Network Sockets 1

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

Today it seems like just about any application we write involves some degree of network interaction. Our processes communicate with each other using *network sockets*.
These sockets are really just one (very common) form of interprocess communication. The general principles we apply in this case apply across interprocess communiction in general.

Network model

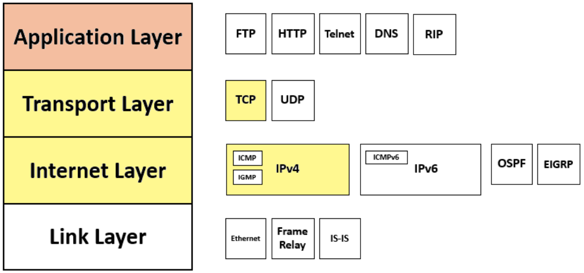

image: Michel Bakni (`https://www.wikidata.org/wiki/Q81411358`)

Servers and clients

We will discuss writing both client and server applications. For our purposes

- A *server* application listens for incoming requests and responds to them;
- A *client* initiates communication by opening a connection to a server.

A particular application may act like a server at some times and like a client at other times.

Network socket flow

1. The server creates a socket.
2. The server binds the socket to an address/port.
3. The server listens for connections to the socket.
4. The client creates a socket.
5. The client connects to the server's socket.
6. the server accepts the client connection.
7. The client and server send and receive data.
8. The client closes its connection, notifying the server.
9. The server closes its connection.

In code

```
# Server:
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # 1
s.bind(('127.0.0.1', 65432))                           # 2
s.listen()                                             # 3
conn, addr = s.accept()                                # 6
data = conn.recv(1024)                                 # 7.2
conn.sendall(b'received')                              # 7.3
data = conn.recv(1024)                                 # 8.2
conn.close()                                           # 9

# Client
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # 4
s.connect(('127.0.0.1', 65432))                        # 5
s.sendall(b'Hello')                                    # 7.1
data = s.recv(1024)                                    # 7.4
s.close()                                              # 8.1
```

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `19-practical` in your practicals repo.
3. Copy the subdirectory, `19-practical` from the class materials into your repo.
4. See the README for directions.
5. We will discuss results in 30ish minutes.

A problem

Our server can only handle one connection at a time. While it's servicing that connection, other clients just queue up and wait.

In particular, calls to `accept()` and `recv()` block waiting on data from the socket.

Even when we solve this problem, our application code needs some sort of concurrency mechanism to handle multiple clients.

Concurrency

We will cover concurrency in more detail next week, but briefly our options are

- Threading
- Forking
- Asyncio
- Select

We will use select today.

Server: setting up

```
import socket
import selectors
...
clients = selectors.DefaultSelector()
listener = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
listener.bind((host, port))
listener.listen()
listener.setblocking(False)
clients.register(listener, selectors.EVENT_READ, data=None)
```

Two things are going on here:

1. We use `setblocking(False)` so that, when we call `accept()` later, our code won't block waiting on a connection.
2. We register our listener with our selector, telling the selector to notify us if the listener receives data.

Server: event loop

Now we basically just loop, waiting for the selector to notify us when a client socket has some data to process.

```
while True:
  events = clients.select(timeout=None)
  for event, mask in events:
      if event.data is None:
          accept_connection(event.fileobj, clients)
      else:
          service_connection(event, mask, clients)
```

When we registered the listener, we set `data=None`. This is why `event.data` is `None` for a new connection.

Server: accept connection

```
def accept_connection(sock, clients):
    conn, addr = sock.accept()
    conn.setblocking(False)
    data = {'addr': addr, 'buffer': b''}
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    clients.register(conn, events, data=data)
```

Again, notice that we set the connection to be nonblocking once we establish it. We register the newly established connection with our `clients` selector.

Server: service connection

```
def service_connection(event, mask, clients):
    sock = event.fileobj
    data = event.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)
        if recv_data:
            data['buffer'] = recv_data
        else:
            client.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data['buffer']:
            sent = sock.sendall(data['buffer'])
            data['buffer'] = b''
```

If we received data, we save it in the buffer to send back to the client later. If we're ready to write to client and we have data in buffer, we send it.

References

- `socket` library:
  `https://docs.python.org/3/library/unittest.html`
- `selectors` library:
  `https://docs.python.org/3/library/selectors.html`
- RealPython article on socket programming:
  `https://realpython.com/python-sockets/`