Supplement: Connecting Processes over TCP

Leonard König, April 12, 2021

From UNIX IPC to TCP/IP

The novel concept of using sockets for communication between processes on the same system, as we did with UNIX Domain Sockets in the last exercise, can be easily adapted to enable communication between processes *across* computers: Within a computer network.

There are many ways computers can communicate with each other but in this exercise we will restrict ourselves to the TCP/IP protocol stack. From a programming point of view, this is an evolutionary change: Virtually the only difference is that our socket is not backed anymore by a 'file' on the disk[1] representing the 'address' via its path (but it is still accessed via a file descriptor!). Instead the server needs to know computers IP within the network and allocate a Port number to listen on. The IP describes the computer itself, the port determines the specific service/server on the computer (as one computer can have multiple servers running).

[1] This also implies you don't need `unlink(3)` anymore!

Thus the invocation of our server program on the command line changes from passing the socket name to passing IP[2] & port:

[2] We pass the IP because the computer might be connected to multiple networks simultaneously, each giving it a different IP! More on that later.

```
$ ./server <address> <port>
```

The client is adapted analogously.

IP Addresses

The IPv4 and IPv6 protocols specify the format and semantics of addresses, examples include:

- `192.168.1.`x or `192.168.100.`x: common home network IPv4
- `160.45.112.11`: IPv4 of andorra at time of writing
- `8.8.8.8`: Google IPv4 DNS
- `127.0.0.1` and `::1`: IPv4/IPv6 localhost on loopback network

In general, even a device that's not connected to any real network has a special 'loopback' network device that... loops back to itself. Say your laptop is logged in to the universities wireless network and simultaneously connected to a different network via LAN as seen in **Figure 1**.

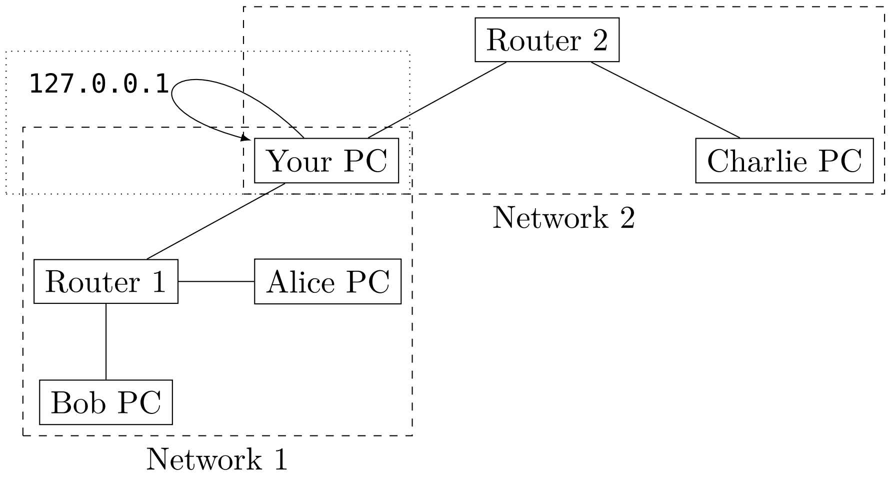

Figure 1 A PC in 3 IPv4 networks (incl. loopback)

It will have three IP addresses: One for the university network (e.g. Network 1), one for the other wired connected network (e.g. Network 2) and another for the pseudo loopback network. While the other two IPs are usually assigned via the router (e.g. using DHCP), the localhost IPv4 is always `127.0.0.1`.[3] This enables us to even use TCP/IP for communication between processes on the same system, as well as using it for remote systems.

[3] There are other ways to construct networks and assign IPs, you don't even really need a router.

It is crucial to understand the IP is not unique for one system, but only unique *within* a single network. If a process on Alice PC wants to connect to a process on Your PC it will need the IP that assigned to your PC within the Network 1. This might be the same or a different IP than the IP assigned to Your PC by Router 2 in Network 2![4]

[4] One could ask: But what if Alice and Charlie have the same IP, how do we distinguish those? For that (and other cases) we need routing information, e.g. type `ip route` on your Linux shell to see your current setup (don't be afraid if the output is confusing).

In this exercise you can test the easiest by simply using localhost `127.0.0.1` as the IP of your device and some port number above or equal to 1024. If you have two devices in the same network available, you can try connecting these. Note however that you might need to configure firewalls for allowing packets to actually arrive.

IP Sockets & Blocking Connections

In the last exercise we used sockets of the type `AF_UNIX`, now we will use IPv4 sockets, i.e. of type `AF_INET`. While we needed to copy our sockets filename into the `struct sockaddr_un` structure before, we now need to fill the `struct sockaddr_in` structure described in **`netinet/in.h(0)`**. This requires our IP and Port not to be given as strings but as `struct in_addr` and `in_port_t`, respectively. Luckily, there are functions to convert an IP in dotted-decimal notation (see examples) into such an IP structure (**`inet_pton(3)`**), as well as converting a port given as some unsigned integer into the required port type (**`htons(3)`**).[5]

[5] Why do we need to call `htons(3)` and what does this name even mean? This has to do with the so-called network byte-order or endianness. If you have two consecutive 2 B values, Byte 1–4, you could order these as B1, B2, B3, B4 or B2, B1, B4, B3, that is the bytes of each value are written 'backwards'. This has many useful properties, however computer architectures differ in their implementation. In order to be communicate across multiple architectures a network byte order has been standardized and we call host-to-network-short to guarantee our short (2 B) value is in the correct byte order.

