, целесообразней использовать потоковые сокеты.

Клиент-сервер

Связывание (binding) socket’ов

- Связать socket значит «прикрепить» определенный адрес (IP адрес и номер порта) к данному socket’у.

Соединение

- Способ использования socket’ов зависит от того, где их необходимо использовать: на клиентской или серверной части.
- Клиентская часть создает соединение путем создания socket’а и вызовом соединяющей функции с определенной адресной информацией. До того как socket не соединится, он не будет связан с адресом.

Это связано с тем, что клиент может использовать любой адрес (IP адрес и номер порта) для соединения с сервером.

Клиент-сервер

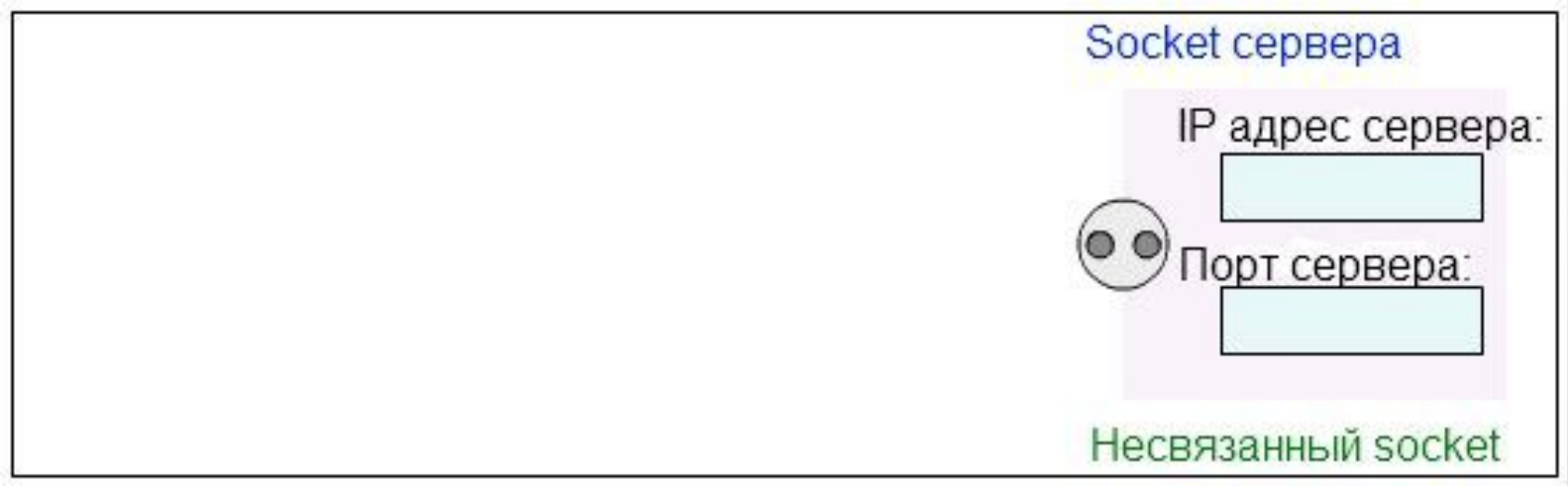
Прослушивание

- На «стороне» сервера дела обстоят немного иначе. Сервер ждет входящих соединений и клиенту необходимо знать IP адрес и номер порта сервера, чтобы установить соединение. Чтобы упростить дело, на сервере всегда используется фиксированный номер порта (обычно это - порт, предусмотренный протоколом по умолчанию).

- Ожидание входящего соединения по определенному адресу называется прослушиванием (listening). Обычно, перед тем как «войти» в режим прослушивания, socket должен быть связан с определенным адресом. Когда номер порта этого адреса установлен и зафиксирован (т.е. не изменится), сервер начинает ждать входящие соединения по этому порту.
  - Например, 80 порт (порт по умолчанию для HTTP) прослушивается большинством серверов.

