Distributed Storage Lab - Week 1

Written by: Marcell Feher <sw0rdf1sh@eng.au.dk>

Prerequisites

- Python 3 is installed on your computer (preferably 3.6 or newer)
- You have a basic understanding of Python and you are able to write Python code. It's okay if you need to look things up sometimes
- You are familiar with the material of Lecture 1
- Recommended reading before the lab: Python socket programming howto

Goals

In this lab, we will learn how to use **sockets** in Python to send messages and files between a simple client and a server, locally. We will extend the example TCP server that was introduced in the lecture, and implement the following functionality:

- Socket Server and Client: send text messages
    - Both on same machine via localhost
    - Client sends message, e.g., name
    - Server receives a message, replies: "I have received: " and the message. Ends connection
    - Server reports IP and message in the terminal
    - Test with colleagues: Use your client to send data to their server

- Socket Server and Client: File transmission
    - Instead of a message, transmit contents of a file and save as file in the server

Sockets in Python

Sockets represent network connections between computers and provide a relatively simple programming model to send and receive data. Computers usually have several hardware network interfaces where they can communicate, for example WiFi, LAN, Bluetooth, etc. These expose their connectivity to the operating system via abstract network interfaces, usually multiple per physical device. You can list these with the **`ifconfig`** terminal command on Linux and macOS, and **`ipconfig`** in Windows. Each network interface can have multiple TCP and UDP connections on several numbered ports, ranging from 1-65535. Ports 1-1023 are reserved for well-known services like HTTP or FTP, the higher ones can be used freely for arbitrary applications. When a connection is established, a *queue* (or *buffer*, we will use these phrases

interchangeably) is created in the memory where the incoming and outgoing packets are stored temporarily.

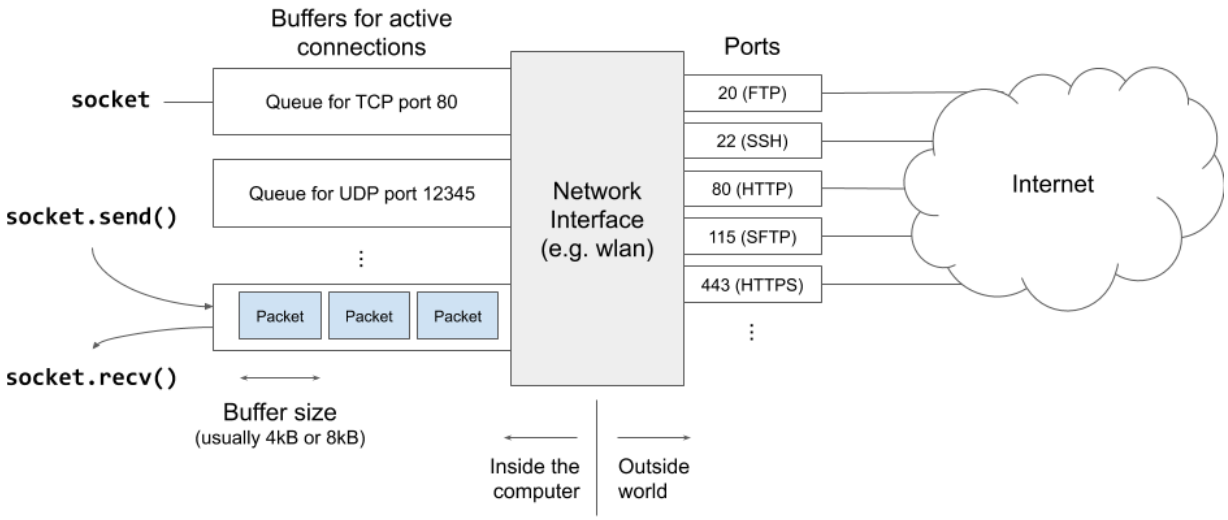

Fig 1: Simplified visualization of a socket

The **`socket`** you work with in Python and other languages gives you a handle to this queue, allowing you to read and write data packets. The size of packets is fixed by the network card, usually 4 or 8 kilobytes (depends on the hardware design of the card). You cannot control this size and it is independent from the type or port of the connection, as well as the service using it, or the remote party connecting to it. What you *can* control is how many bytes you read from the queue with **`socket.recv()`** or write to it with **`socket.send()`**. To make your socket program efficient, a good practice is to read and write in units of 4kB (4096 bytes).

The biggest challenge in working with sockets comes from the low level interface that handles raw byte packets, and it doesn't know (or care) about what they mean. Furthermore, sockets don't necessarily handle every byte that you write to them or try to read from them. Since a socket is just a manager for a memory buffer, its main focus is to keep the buffer healthy, which practically means not full. The **`send`** and **`recv`** methods return when the buffer is full (**`send`**) or empty (**`recv`**), and from the response you can figure out how many bytes were actually sent or received. It is the programmer's responsibility to call the socket again until every byte of the message is processed.

