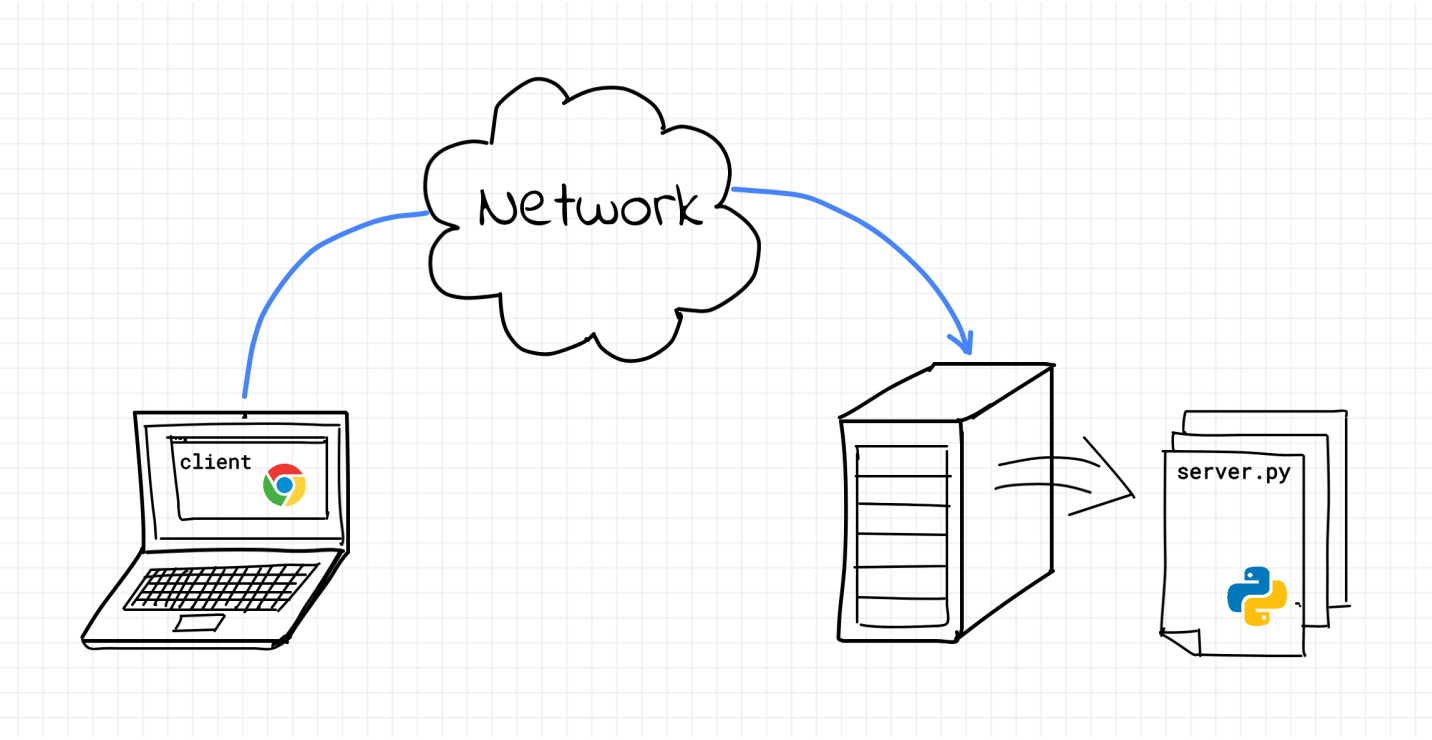
Writing Web Server in Python: sockets

December 27, 2020 (Updated: January 1, 2021)
• Tutorial - Writing Web Server in Python, Networking

What is a web server?

Let's start by answering the question: *What is a web server*?

First off, it's a server (no pun intended). A server is a process [*sic*] serving clients. Surprisingly or not, a server has nothing to do with hardware. It's just a regular piece of software run by an operating system. Like most other programs around, a server gets some data on its input, transforms data in accordance with some business logic, and then produces some output data. In the case of a *web server*, the input and output happen over the network via *Hypertext Transfer Protocol (HTTP)*. For a web server, the input consists of HTTP requests from its clients - web browsers, mobile applications, IoT devices, or even other web services. And the output consists of HTTP responses, oftentimes in form of HTML pages, but other formats are also supported.