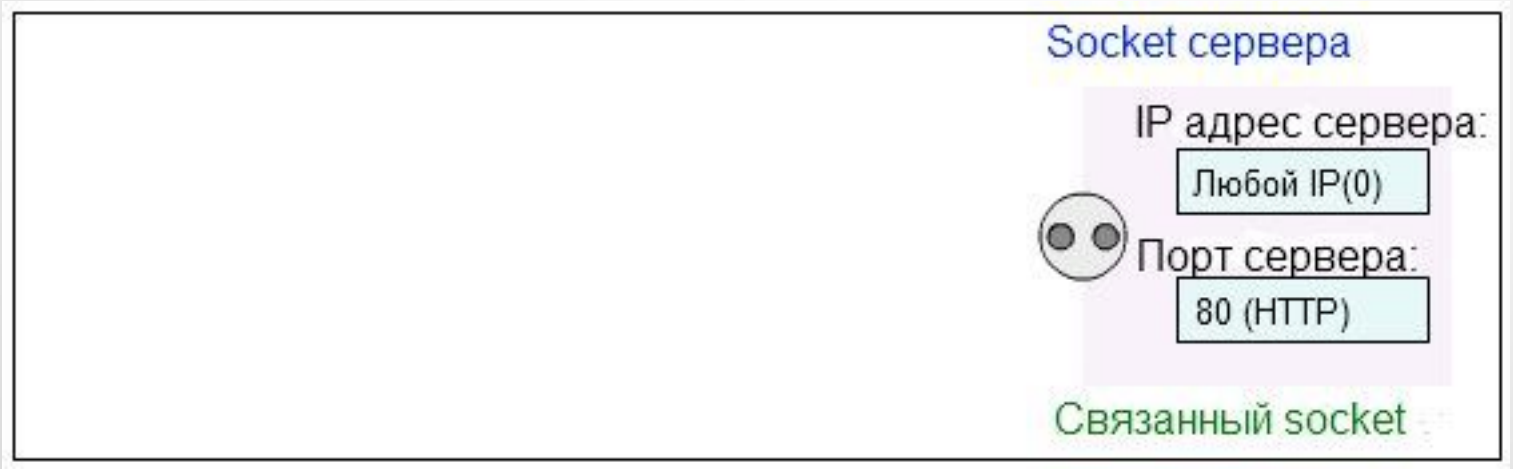
- Когда клиент запрашивает соединение с сервером, сервер разрешит ему (или нет) и породит новый socket, который будет конечной точкой связи. Благодаря этому, socket, по которому происходило прослушивание, не используется для передачи данных и может находиться в режиме прослушивания дальше, «принимая» новых клиентов.

Создание socket’а для сервера



Сервер создает новый socket. Вновь созданный socket еще не связан с IP адресом и портом.

Связь socket’a



Так как наш сервер является сервером какого-нибудь сайта, то порт установлен 80 (порт по умолчанию для HTTP). Однако IP адрес установлен «нулевой», указывая на то, что сервер готов получить соединение от любого IP адреса, доступного компьютеру, на котором он запущен.

- В этом примере предполагается, что у сервера есть три IP адреса: внешний, внутренний и адрес «внутренней петли».

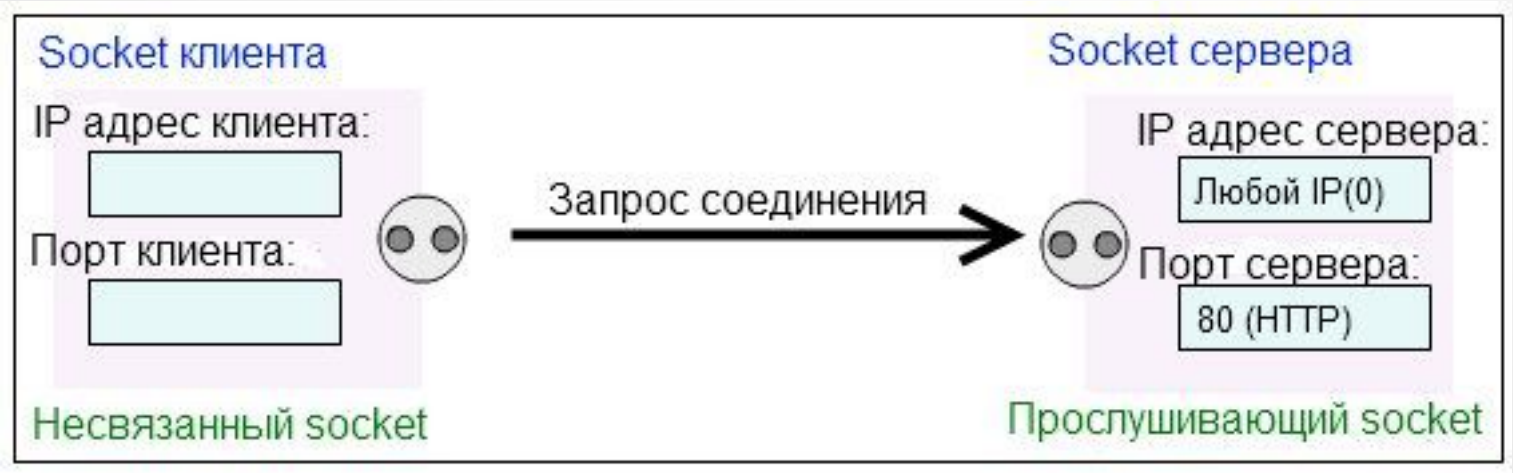
Сервер в режиме прослушивания

После того как socket связан с определенным адресом, он «переходит» в режим прослушивания и ждет входящих соединений по 80ому порту.