On top of this, you are responsible for designing a message format, figuring out how to signal message size and the data structure encoded in the byte stream. Since a socket is not a request-response protocol (like HTTP, which builds on top of raw TCP sockets), there is no

built-in “end of transmission” signal either. It just stays open until the connection is closed by either party, or eventually if there’s no response (e.g. the remote party crashed).

There are two ways to overcome this problem:

1. You can use each socket connection to send exactly one message, and close the connection right afterwards. When **`recv`** returns zero bytes, you can be sure that the remote party closed the connection and there won’t be any other data arriving on this socket. This is how HTTP works.
2. Alternatively, you can design your own message scheme that allows the receiver to figure out when the complete message has been received, so it can handle it, and keep using the same socket. This is how Netflix, Zoom, Skype and similar services work, where the data flow is continuous and might even be fully duplex (sending and receiving at the same time). Alternatively, you can use constant size messages, but this is highly inefficient in most real use cases.

Keep in mind that a socket-based protocol doesn’t have to be strictly request-response based. Sockets allow you to send and receive at any given time. Of course, this makes things a lot more complicated, especially if the outgoing stream somehow depends on the incoming stream. In this course we will only use request-response type protocols, which are more straightforward to understand and implement correctly.

To illustrate all this, let’s examine the following code snippet (source)

```
# Fixed message length, actual value depends on the protocol
MSGLEN = 1234

class MySocket:
    """
    Demonstration class only - coded for clarity, not efficiency
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
```

```
                raise RuntimeError("Socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == b'':
                raise RuntimeError("Socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)
```

This socket protocol uses fixed size messages (**MSGLEN**) and keeps the connection open, which makes the receiving code really simple. Notice how both the sending and receiving methods use a while loop to write and read the socket buffer until the whole message has been dealt with.

The **mysend** method tries sending the whole message at once, but also handles the case when not all of it has been sent successfully by updating the **totalsent** byte counter. In the next iteration, it sends the remainder of the message and adjusts the counter again, until the whole message is out of the door (**send()** returns zero, the buffer is empty).

The **myreceive** class method knows exactly the message size (**MSGLEN**), but it still reads the buffer in 2kB chunks and reconstructs the message iteratively. If **recv()** returns nothing, the receiver knows for sure that the connection is broken and this socket is no longer usable. Notice that **myreceive** does not assume that the received bytes will always be 2kB until the last fragment. Instead, it just appends whatever arrived to the **chunks** list. This follows the philosophy of sockets: you can ask the socket to process (send or receive) a number of bytes, but need to check the result and be prepared when not all of them have been handled.

You might be wondering, how does **myreceive** know that it didn't receive a valid "empty message". Simple: there are no empty messages in a socket connection. This is a low-level API, just a dumb pipe between two communicating parties. Until the pipe is empty, **recv** blocks program execution. When something is sent through it, execution continues and **recv** returns it (well, actually just a buffer-size first part of it - the caller has to call **recv** again for the rest). When it returns zero bytes, that's an indication to the program that the other party is gone.

A more complicated protocol that doesn't close the socket after each transmission and has variable-size messages must come up with its own system of signalling message lengths. The complication comes from the fact that sometimes you don't know in advance how many bytes will be necessary to represent the message size that follows. If you design too many, then you're wasting bandwidth. If too few, your message overflows and the protocol is completely ruined. In practice, a good message schema can be the following:

- The first byte is a message type identifier. The receiver should know what kind of message follows, whether it's fixed or variable length, or the remote party will close the connection afterwards. If you have more than 256 message types, you'll need 2 or more bytes to encode it.
- If the message is fixed length, read as many bytes as expected and treat this as one message. You probably have leftover bytes after the message that you already read out from the buffer, these must be stored and parsed as the beginning of a new message. Keep in mind, if you have multi-byte message types, you might not have all bytes in the leftover buffer to parse it. Yeah, sockets suck.
  Fixed length messages are ideal for fixed size data types like booleans, integers or floating point values, or structs composed purely from these types.
- When the message type suggests a variable-length message, the next few bytes should encode the length. Typically the message type determines how many bytes this is, but you can also use fixed length for this (e.g. 4 bytes if you know for sure the message will never be longer than 65kB).
  When you want to send a string or your data structure has strings in it, and you know it will never be longer than a maximum size, this message type is ideal. It's also okay if you are dealing with files of a reasonable size
- If the remote party will close the connection, you just read until `recv` returns zero bytes. This method works well for sending files in a large range of sizes.

Tasks

We start with simple, local Python server and client programs, where the client sends TCP messages to the server, all within your local computer. Then, we extend this in multiple directions: starting a new thread on the server to handle each incoming request, reading (and later parsing) the request contents, and finally, uploading binary files to the server.

These tasks are meant to get you started with Python sockets and give you a sense of the challenges and problems that we will solve with higher level tools later.

If you are stuck and cannot find the solution yourself, post a question on the Blackboard Discussion Board!

Task 1: Simple server and client

Implement a socket server in Python, that listens on TCP port 9000 locally for incoming connections. When a connection is established, print the remote address to the terminal window and close the socket.
Write a simple client program that connects to the server and print a status message to the terminal every time when the user hits the Enter key.