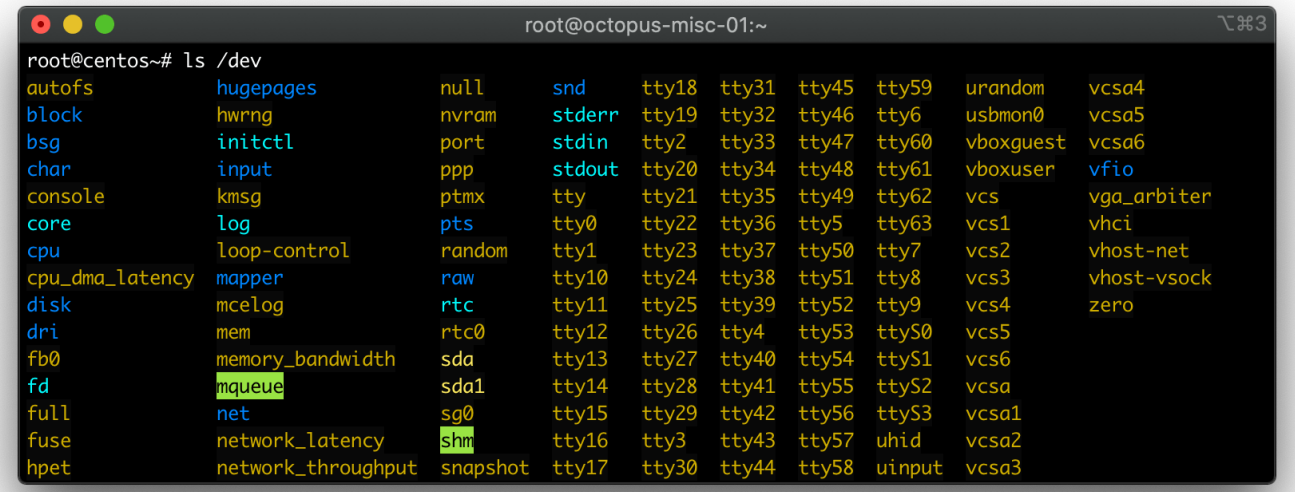


What is this *Hypertext Transfer Protocol*? Well, at this point, it'd be sufficient to think of it as a text-based (i.e. human-readable) data exchange protocol. And the word *protocol* can be

explained as a sort of convention between two or more parties on data transfer format and rules. No worries, there is going to be an article covering the details of HTTP later, while the rest of this article will be focused on how computers send arbitrary data over the network.

What networking programming look alike?

In Unix-like operating systems, it's pretty common to treat I/O devices as files. Much like regular files on disk, computer mice, printers, modems, etc can be opened, read/written, and then closed.