Создание socket’a для клиента

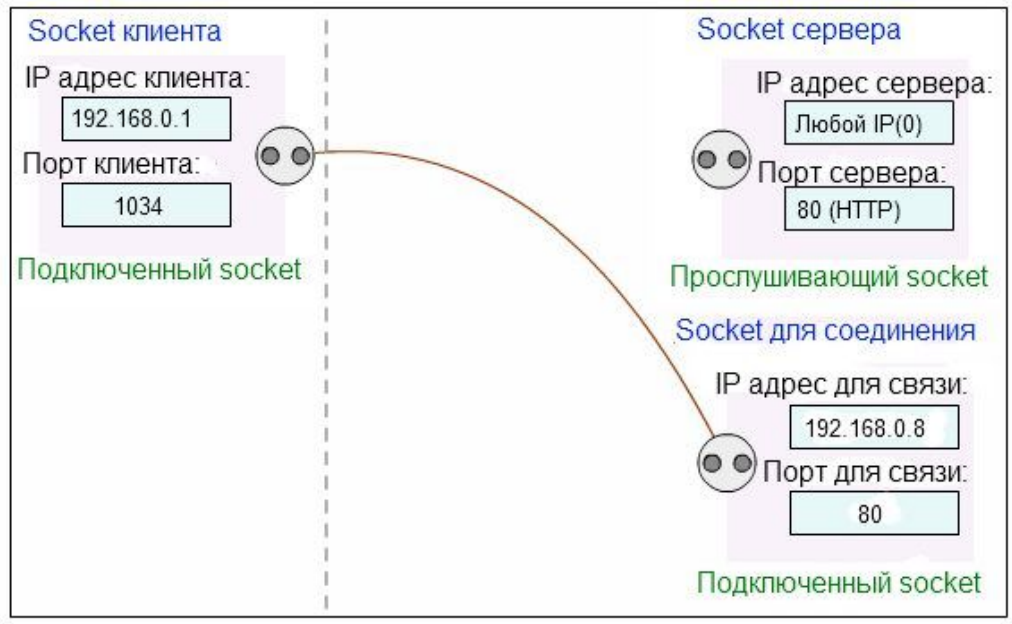
Предположим, что клиент и сервер находятся в одной локальной сети. Клиент хочет запросить страницу с сервера. Чтобы передача данных осуществлялась, клиенту необходим socket, поэтому он и создает его.

Подключение клиента к серверу



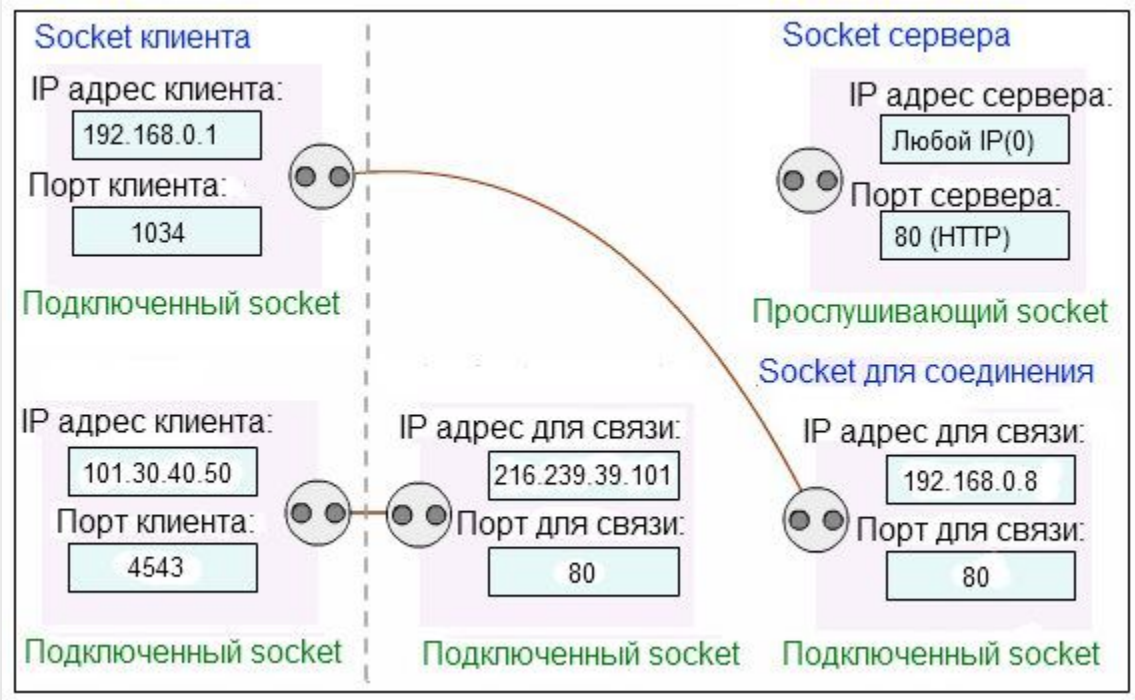
Socket клиента остался несвязанным и пытается запросить соединение с сервером.

Сервер принимает соединение



- Прослушивающий socket замечает, что кто-то пытается подключиться. Он разрешает подключение, создавая новый socket, связывая его с одним из адресов, который может «достичь» клиент (в нашем примере клиент и сервер в одной локальной сети, поэтому IP любой в диапазоне 192.168.x.x).
- Socket клиента и socket сервера для подключения будут осуществлять передачу данных друг другу, в то время как прослушивающий socket будет ждать новое соединение. Следует заметить, что socket клиента связан с IP адресом и номером порта клиента пока клиент подключен к серверу.

Подключение других клиентов



Если другой клиент (из внешней сети) подключается, сервер создает еще один socket для взаимодействия с ним (новым клиентом).
Следует заметить, что IP адрес, с которым связывается только что созданный socket, отличается от того, с которым связался первый socket. Это возможно, потому что прослушивающий socket сервера не связан с определенным IP адресом. Если бы он был связан с адресом 192.168.0.8, то второй клиент не смог бы подключится.

