Supplement: IPC with UNIX-Domain-Sockets

Leonard König, April 12, 2021

Server-Client Principle & Protocols

A common principle in system architecture is splitting functionality into a server and a client. The server provides some functionality to the clients that connect to it. This pattern can be used to let processes communicate on the same computer as well as over a network.[1] The *critical* part is the communication: How does the client know 'where' the server is? What 'language' do both parties speak?

[1] A rather user-visible example is a web (or HTTP) server with the client being the browser you use to access it.
But Microkernels also embrace this idea to provide different functionality as small server processes in userspace that are orchestrated by the kernel proper.

These questions are answered by protocols. But just like the applications themselves, the protocols aren't monolithic but modular as well. We can have a protocol that handles the address resolution (think: Calling Alice who has Bobs number and ask her for his), the addressing itself (Alices & Bobs number) and one to actually make the connection (dialing, picking up). We can even have additional protocols that are less set in stone, like saying 'Hello, Alice/Bob here' when picking up the phone.

The protocol stack of the Internet would consist of IP (the number) and TCP (calling / picking up) with DNS (Alice in this case) and eg. HTTP sitting 'above'. Below IP there are other layers which define the physical transport from one networked device to another.

Connection-oriented vs. Connection-less

Furthermore we can broadly distinguish server/client systems into connection-oriented and connection-less protocols. While connection-oriented protocols compare to phonecalls above, connection-less protocols can be viewed as packages sent by post: They don't need a stable connection, but they don't necessarily arrive on time, in order or at all. However, and this is where the analogy is unrealistic, they might be faster delivering the information.[2] Additionally, connection-oriented systems are blocking by default, that is, only one client at a time is supported (line busy).

[2] Well, it *is* probably faster to pack three books into a packet and send it rather than reading them over telephone and transcribing them on the other end.

The addressing protocol IP supports both connection-oriented and connection-less protocols on top. Can you name an example for each?

UNIX Domain Sockets

We can use a server/client architecture for IPC between different processes on the same system. In UNIX, the server creates a special file (a socket) that doesn't really have 'content' on the disk. The client then specifies the filename or path to the servers socket (the servers 'phone number') in order to connect. Both processes can then read and write arbitrarily to the 'file', and each would receive the other partys messages.

In this exercise you should create a connection-oriented server which accepts a connection over UNIX domain sockets by one client at a time. The client can enter messages which will be displayed at the servers side.

```
$ ./server server.socket
Server is listening on 'server.socket' ...
```

```
$ ./client server.socket
Client connecting to 'server.socket' ...
Your Message:
```

Listing 1 UNIX Server & Client

Berkeley Sockets API

To program virtually any server and client program on UNIX we use the Berkeley Sockets API. This API allows for different protocols 'below'. We will now go through the steps you need to setup a server and a client with a connection-oriented approach.

Both, server and client create one socket signifying the endpoints of the connection. To create a socket we use the function **socket(3)** which takes three arguments, the domain (address family, eg. `AF_INET` for IPv4 or `AF_UNIX` for UNIX), the type (`SOCK_STREAM` for connection-oriented or `SOCK_DGRAM` for connection-less) and the addressing protocol family, which is usually the same value as the address family and can be set to `0` in that case.[3] After server and client created a socket (which, in the API, is simply a file descriptor stored as an `int`), the code is different in the server and client program.

[3] Actually, the appropriate value for the `AF_UNIX` domain would be `PF_UNIX`, but both constants are often equal. Just using the default of `0` for the protocol is best-practice anyhow.

The server needs to bind the abstract socket to a concrete address (`struct sockaddr`) with the appropriate value (a filename in our case) and call **bind(3)**. Afterwards the server needs to start to **listen(3)** on our socket. Here we also need to specify a number of additional allowed connections to the same socket while we are busy talking to one connection, the so-called backlog.

The clients equivalent to bind & listen is initiating a connection. In a similar way as we bound the socket to an address for the server, we now create `struct sockaddr` with the same content for the client to **connect(3)** to.

Now that the connection is established, both server and client can simply **read(3)** and **write(3)** to their respective sockets to receive or send data. You cannot use `fread`, `fwrite` or even `fprintf` here but need to use these low-level interfaces we worked with in the first exercise.

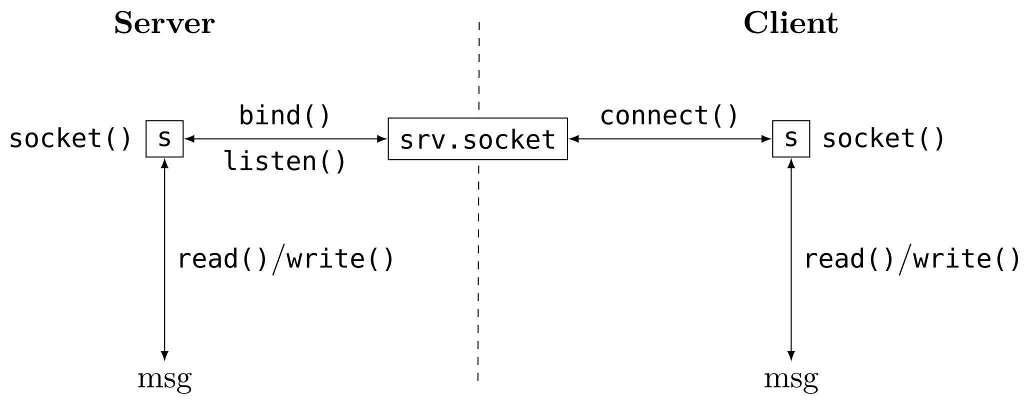

Figure 2 UNIX IPC with Berkeley Sockets

To close the connection you can simply **close(3)** the socket/filedescriptor.[4]

[4] Actually you could keep the fd and only call `shutdown` but `close` does that internally for us already.