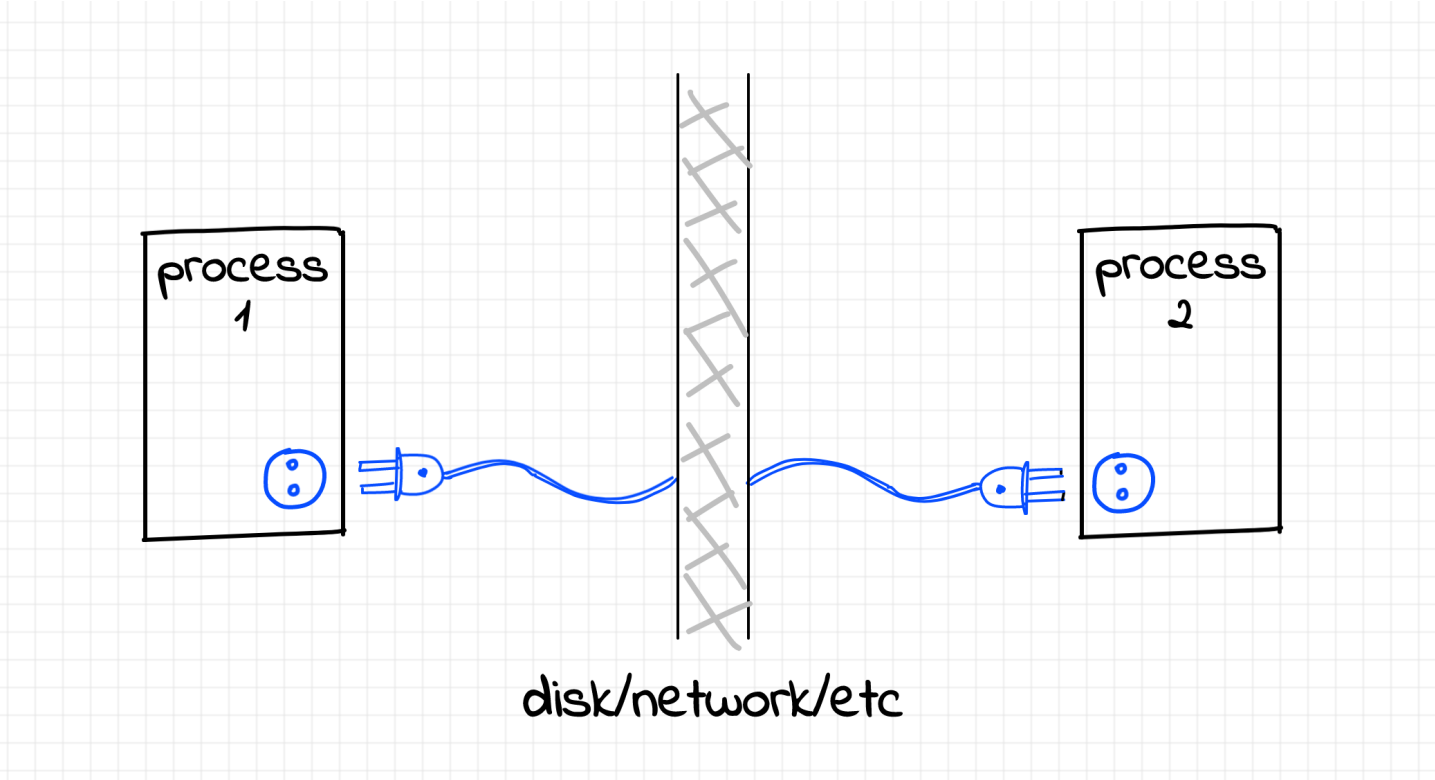
For every opened file, the operating system creates a so-called file descriptor. Simplifying a bit, a file descriptor is just a unique integer identifier of a file within a process. The operating system provides a set of ~~functions~~ system calls to manipulate files that accept a file descriptor as an argument. Here is a canonical example with `read()` and `write()` operations:

```
// C-ish pseudocode

int fd = open("/path/to/my/file", ...);

char buffer[2048];
read(fd, buffer, 2048);
write(fd, "some data", 10);

close(fd);
```