Принципы сокетов

- Каждый процесс может создать слушающий сокет (серверный сокет) и привязать его к какому-нибудь порту операционной системы (в UNIX непривилегированные процессы не могут использовать порты меньше 1024). Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и т.д.

- Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (файл сокета) по заданному пути, через который смогут сообщаться любые локальные процессы путём чтения/записи из него (**IPC-сокет**). Сокеты типа INET доступны из сети и требуют выделения номера порта.

- Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

Принципы сокетов

- В программе сокет идентифицируется дескриптором - это просто переменная типа **int**. Программа получает дескриптор от операционной системы при создании сокета, а затем передает его сервисам ***socket API*** для указания сокета, над которым необходимо выполнить то или иное действие.

Основные функции сокетов

Общие:

Socket - Создать новый сокет и вернуть файловый дескриптор
Send - Отправить данные по сети
Receive - Получить данные из сети
Close -Закрыть соединение

Серверные:

Bind - Связать сокет с IP-адресом и портом
Listen - Объявить о желании принимать соединения. Слушает порт и ждет когда будет установлено соединение
Accept - Принять запрос на установку соединения

Клиентские:

Connect - Установить соединение

Функция socket()

Функция **socket()** создаёт конечную точку соединения и возвращает дескриптор и принимает три аргумента:

1. domain указывающий семейство протоколов создаваемого сокета

- AF_INET для сетевого протокола IPv4
- AF_INET6 для IPv6
- AF_UNIX для локальных сокетов (используя файл)

2. type

- SOCK_STREAM (надёжная потокоориентированная служба (сервис) или потоковый сокет)
- SOCK_DGRAM (служба датаграмм или датаграммный сокет)
- SOCK_RAW (Сырой сокет — сырой протокол поверх сетевого уровня)

3. Protocol

- Функция возвращает −1 в случае ошибки. Иначе, она возвращает целое число, представляющее присвоенный дескриптор.

Функция bind()

Функция **bind()** связывает сокет с конкретным адресом. Когда сокет создается при помощи **socket()**, он ассоциируется с некоторым семейством адресов, но не с конкретным адресом. До того как сокет сможет принять входящие соединения, он должен быть связан с адресом. **bind()** принимает три аргумента:

1. sockfd — дескриптор, представляющий сокет при привязке.
2. serv_addr — указатель на структуру sockaddr, представляющую адрес, к которому привязываем.
3. addrlen — поле socklen_t, представляющее длину структуры sockaddr.

Функция возвращает 0 при успехе и −1 при возникновении ошибки.

Функция listen()

Функция **listen()** подготавливает привязываемый сокет к принятию входящих соединений. Данная функция применима только к типам сокетов **SOCK_STREAM** и **SOCK_SEQPACKET**. Принимает два аргумента:

1. sockfd — корректный дескриптор сокета.
2. backlog — целое число, означающее число установленных соединений, которые могут быть обработаны в любой момент времени. Операционная система обычно ставит его равным максимальному значению.

- После принятия соединения оно выводится из очереди. В случае успеха возвращается 0, в случае возникновения ошибки возвращается −1.

Функция accept()

Функция **accept()** используется для принятия запроса на установление соединения от удаленного хоста. Принимает следующие аргументы:

1. sockfd — дескриптор слушающего сокета на принятие соединения.
2. cliaddr — указатель на структуру sockaddr, для принятия информации об адресе клиента.
3. addrlen — указатель на socklen_t, определяющее размер структуры, содержащей клиентский адрес и переданной в **accept()**. Когда **accept()** возвращает некоторое значение, socklen_t указывает сколько байт структуры cliaddr использовано в данный момент.

- Функция возвращает дескриптор сокета, связанный с принятым соединением, или −1 в случае возникновения ошибки.

Функция connect()

Функция **connect()** устанавливает соединение с сервером.

Некоторые типы сокетов работают без установления соединения, это в основном касается UDP-сокетов. Для них соединение приобретает особое значение: цель по умолчанию для посылки и получения данных присваивается переданному адресу, позволяя использовать такие функции как **send()** и **recv()** на сокетах без установления соединения.

Загруженный сервер может отвергнуть попытку соединения, поэтому в некоторых видах программ необходимо предусмотреть повторные попытки соединения.

Возвращает целое число, представляющее код ошибки: 0 означает успешное выполнение, а −1 свидетельствует об ошибке.

Передача данных

Для передачи данных можно пользоваться стандартными функциями чтения/записи файлов **read** и **write**, но есть специальные функции для передачи данных через сокеты:

- send() - отправка данных
- recv() - чтение данных из сокета
- sendto() - отправка данных
- recvfrom() - чтение данных из сокета
- sendmsg() - отправляет сообщения в сокет
- recvmsg() - получить сообщение из сокета

- Нужно обратить внимание, что при использовании протокола TCP (сокеты типа **SOCK_STREAM**) есть вероятность получить меньше данных, чем было передано, так как ещё не все данные были переданы, поэтому нужно либо дождаться, когда функция recv возвратит 0 байт, либо выставить флаг **MSG_WAITALL** для функции recv, что заставит её дождаться окончания передачи. Для остальных типов сокетов флаг **MSG_WAITALL** ничего не меняет (например, в UDP весь пакет = целое сообщение).

