Sockets: Overview

Matt Ruffalo
Adapted from content by Stuart Morgan, 2003

April 9, 2013

Case Western Reserve University EST. 1826

Sockets from 10,000 feet

- Overall idea: provide a reasonably high-level method of communicating across an arbitrary networks (LAN, Internet, etc.) without worrying about hardware.
- Lets you use TCP or UDP (more on those later) with the same API
  - A few ugly API details, but in the abstract they are very straightforward and uniform.

ISO/OSI Networking Model

7 layers

7 Application
6 Presentation
5 Session
4 Transport
3 Network
2 Data Link
1 Physical

Useful in theory, but network communication doesn't actually work this way

ISO/OSI Networking Model

7 layers

- In principle, a layer only has to deal with the one directly below it
- Transport layer relies on network layer for host addressing
- Network layer uses data link layer to communicate over physical media
- Data link layer doesn't care about sending voltage signals over a wire

ISO/OSI Networking Model

7 layers

- Real Internet isn't implemented this way
- Top three layers are blended together
- Unix socket communication deals with the third and fourth layers (transport and network)

A few more details

Connections or not?

- There are two models of socket programming: connection-oriented (TCP) and connectionless (UDP).
- Each has strengths in different applications
- Sockets are identified the same in both protocols: (IP, port)
- Different services use different ports by convention
  - HTTP: 80, SMTP: 25, SSH: 22, ...
- Sockets are "bound" to port numbers on creation

Connection-oriented paradigm

Server	Client
Create a socket with `socket`	Create a socket with `socket`
Bind the socket to a port with `bind`	
Set up a connection queue with `listen`	
Establish a connection with `accept`	Request a connection with `connect`
Read and write data with `read` and `write`	Read and write data with `read` and `write`

Connectionless paradigm

Server	Client
Create a socket with `socket`	Create a socket with`socket`
Bind the socket to a port with `bind`	Bind the socket to a port with `bind`
Send and receive data with `recvfrom` and `sendto`	Send and receive data with `recvfrom` and `sendto`

A few things to note:

- The commands have intuitive names. It's fairly clear what each of these commands does, which makes the models fairly easy to use.
  - Parameters are *not* so straightforward
- Not a C-specific model. Although the details of these commands certainly differ from language to language, the overview above is fairly constant from language to language.
  - Python and Java work in almost exactly the same way.

Main difference: reliability of data stream

Say you are standing in the Strosacker balcony, and your friend is on the stage, and you want to pass notes (don't ask why you can't just shout, it's an analogy)

- To simulate TCP, you get a long string, and throw one end down to your friend. Each of you holds one end tight, and to pass a note down you punch a hole in it, and thread it onto the string, where gravity pulls it down to your friend, who will get every note, all in the order that you sent them.

Main difference: reliability of data stream

Say you are standing in the Strosacker balcony, and your friend is on the stage, and you want to pass notes (don't ask why you can't just shout, it's an analogy)

- For UDP, you write all your messages on paper airplanes and throw them to your friend. Now, most will get there (assuming you make good airplanes), but occasionally, some will go off in a random direction, or crash right away, so your friend never gets them. What's more, some of them may do a lot of looping and hovering, and take a long time to arrive, while others will go directly, so there's no guarantee they will get there in the order you sent them.

Server

```
main {
  srv = socket()
  bind(srv, 80)
  listen(srv)
  while (1) {
    clnt = accept(srv)
    if (fork() == 0) {
      serveRequest(clnt)
      exit()
    }
  }
}
```

```
void serveRequest(clnt) {
  request=read(clnt)
  if (request starts with "GET")
    write(clnt, "response")
  close(clnt)
}
```

Client

```
main {
  client = socket()
  connect(client, hostname, 80)
  write(client, "GET /index.html HTTP/1.0")
  read(client, response)
  printf(response)
  close(client)
}
```