Since network communication is also a form of I/O, it'd be reasonable to expect that it should boil down to working with files as well. And indeed, there is a special kind of file for that called

sockets.

A socket is yet another piece of abstraction provided by the operating system. As it often happens with computer abstractions, the concept borrowed from the real world - more specifically from the AC power sockets. A pair of sockets allow two processes to talk to each other. In particular, over the network. A socket can be opened, data can be written to the socket or read from it. And of course, when the socket is not needed anymore, it should be closed.



Sockets are pretty diverse, and there are many ways to use them for inter-process communication. For instance, network sockets can be utilized when two processes reside on different machines. For local processes, Unix domain sockets may be a better choice. But even these two kinds of sockets can be of different types: datagram (or UDP), stream (or TCP), raw sockets, etc. This variety may seem complicated at first, but luckily there is a more or less generic approach on how to use sockets of any kind in code. Learning how to program one of these socket types will give you the ability to extrapolate the knowledge to others.

```
// C-ish pseudocode again

int fd = socket(SOCK_TYPE_TCP);

sockaddr serv_addr = { /* ... */ };
connect(fd, serv_addr);

char buffer[2048];
read(fd, buffer, 2048);
write(fd, "some data", 10);
```

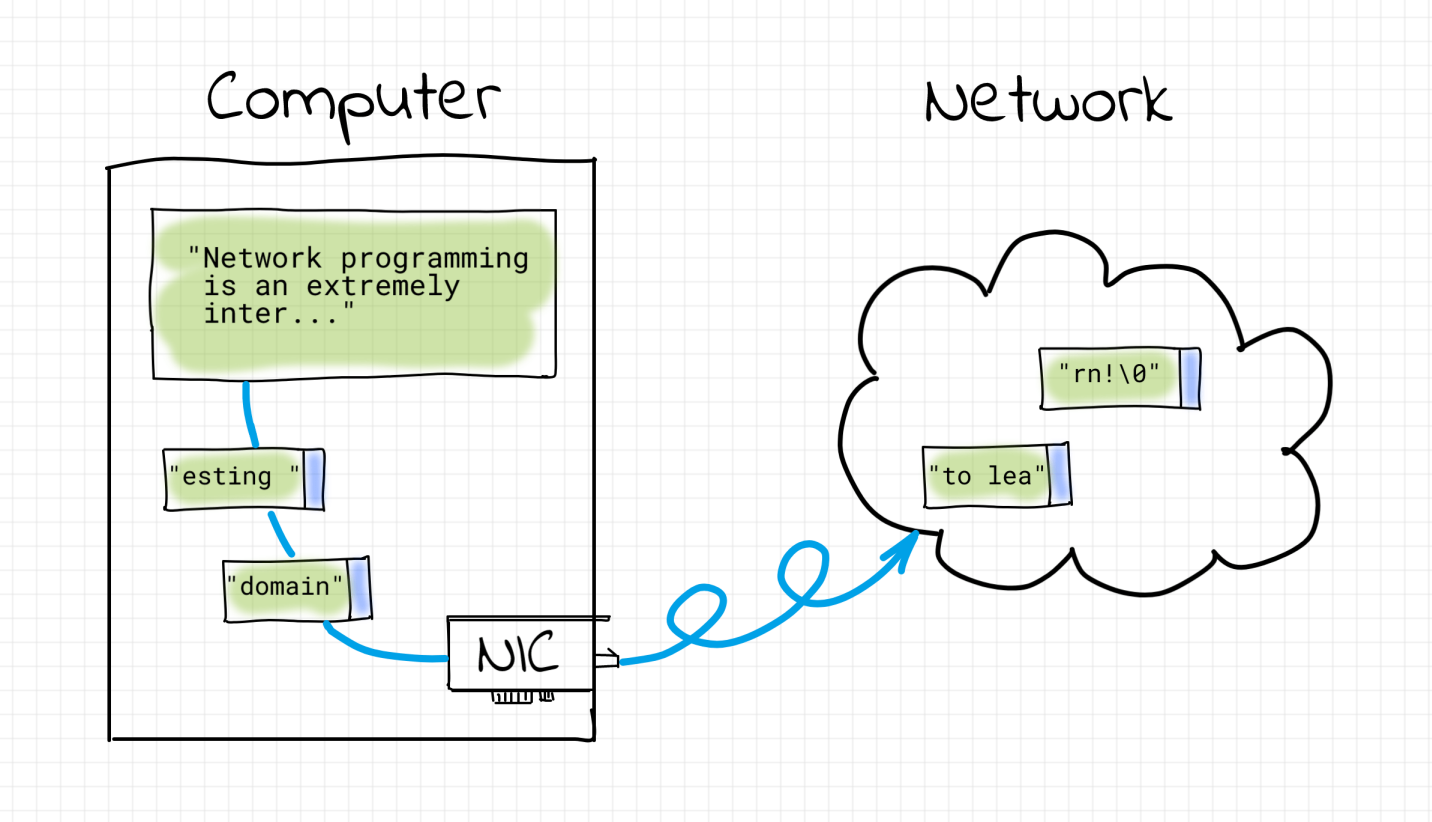
```
close(fd);
```

Further in this article, we'll focus on a form of client-server communication via network sockets using TCP/IP stack of protocols. Apparently, this is the most widely used form nowadays, in particular, because browsers use it to access web sites.

How programs communicate over network

Imagine there is an application that wants to send a relatively long piece of text over the network. Let's suppose the socket has already been opened and the program is about to `write` (or, in networking parlance, `send`) this data to the socket. How this data will be transmitted?

Computers live in a discrete world. Network interface cards (NIC) transmit data in small portions - a few hundred bytes at once. At the same time, in general, the size of the data that a program may want to send is not limited anyhow and can exceed hundreds of gigabytes. To transmit an arbitrary large piece of data over the network, it needs to be chunked and every chunk should be sent separately. Logically, the chunk max size should not exceed the limitation of the network adapter.



Each chunk consists of two parts - control information and the payload. The control information includes source and destination addresses, the chunk size, a checksum, etc. While the payload