Передача данных через UNIX сокеты

- Сокет домена Unix (англ. Unix domain socket, UDS) или **IPC-сокет** (**сокет межпроцессного взаимодействия**) — конечная точка обмена данными, подобная Интернет-сокету, но не использующая сетевой протокол для взаимодействия (обмена данными). Используется в операционных системах, поддерживающих стандарт **POSIX**, для межпроцессного взаимодействия.

- Доменные соединения Unix являются по сути байтовыми потоками, сильно напоминая сетевые соединения, но при этом все данные остаются внутри одного компьютера (то есть обмен данными происходит локально). **UDS (Unix domain socket)** используют файловую систему как адресное пространство имен, то есть они представляются процессами как иноды в файловой системе. Это позволяет двум различным процессам открывать один и тот же сокет для взаимодействия между собой. Однако, конкретное взаимодействие, обмен данными, не использует файловую систему, а только буферы памяти ядра.

- **Инод** (**индексный дескриптор**) — это структура данных в традиционных для ОС UNIX файловых системах (ФС), таких как UFS. В этой структуре хранится метаинформация о стандартных файлах, каталогах или других объектах файловой системы, кроме непосредственно данных и имени.

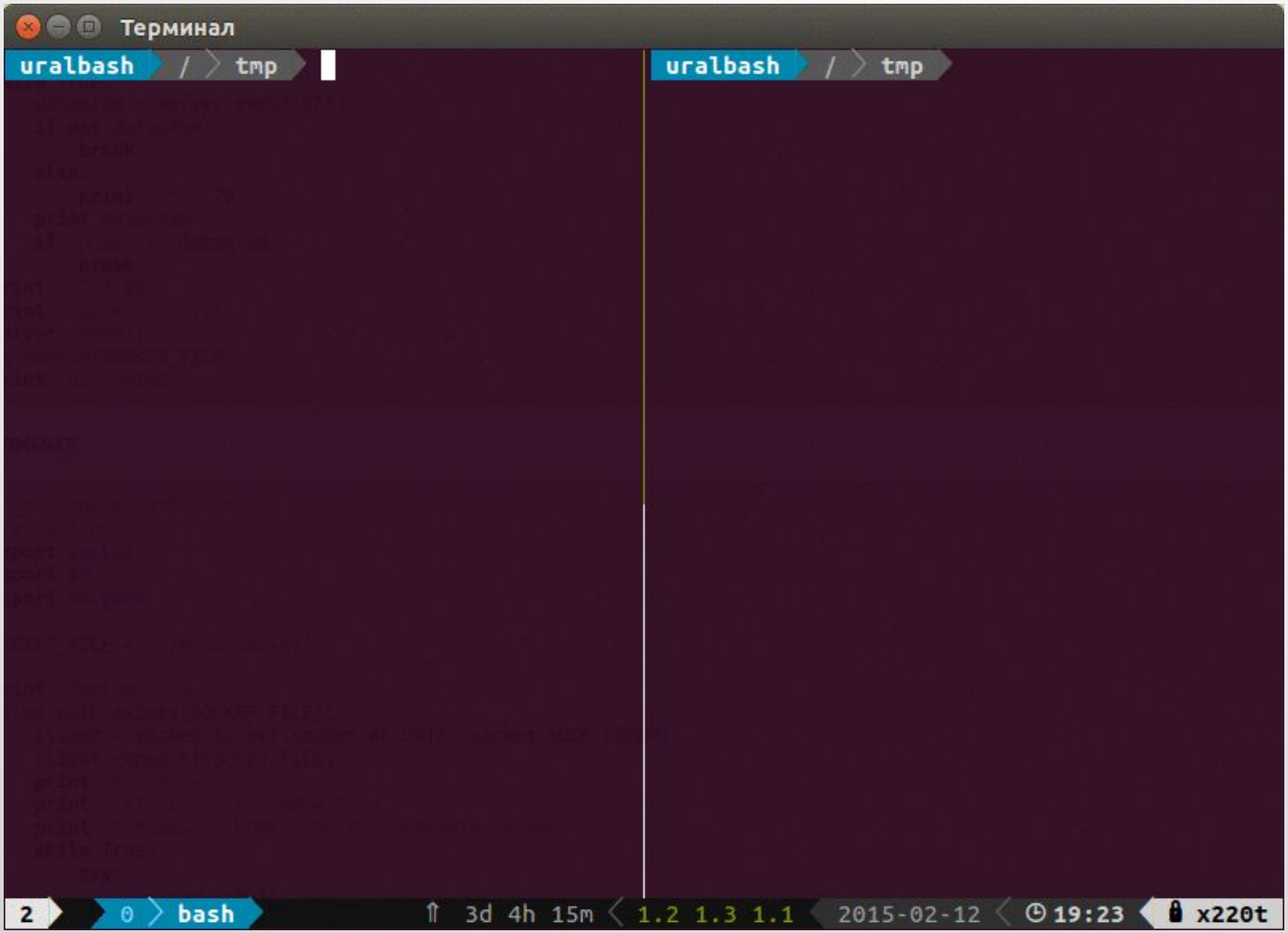
Пример передачи в одну сторону через UNIX сокеты - сервер

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim:fenc=utf-8
#
# Copyright © 2015 uralbash <root@uralbash.ru>
#
# Distributed under terms of the MIT license.

