Network Sockets 2

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

Last time we saw how to get network sockets working from the client and the server side. Our basic implementations had a lot of problems. One of them was that our server could only respond to one client at a time. We overcame that by using selectors to multiplex over our client sockets.

This time we will see how to overcome another big problem.

Network model

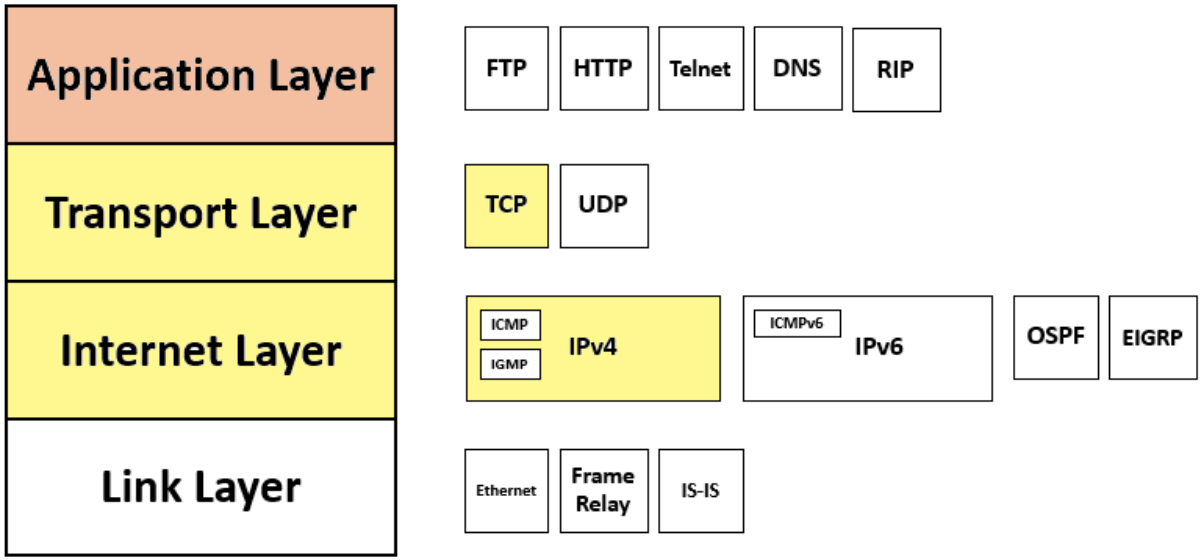


image: Michel Bakni (`https://www.wikidata.org/wiki/Q81411358`)

The problem

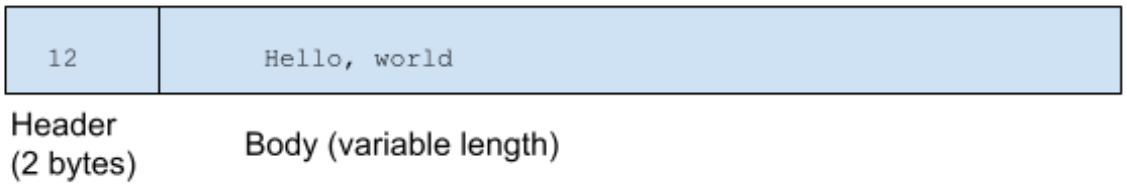
```
# Server:
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('127.0.0.1', 65432))
s.listen()
conn, addr = s.accept()
data = conn.recv(1024)
```

What if we want to send and receive longer messages?

Two options

1. Require that our application uses fixed-length messages.
2. Make the messages indicate how long they are.

A simple header



Warning

We're heading into territory where we're manipulating things byte-by-byte. None of this is rocket science, but there are a lot of details to worry about.

Message: send

```
import struct

def send(txt, sock):
  msgbody = bytes(txt.encode('utf-8'))
  msglen = len(msgbody)
  header = struct.pack('>H', msglen) # 2-byte unsigned int
  message = header + msgbody
  sock.sendall(message)
```

Message: receive

```
def receive(sock):
    data = b''
    while len(data) < 2:
        data = sock.recv(4)
    body_length  = struct.unpack('>H', data[:2])[0]
    print(body_length)
    data = data[2:]
    while len(data) < body_length:
        data += sock.recv(4)
    return data.decode('utf-8')
```

(Yes, calling `sock.recv(4)` is a bit silly, but it demonstrates our process with small messages.)

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `20-practical` in your practicals repo.
3. Copy the subdirectory, `20-practical` from the class materials into your repo.
4. See the README for directions.
5. We will discuss results in 30ish minutes.

More Cowbell

We now have a de facto “application protocol” that lets us send and receive utf-8 strings of up to about 64kb length. We can do better.

If we could define a larger header with multiple fields, then we could attach more metadata to our application messages.

Header format

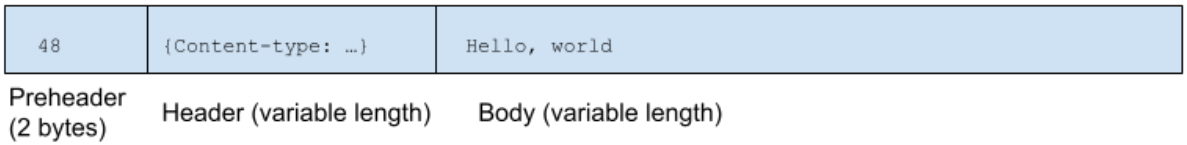
It's our application, so we can do anything we want with the header. We probably want it to be text. Using JSON makes it easy to parse.

```
{
 'Context-type': 'text/plain',
 'Content-encoding': 'utf-8',
 'Content-length': 46
}
```

We put a header like this before our body. An application can use this to determine how to process the body.

A full header

Now we have a new problem. How long is the header? But we already solved this.



The preheader tells us how long the header is. The header tells us how long the body is.

Writing a message

The process of writing a message is

1. Get your message body and convert it to bytes.
2. Prepare your header, now that you know how long the body is, and convert it to bytes.
3. Prepare your preheader with the length of your header.
4. Assemble the three parts.
5. Send the message.

Reading a message

The process of reading a message is

1. Get at least two bytes in an input buffer.
2. Process the preheader and remove it from the buffer.
3. Collect enough bytes in the buffer to hold the header.
4. Process the header and remove its bytes from the buffer.
5. Collect the bytes for the body and process it.
6. Send the message.

At this point we probably have enough structure to warrant a `Message` class.