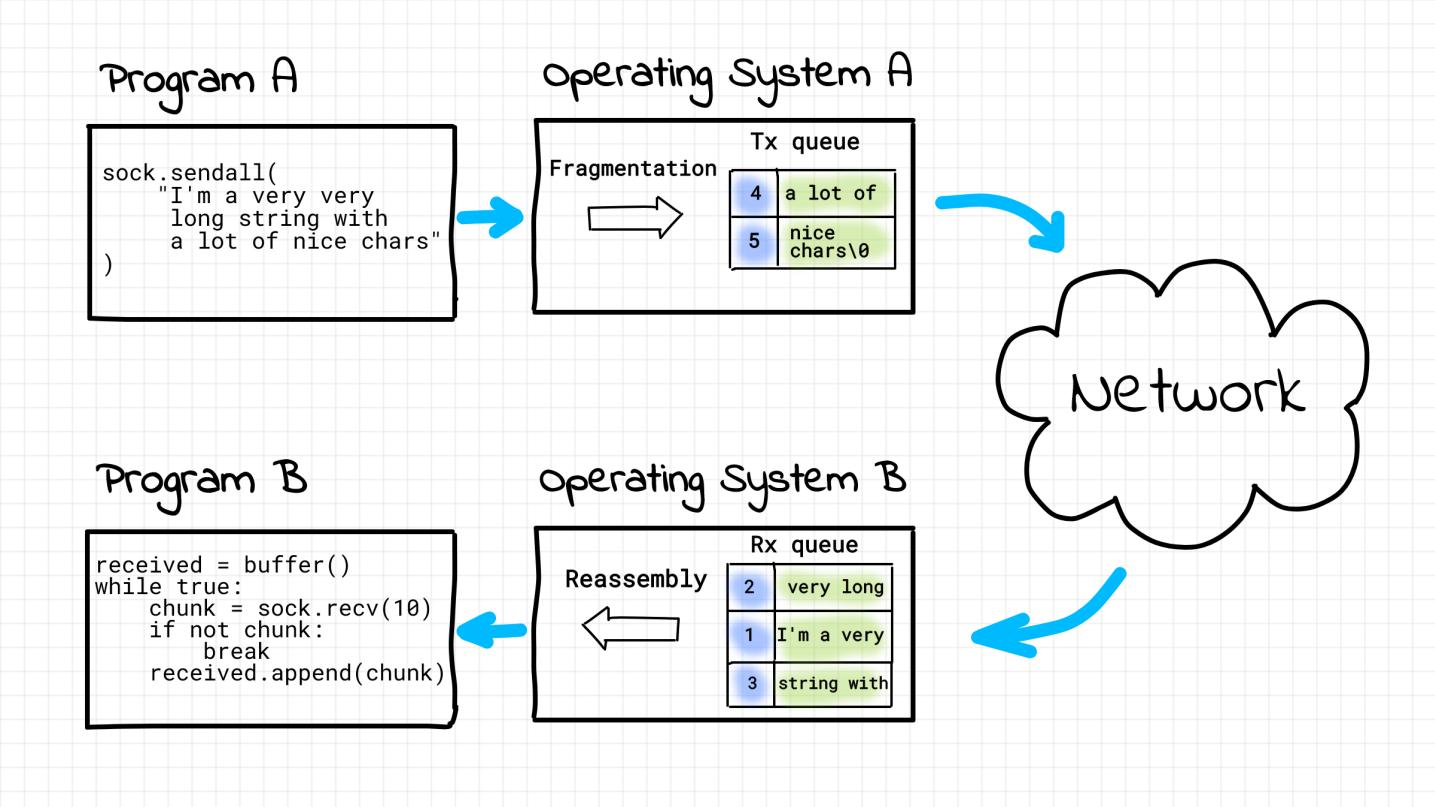
is, well, the actual data the program wants to send over.

More often than not, to address computers in the network, they are assigned so-called IP addresses. The acronym *IP* stands for *Internet Protocol* - a famous protocol that made the **inter**connection of **networks** possible giving birth to the **Internet**. The Internet Protocol is primarily responsible for 3 things:

- addressing host interfaces;
- encapsulating the payload data into packets (i.e. aforementioned *chunking*);
- routing packets from a source to a destination across one or more IP networks.

▶ *Disclaimer on intentional simplification of chunking problem.*

On its way from the origin to the destination, an IP packet usually passes a handful of intermediate hosts. This series of hosts constitutes a *route*. There could be (and usually is) more than one route for an arbitrary (origin, destination) pair. And since multiple routes are possible contemporaneously, it's perfectly fine for IP packets with the same (origin, destination) pair to take different routes. Getting back to the problem of sending a long piece of text over the network, it can happen that *chunks*, i.e. IP packets, the text was split on will take different routes on their way to the destination host. However, different routes may have different delays. On top of that, there is always a probability of packet loss because neither intermediate hosts nor links are fully reliable. Thus, IP packets may arrive at the destination in an altered order.