"""
Unix socket server
"""
import os
import socket

SOCKET_FILE = './echo.socket'

if os.path.exists(SOCKET_FILE):
    os.remove(SOCKET_FILE)

print("Открываем UNIX сокет...")
server = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
server.bind(SOCKET_FILE)

print("Слушаем...")
while True:
    datagram = server.recv(1024)
    if not datagram:
        break
    else:
        print("-" * 20)
    print(datagram)
    if "DONE" == datagram:
        break
print("-" * 20)
print("Выключение...")
server.close()
os.remove(SOCKET_FILE)
print("Выполнено")
```

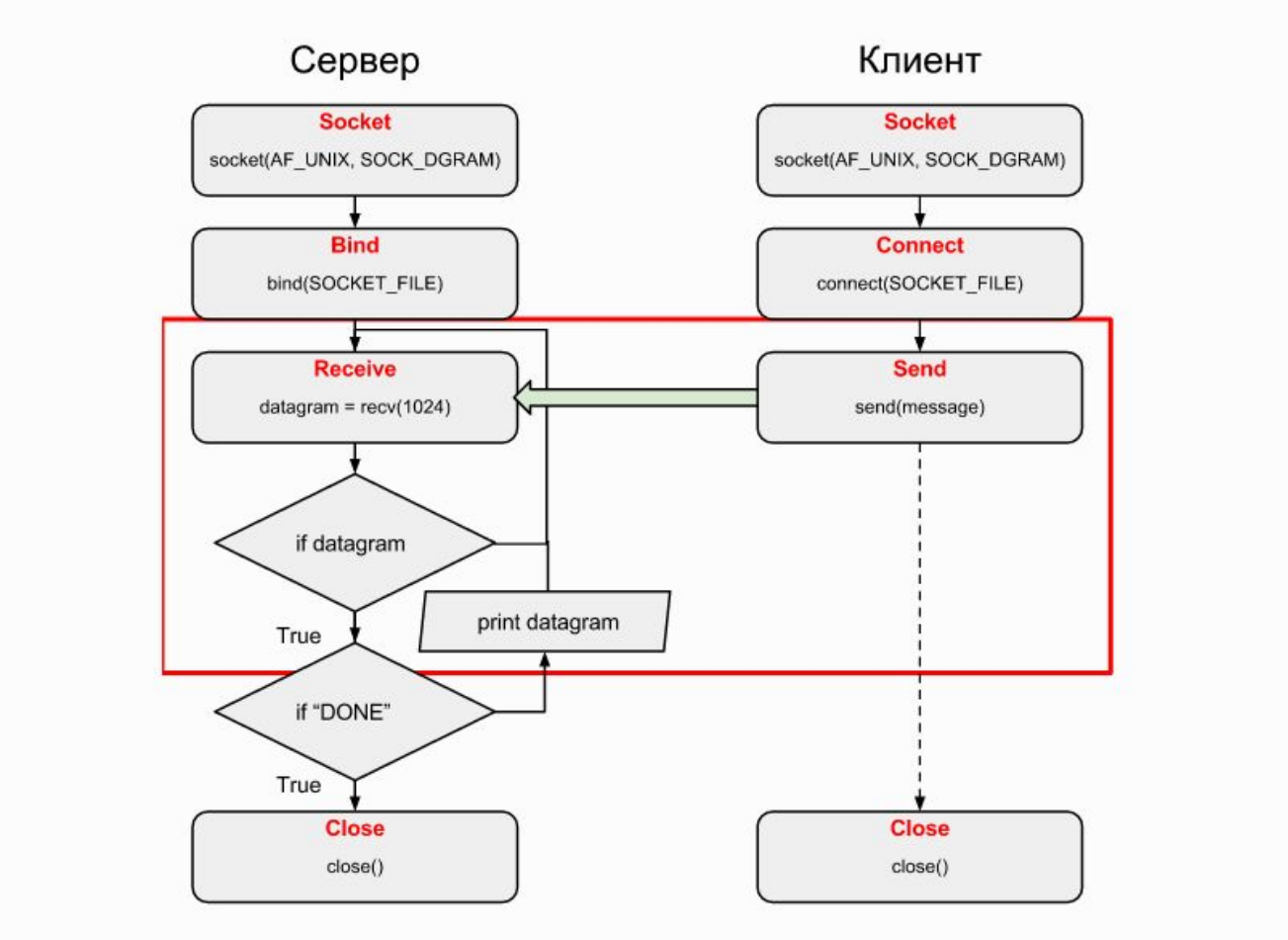
Пример передачи в одну сторону через UNIX сокеты - клиент

```python
#! /usr/bin/env python
# -*- coding: utf-8 -*-
# vim:fenc=utf-8
#
# Copyright © 2015 uralbash <root@uralbash.ru>
#
# Distributed under terms of the MIT license.