Both the server and client should run indefinitely until the user terminates the program manually (Ctrl+C).

Notice that the server does not do anything with the received data, and does not send anything back.

Test the communication by hitting Enter in the client a few times, and verify that you see the intended console messages in both terminal windows.

Task 2: Multithreaded server

Modify the server code to start a new background thread for each new incoming connection, and handle it there.

> **Note**
> Python doesn't support actual multithreading, where threads are executed on different CPU cores in parallel. The **`threading`** python package overlaps execution of the logical threads in a single physical thread. Real parallel execution can be implemented using the **`multiprocessing`** package, which starts a new process and runs a piece of code there. However, a new process has a lot higher overhead than a new thread. So much, that it's not acceptable when serving small HTTP requests.

Task 3: Welcome message

Extend the client to read a string from the standard input, and send that to the server. Limit the string size to 4kB (just drop the rest).
Extend the server to convert the received bytes to string and write it to the console, together with the remote IP address. The server should respond by "Message: {received message}".

Task 4: Different message types

Define a message format where you have 2 message types.

1. Send a short variable length string, up to 4 kB
2. Send arbitrary size binary data (e.g. file contents).

The client should first ask the user which type of message they want to send. If string type was selected, read the string from the input and send it to the server in a #1 type message. If binary type was selected, generate a random byte array and send that to the server as a #2 type message. The client should write what happened to the console. Keep doing this until the user exits the program (e.g. after sending the message, the client should again ask the user what message type to send). Close the connection after each message.

Extend the server to parse the received message, and handle it according to the message type. If a string was received, print it to the console. If binary data was sent by the client, store it as a file with a random generated filename.

Hints

To send an integer like the message type via a socket, it must be converted to a Python **`bytes`** object (the only type that **`socket.send()`** can send). There is a converter function for this: int.to_bytes(). Since both the client and the server are on the same machine, it's best to use the system's default endianness as the byteorder parameter: **`sys.byteorder`** (need to import the **`sys`** package to access this). When receiving an integer on the server side, the bytes object needs to be converted back to an integer using the int.from_bytes() function.

```
import sys

# Client: convert int to bytes
MESSAGE_TYPE_STRING = 1
...
MESSAGE_TYPE_STRING.to_bytes(length=1, byteorder=sys.byteorder)

# Server: convert bytes to int
message_type_bytes = clientsocket.recv(1)
message_type = int.from_bytes(message_type_bytes, byteorder=sys.byteorder)
```

To save a **`bytes`** object as a file, you can use the following function, which also takes care of generating a random filename if nothing is given.

```
import random, string

def write_file(data, filename=None):
    """
    Write the given data to a local file with the given filename
    :param data: A bytes object that stores the file contents
    :param filename: The file name. If not given, a random string is generated
    :return: The file name of the newly written file, or None if there was an error
    """
    if not filename:
        # Generate random filename
        filename_length = 8
        filename = ''.join([random.SystemRandom().choice(string.ascii_letters +
string.digits) for n in range(filename_length)])
        # Add '.bin' extension
        filename += ".bin"

    try:
        # Open filename for writing binary content ('wb')
        # note: when a file is opened using the 'with' statement,
```

```
        # it is closed automatically when the scope ends
        with open('./'+filename, 'wb') as f:
            f.write(data)
    except EnvironmentError as e:
        print("Error writing file: {}".format(e))
        return None

    return filename
```

Task 5: Send data size

Extend message type #2 to include the data size before sending the data itself. You can choose a number of bytes to encode this information, and make sure the client never tries to send data more than what can be encoded on this many bytes. Alternatively, you can introduce different message types that define the data size bytes. For example:

```
# String message
MESSAGE_STRING  = 1

# Data message, data size is encoded on 1 byte (data size: up to 255 bytes)
MESSAGE_DATA_1B = 2

# Data message, data size is encoded on 2 bytes (data size: up to 64 kB)
MESSAGE_DATA_2B = 3

# Data message, data size is encoded on 3 bytes (data size: up to 16 MB)
MESSAGE_DATA_3B = 4

...
```

The server should verify that every byte was received before saving the file.

Task 6: Upload a file

Copy a few test files to the directory where the client is. The client reads a filename string (e.g. “test.pdf”) from the user in each iteration of the main while loop. When it’s given, the client tries to read the file from its local folder into the memory. If the file exists, the client first sends the file name to the server as a #1 type message, and waits for a response. If the response is the string “OK”, the client sends the file contents as a #2 type message (including data size) and close the connection. The server writes the file contents to the local folder, using the given file name. Both the client and server must handle files over 4kB correctly (send and receive multiple chunks).

Extra task 1: After a file upload is complete, keep the socket connection open and re-use the same socket for additional uploads.

Extra task 2: Create the file on the server immediately when you receive the file name, and append to the contents every time you receive data. This way you are not allocating memory for the whole file on the server until all fragments arrive.