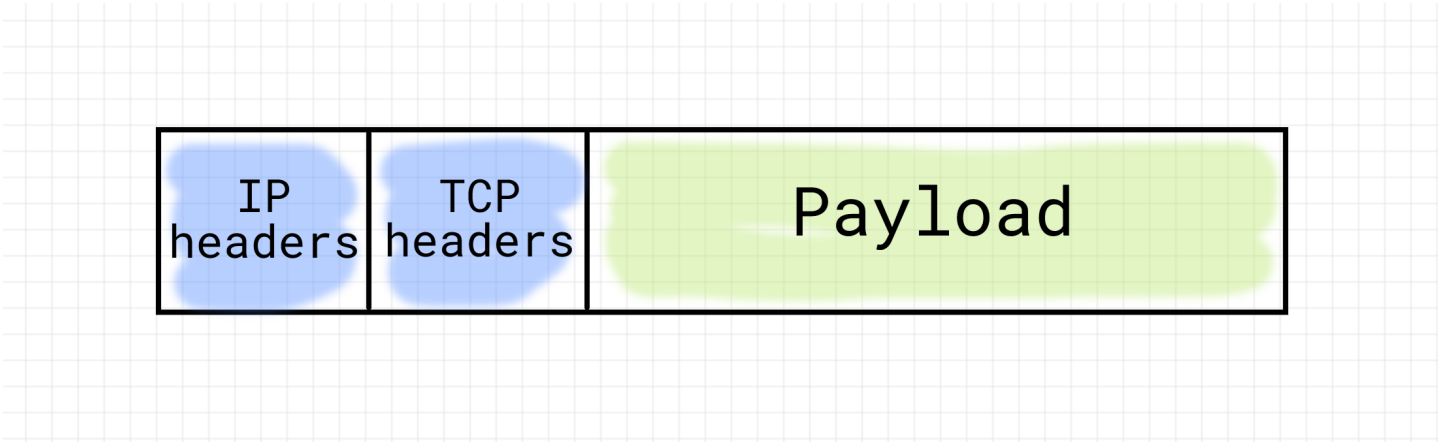


Generally speaking, not every use case requires strict packet ordering. For instance, voice and video traffic are designed to tolerate some amount of packet loss because retransmission of

packets would lead to an unacceptable latency increase. However, when a browser loads a web page using HTTP, we expect letters and words on it to be ordered exactly the same way they were meant to by the page's creator. That's how the need for a reliable, ordered, and error-checked packet delivery mechanism arises.

As you probably already noticed, problems in the networking domain tend to be solved by introducing more and more protocols. And indeed, there is another famous Internet protocol called *Transmission Control Protocol* or simply *TCP*.

TCP is based on its underlying protocol, IP. The primary goal of TCP is to provide the reliable and ordered delivery of a stream of bytes between applications. Thus, if we feed our (encoded) text to a TCP socket on one machine, it can be read unaltered from the socket on the destination machine. Not to be concerned with the packet delivery problems, HTTP relies on the capabilities of TCP.



To achieve the in-order and reliable delivery, TCP augments the control information of every chunk with the auto-increment sequence number and the checksum. On the receiving side, the reassembly of the data happens based not on the packets arrival order, but the TCP sequence number. Additionally, the checksum is used to validate the content of the arriving chunks. Malformed chunks are simply rejected and not acknowledged. The sending side is expected to retransmit the chunks that haven't been acknowledged. Obviously, some sort of buffering is required on both sides to implement this.

On a single machine at any given time there can be many processes communicating via TCP sockets. Thus, there should be as many independent sequence numbers and buffers as the communication sessions. To address this, TCP introduces the concept of *connection*. Simplifying a bit, a *TCP connection* is a sort of agreement between transmitting and receiving sides on the initial sequence numbers and the current state of transmission. A connection needs to be established (by exchanging a few control packets at the very beginning, so-called *handshake*), maintained (some packets need to be sent both ways, otherwise the connection

may time out, so-called *keepalive*), and when the connection is not needed anymore, it should be closed (by exchanging a few other control packets).

Last but not least... An IP address defines a network host as a whole. However, between any two hosts, there might be many simultaneous TCP connections. If the only addressing information in our chunks were the IP addresses, it would be virtually impossible to determine the affiliation of chunks with connections. Thus, some extra addressing information is required. For that, TCP introduces the concept of *ports*. Every connection gets a pair of port numbers (one for the sender, one for the receiver) that uniquely identifies the TCP connection between pair of IPs. Hence, any TCP connection can be fully identified by the following tuple: `(source IP, source port, destination IP, destination port)`.

Implementing simple TCP server

It's practice time! Let's try to create our own tiny TCP server in Python. For that, we'll need the `socket` module from the standard library.

For a novice, the main complication with sockets is the existence of an apparently magical ritual of preparing sockets to work. However, combining the theoretical background from the beginning of this article with the hands-on part of this section should turn the magic into a sequence of meaningful actions.

In the case of TCP, the server- and client-side socket workflows differ. A server passively waits for the clients to connect. A priori, the IP address and TCP port of the server are known to all its potential clients. On the contrary, the server doesn't know addresses of its clients until the moment those connect. I.e, clients play the role of the communication initiators by actively connecting to servers.

However, there is more to it than just that. On the server-side, there is actually two types of sockets involved - the aforementioned server socket waiting for connections and, surprise, surprise - client sockets! For every established connection, there is one more socket created on the server-side, symmetrical to its client-side counterpart. Thus, for *N* connected client, there will always be *N+1* sockets on the server-side.