"""
Unix socket client
"""
import os
import socket

SOCKET_FILE = './echo.socket'

print("Подключение...")
if os.path.exists(SOCKET_FILE):
    client = socket.socket(socket.AF_UNIX, socket.SOCK_DGRAM)
    client.connect(SOCKET_FILE)
    print("Выполнено.")
    print("Ctrl-C что бы выйти.")
    print("Отправьте 'DONE' что бы выключить сервер.")
    while True:
        try:
            x = input("> ")  # for py2 use raw_input
            if "" != x:
                print("ОТПРАВЛЕНО: %s" % x)
                client.send(x.encode('utf-8'))
                if "DONE" == x:
                    print("Выключение.")
                    break
        except KeyboardInterrupt as k:
            print("Выключение.")
            break
    client.close()
else:
    print("Не могу соединиться!")
print("Выполнено")
```

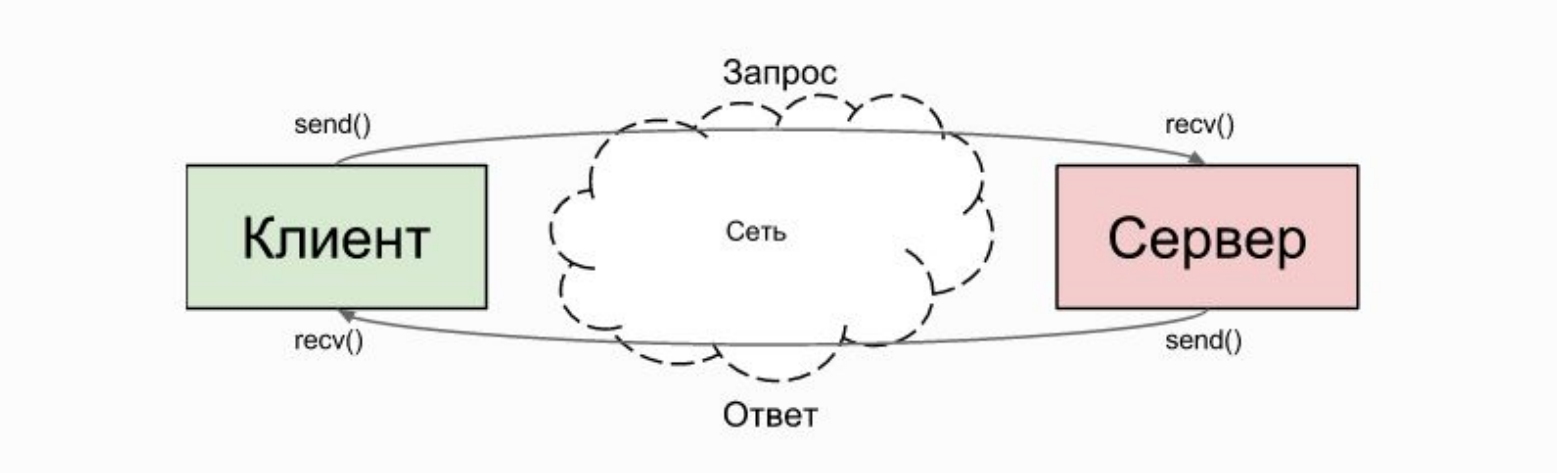
Пример передачи в одну сторону



Пример передачи в одну сторону - схематичное отображение



Передача данных через INET сокеты



Передача данных через INET сокеты - TCP пример

TCP
клиент

```python
import socket

TCP_IP = '127.0.0.1'
TCP_PORT = 5005
BUFFER_SIZE = 1024
MESSAGE = "Hello, World!"

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((TCP_IP, TCP_PORT))
s.send(MESSAGE)
data = s.recv(BUFFER_SIZE)
s.close()

print("received data: {}".format(data))
```

В роли клиента может выступать утилита telnet:

$ telnet localhost 5005

Передача данных через INET сокеты - TCP пример

TCP

```python
import socket

TCP_IP = '127.0.0.1'
TCP_PORT = 5005
BUFFER_SIZE = 20  # Normally 1024, but we want fast response

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(1)

conn, addr = s.accept()
print("Connection address: {}".format(addr))
while 1:
    data = conn.recv(BUFFER_SIZE)
    if not data:
        break
    print("received data: {}".format(data))
    conn.send(data)  # echo
conn.close()
```

Способы определения длины сообщения:

- Передать отдельно
- Читать до разделителя (в http это пустая строка)
- Передать в заголовке (в http это content-length)
- Договориться что размер будет фиксированным (как в примере)
- Читать данные пока не вернется 0

Передача данных через INET сокеты - TCP пример

Передача данных через INET сокеты - UDP пример

UDP