That's it, we've adapted our local UNIX IPC client and server to be able to communicate over network using TCP/IP! However, when viewing connection-oriented systems, we have another issue to deal with: As soon as someone connects, the whole system is blocked. It would be rather bad if only one person could access a website at a time, telling everyone to wait until the current client is finished. In the previous exercise we explicitly allowed that behavior, we now want to fix this issue.

To solve it we need to have another closer look at the funciton **`accept(3)`**. In fact, this function does not simply 'pick up the phone' but actually only acts as some kind of 'secretary'. It accepts the connection *and creates a new socket*, forwarding the 'call' to it s.t. this new *connection* socket is the one actually used for further communication with our client (Step 2 in **Figure 2**). The original *listen* socket can still be 'called', but currently our process is busy talking and can't 'pick up'.

Ideally, we find a way to accept new connections (Step 3) while talking to the clients, that way our system is only blocked for as long as the 'secretary' needs to forward the 'call'.

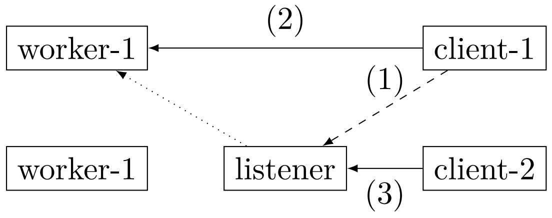

Figure 2 Connection-Oriented Sockets with a Listener

There are two ways to listen for new connections while talking to a client that we will both cover, but you need only implement one!

Extend using `fork(3)`

The first way to deal with this problem is to actually have multiple processes as part of our server program, one for listening and one for each connection to a client.[6] Specifically, when we start our program, our OS spawns one process as an instance of our program. However, this running process can then spawn another process from itself, effectively creating an almost identical copy. This technique is called '**`fork(3)`**ing' and the way to distinguish our process copy, called the 'child', from the orignal 'parent', is to look at the return value of the `fork(3)` system call, as it's different for parent and child.

[6] This is not using threads, but whole processes! Threads would be another way to deal with this, but we don't cover that here.

```
pid_t pid = fork();
if (pid == 0) {
    printf("We are the child\n");
    // child functionality here ...
} else {
    printf("We are the parent, child is %ld\n", pid);
    // parent functionality here ...
}
```

Listing 3 Example using `fork(3)`

More specifically the variable holds the process ID of our child *if we are the parent*, and the value `0` otherwise.

This means that our program contains both, the code of our child (or even children) and of the parent process, and we check with our if-statement which part of our program is to be executed by each process.

The idea is now to have our infinite loop run in our parent process as you can see in **Figure 4**. Each time it `accept(3)`s a connection by a client it `fork(3)`s to create a new process that actually deals with the connection. Meanwhile the parent immediately goes back to wait for new incoming connections it can `accept(3)`.

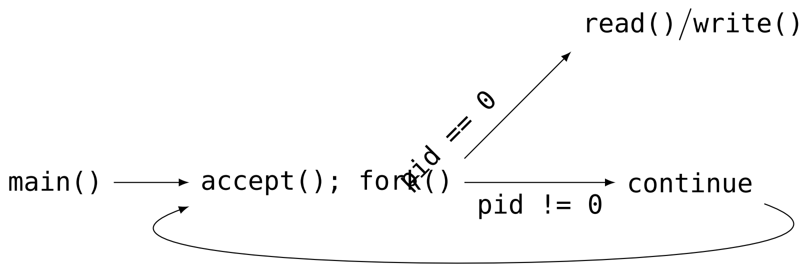

Figure 4 State diagram for `fork(3)`

Extend using `select(3)`

Instead of using processes running in parallel, we can use our one existing process to regularly walk over all sockets (the listen socket and the connection sockets), polling each whether new data is available. This technique, called synchronized I/O Multiplexing, is often implemented using **`select(3)`**.[7] This function introduces a data type called `fd_set` which can hold multiple file descriptors (i.e. our sockets) and is modified and read through the macros `FD_ZERO`, `FD_SET`, `FD_CLR` and `FD_ISSET`. At first, we only have one socket, the listen socket in our structure, so we do the following:

[7] Another alternative is to use **`poll(3)`** but we will not cover it, it's functionally equivalent.

```
fd_set fds;
FD_ZERO(&fds);
FD_SET(listen_sock, &fds); // add it to the set
```

The 'walking over all sockets' happens in two steps. First call our `select(3)` function, passing our set of file descriptors. But it needs another information to work: The highest fd in the set, plus one:

```
int max = listen_sock; // the only one in the set
fd_set activefds = fds;
select(max+1, &activefds, NULL, NULL, NULL);
```

This code will block (do nothing) as long as no fd in the set receives data. As soon as some sockets are 'active' in the sense that they received data, the function will remove all other, inactive sockets from the set (hence we copy it beforehand). Now we have a set of all sockets and a set of those that are ready to be read.

We now can loop over all *possible fds*, discarding those that aren't part of the set:

```
// fds are just integers!
for (int fd = 0; fd < max+1; fd++) {
    if (!FD_ISSET(fd, &activefds)) {
        continue;
    }
    /* fd is 'ready' */
}
```

As soon as we've found a fd ready to be read, we need to distinguish two cases: Is this fd our listen socket, or is it some connection socket? We can simply check whether `fd == listen_sock` is true and then accept the new connection, yielding a new connection socket in return. This socket needs to be added to our original fd set *and* `max` *needs to be updated!*

Otherwise we simply handle the connection, echoing the clients message on the server side. If the client asks the server to hang up the connection, we `close(3)` the connection socket and remove the respective fd from our set.[8]

[8] We don't need to update `max`.

Note: This option is not 'as parallel' as using processes, as we are busy as long as we read data from a client. However, hopefully, the consecutively sent data from the client is rather short, and we can go back to polling everyone rather quickly.