Create server TCP socket

So, let's create a server socket:

```
# python3
```

```
import socket

serv_sock = socket.socket(
    socket.AF_INET,      # set protocol family to 'Internet' (INET)
    socket.SOCK_STREAM,  # set socket type to 'stream' (i.e. TCP)
    proto=0              # set the default protocol (for TCP it's IP)
)

print(type(serv_sock))   # <class 'socket.socket'>
```

Wait a minute... Where is the promised `int fd = open("/path/to/my/socket")` ? The truth is that the system call `open()` is too limited for socket use case because it doesn't allow to pass all the needed parameters, such as protocol family, socket type, etc. Therefore, for sockets, a dedicated system call `socket()` has been introduced. Similarly to `open()` , after creating an endpoint for communication, it returns a file descriptor that refers to that endpoint. As of the missed `fd = ...` part, Python is an Object-Oriented language. Instead of functions, it tends to use classes and methods. The `socket` module from Python's standard library is actually a thin OO-wrapper around the set of socket-related calls. Oversimplifying, it can be thought of something like that:

```
class socket:  # Yep, the name of the class starts from a lowercase letter...
    def __init__(self, sock_family, sock_type, proto):
        self._fd = system_socket(sock_family, sock_type, proto)

    def write(self, data):
        system_write(self._fd, data)

    def fileno(self):
        return self._fd
```

That is, if someone really needs it, the integer file description can be obtained as follows:

```
print(serv_sock.fileno())  # 3 or some other small integer
```

Bind server socket to network interface

Since in general, a single server machine may have more than one network adapter, there should be a way to *bind* the server socket to a particular interface by assigning a local address of this interface to the socket:

```
# Use '127.0.0.1' to bind to localhost
# Use '0.0.0.0' or '' to bind to ALL network interfaces simultaneously
# Use an actual IP of an interface to bind to a specific address.

serv_sock.bind(('127.0.0.1', 6543))
```

On top of that, `bind()` requires a port to be specified. The server will be waiting or, in networking parlance, *listening* for client connections on that port.

Wait for client connections

Next, the socket needs to be explicitly turned into a *listening* state:

```
serv_sock.listen(10)  # 10 - is a size of the connections queue, so-called bac
```

After this call, the operating system makes the server socket ready to *accept* incoming connections. However, our code is not ready for that yet. But first, let's briefly touch on the backlog part.

As we already know, network communication happens through sending packets. At the same time, TCP requires *establishing* connections. Hence, to establish a TCP connection, a client and a server need to exchange a few control (i.e. w/o any business data) packets (so-called *handshake*). And due to network delays, it's not an instant procedure.

Usually, TCP is implemented on the operating system level, and in our programs, we don't deal with the lower-level details, such as handshake. The backlog parameters define the size of the queue of the established connections. Until the number of connected clients is lower than the backlog size, the operating system will be establishing new connections and queuing them. However, when the number of the established connection reaches the backlog size, any new connections will be explicitly rejected or implicitly ignored (depends on the OS configurations).

Accept client connection

To obtain an established connection from the backlog queue, we need to do the following:

```
client_sock, client_addr = serv_sock.accept()
```

However, the queue of established connections may be empty. In such a case, the `accept()` call will block the execution of the program until the next client connects (or the program is interrupted by a signal, but it's off-topic for this article).

After *accepting* the very first client connection, there will be two sockets on the server-side: the already familiar `serv_sock` in the `LISTEN` state and the new `client_sock` in the `ESTABLISHED` state. Interestingly enough, the `client_sock` on the server-side and the corresponding socket on the client-side are so-called peer endpoints. I.e. they are of the same kind, data can be written into or read from any of them, and they both can be closed using `close()` call efficiently terminating the connection. None of these actions will affect the listening `serv_socket` anyhow.

Get client socket IP and port

Let's take a look at the server and client peer endpoint addresses. Every TCP socket can be identified by two pairs of numbers: `(local IP, local port)` and `(remote IP, remote port)`.

To learn the remote IP and port of the newly connected client, the server can inspect `client_addr` variable returned by the successful `accept()` call:

```
print(client_addr)  # E.g. ('127.0.0.1', 54614)
```

Alternatively, `socket.getpeername()` method of the server-side peer endpoint `client_sock` can be used to learn the *remote* address of the connected client. And to learn the *local* address that the server operating system allocated for the server-side peer endpoint, one can use `socket.getsockname()` method.

In the case of our server it may look something like this:

```
serv_sock:
  laddr (ip=<server_ip>, port=6543)
  raddr (ip=0.0.0.0, port=*)

client_sock:  # peer
  laddr (ip=<client_ip>, port=51573)  # 51573 is a random port assigned by the
  raddr (ip=<server_ip>, port=6543)   # it's a server's listening port
```

Send and receive data over socket

Here is a simple example of receiving some data from the client and then sending it back (so-called echo-server):

```
# echo-server

data = client_sock.recv(2048)
client_sock.send(data)
```

Well, where are the promised `read()` and `write()` calls? While it's possible to use these two calls with socket file descriptors, as with the `socket()` system call, they don't allow to specify all the potential needed options. Therefore, for sockets, `send()` and `recv()` system calls have been introduced. From the `man 2 send`:

> The only difference between send() and write() is the presence of flags. With a zero flags argument, send() is equivalent to write().

...and `man 2 recv`:

> The only difference between recv() and read() is the presence of flags. With a zero flags argument, recv() is generally equivalent to read().