```
1   import socket
2
3   UDP_IP = "127.0.0.1"
4   UDP_PORT = 5005
5   MESSAGE = "Hello, World!"
6
7   print("UDP target IP: {}".format(UDP_IP))
8   print("UDP target port: {}".format(UDP_PORT))
9   print("message: {}".format(MESSAGE))
10
11  sock = socket.socket(socket.AF_INET,     # Internet
12                       socket.SOCK_DGRAM)  # UDP
13  sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
```

В роли клиента может выступать утилита netcat:

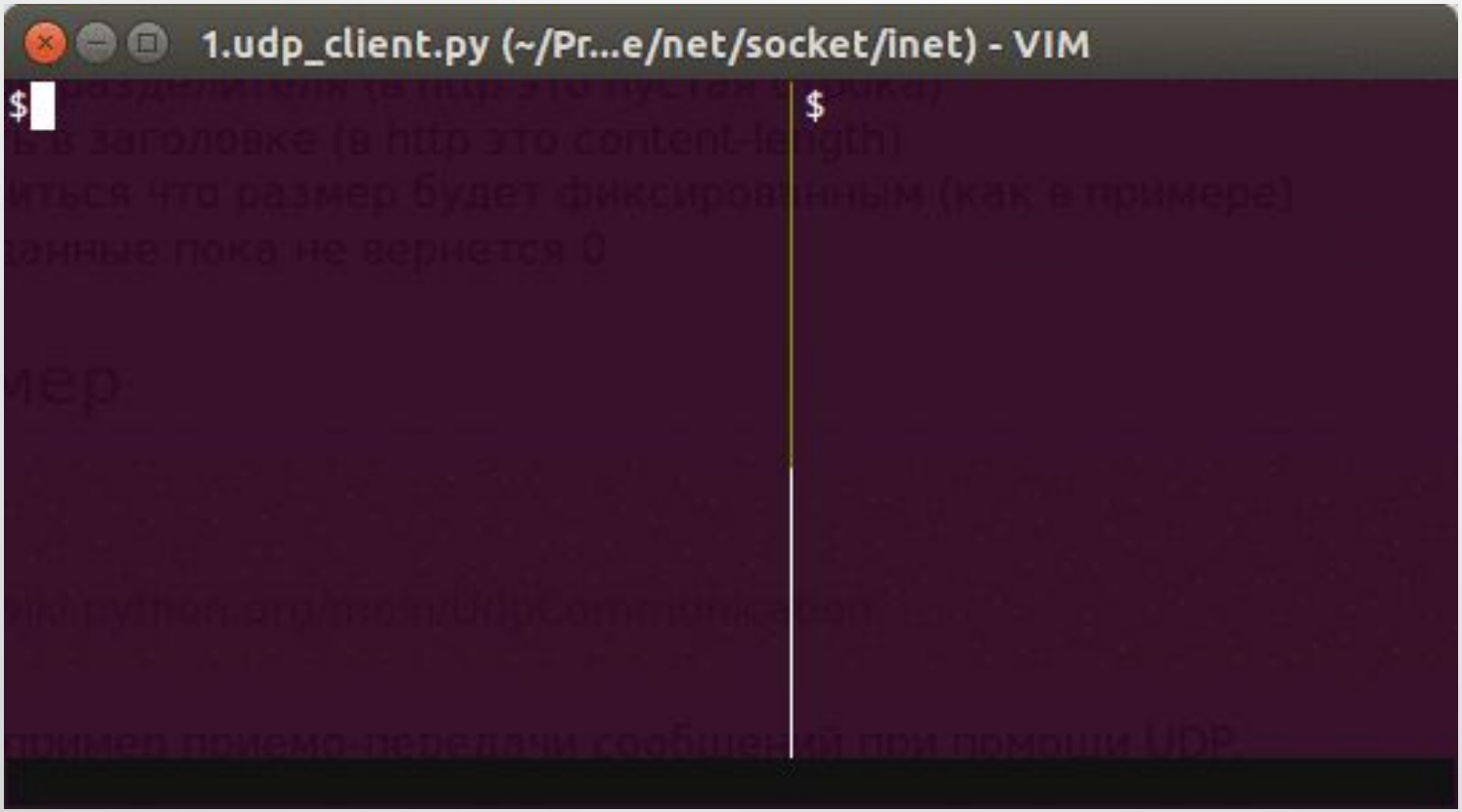
$ nc 127.0.0.1 5005 -u

Передача данных через INET сокеты - UDP пример

UDP

```
1   import socket
2
3   UDP_IP = "127.0.0.1"
4   UDP_PORT = 5005
5
6   sock = socket.socket(socket.AF_INET,     # Internet
7                        socket.SOCK_DGRAM)  # UDP
8   sock.bind((UDP_IP, UDP_PORT))
9
10  while True:
11      data, addr = sock.recvfrom(1024)  # buffer size is 1024 bytes
12      print("received message: {}".format(data))
```

Передача данных через INET сокеты - UDP пример



Сырые сокеты

- **Сырой сокет (raw)** - разновидность сокетов Беркли, позволяющий собирать TCP/IP-пакеты, контролируя каждый бит заголовка и отправляя в сеть нестандартные пакеты.

- Данные сокеты позволяют получить доступ к базовому протоколу передачи данных, это дает нам большие возможности создания таких сетевых утилит как Ping, Traceroute, а также для организации IP Spoofing, DDoS, ICMP-flood, и многого еще =)