Behind the apparent simplicity of the above snippet there is a serious problem. Both `recv()` and `send()` calls actually work trough so-called network buffers. The call to `recv()` returns as soon as *some* data appears in the buffer on the receiving side. And of course *some* rarely means *all*. Thus, if client wanted to transmit, say 1800 bytes of data, `recv()` may return as soon as the first 1500 bytes are received (numbers are arbitrary in this example) because the transmition got chunked into two portions.

The same is true about the `send()` method. It returns the actual number of bytes that have been written to the buffer. However, if the buffer has less space available than the attempted piece of data, only part of it will be written. So, it's up to the sender to make sure that the rest of the data will be eventually transmitted. Luckily, Python provides a handy `socket.sendall()` helper which does the sending loop for you under the hood.

This actually lead to interesting considerations when it comes to designing data exchange over TCP:

> messages must either be fixed length (yuck), or be delimited (shrug), or indicate how long they are (much better), or end by shutting down the connection.

Detect client is done with sending (shutdown)

Notice, that the first three options still may lead to a situation when the socket on the server-side will be waiting for `recv()` call to return indefinitely long. It can happen if the server would want to receive K messages from the client while the client would want to send only M messages, where M < K. Thus, it's up to the higher-level protocol designers to decide on the communication rules.

However, there is a simple way to indicate that the client is done with sending. The client socket can `shutdown(how)` the connection specifying `SHUT_WR` for `how`. This will lead to a `recv()` call on the server-side returning 0 bytes. Thus, we can rewrite the receiving code as follows:

```
chunks = []
while True:
    data = client_sock.recv(2048)
    if not data:
        break
    chunks.append(data)
```

Close sockets

When we are done with a socket, it should be closed:

```
socket.close()
```

Closing a socket explicitly will lead to flushing its buffers and shutting down the connection gracefully.

Simple TCP server example

Finally, here is the full code of the TCP echo-server:

```
# python3

import socket

# Create server socket.
serv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto=0)

# Bind server socket to loopback network interface.
serv_sock.bind(('127.0.0.1', 6543))

# Turn server socket into listening mode.
```

```
serv_sock.listen(10)

while True:
    # Accept new connections in an infinite loop.
    client_sock, client_addr = serv_sock.accept()
    print('New connection from', client_addr)

    chunks = []
    while True:
        # Keep reading while the client is writing.
        data = client_sock.recv(2048)
        if not data:
            # Client is done with sending.
            break
        chunks.append(data)

    client_sock.sendall(b''.join(chunks))
    client_sock.close()
```

Save it to `server.py` and run via `python3 server.py`.

Implementing simple TCP client

Things are much simpler on the client-side. There is no such thing as a listening socket on the client-side. We just need to create a single socket endpoint and `connect()` it to the server before sending some data:

```
# python3

import socket

# Create client socket.
client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to server (replace 127.0.0.1 with the real server IP).
client_sock.connect(('127.0.0.1', 6543))

# Send some data to server.
client_sock.sendall(b'Hello, world')
client_sock.shutdown(socket.SHUT_WR)

# Receive some data back.
chunks = []
while True:
    data = client_sock.recv(2048)
    if not data:
        break
```

```
    chunks.append(data)
print('Received', repr(b''.join(chunks)))

# Disconnect from server.
client_sock.close()
```

Save it to `client.py` and run via `python3 client.py`.

Socket server vs HTTP server

The server we implemented above is clearly was a simple TCP server. However, it's not a web server (yet). While (almost?) every web server is a TCP server, not every TCP server is a web server of course. To turn this server into a web server, we would need to teach it how to deal with HTTP. I.e. the data transmitted via sockets should be formatted in accordance with the set of rules defined by the Hypertext Transfer Protocol and our code should know how to parse it.

Wrapping up

Memorizing stuff without understanding it is a poor strategy for a developer. Sockets programming is a perfect example when looking at the code without the theoretical background can be simply overwhelming. However, once the understanding of the moving parts and constraints is gained, all these magical manipulations with the socket API turn into a meaningful set of actions. And don't be afraid of spending time on basics. Network programming is a fundamental knowledge that is vital for the successful development and troubleshooting of advanced web services.

Further reading

- ❤️ Socket Programming HOWTO on docs.python.org.
- Beej's Guide to Network Programming - basic socket programming in C.
- UNIX Network Programming - advanced socket programming.

server, TCP

Written by Ivan Velichko

Follow me on twitter @iximiuz

Liked this article? Let it be the beginning of a great friendship. Leave your email so I could notify you about new articles or any other interesting happenings around the topics of this blog. No spam whatsoever, I promise!

Email address

Subscribe!

What do you think?

6 Responses

 Upvote  Funny  Love  Surprised  Angry  Sad