Supplement: TCP Fileserver

Leonard König, April 12, 2021

A (very) simple Fileserver protocol

We can now send messages over TCP back and forth, why not put this to good use? In this exercise you are tasked to only write a server (as that is the intersting part).

Specifically, a file allows a client to connect and send a message containing a filename. The server than looks up the specified file and opens it (if it exists), then copies the content through the filedescriptor of the socket to the client. You should already have the groundwork for creating the TCP based server, but you also know how to copy files through filedescriptors—you did just that in the `cat(1)` exercise at the beginning of the semester!

The only difference is that back then you used the functions `read(3)` and `write(3)` while now we use `recv(3)` and `send(3)`. In fact, these functions are equivalent, or more precisely, `recv(3)` and `send(3)` with their last argument set to zero behave exactly as `read(3)` and `write(3)` respectively do:

```
read(fd, buf, nbyte) ~ recv(fd, buf, nbyte, 0);
write(fd, buf, nbyte) ~ send(fd, buf, nbyte, 0);
```

So all you need to do is put those two parts together: Server creates and listens, then accepts a connection, the child `recv(3)`s the filename from the client, `open(3)`s the file and copies it to the socket just the way we did before.

Remember though, that if you enter a message on the client (be it your own, `nc(1)` or `telnet(1)`) and hit the `ENTER` key, this will not only send the filename but also append a `\n` character! Instead you want to use `^D` (`Control+D`) to send the ASCII EOT character, submitting the filename to the server without appending `\n`.

IPv4 and IPv6 (Optional)

Our current implementation only supports IPv4, and while this will likely co-exist with the newer IPv6 standard for quite some time,

we want to be future proof. This goes a little beyond the bare minimum of the task but we highly recommend you to continue reading ;)

A big goal in software development is to be 'portable'. That means that our code can run on different platforms, be it Windows, Linux, macOS, FreeBSD, or, like now, IPv4 and IPv6. Ideally, we don't really need to write two almost identical versions of the program or deal with details of either protocol. The key to enable portability is abstraction, we've covered a few portable abstractions already. Sockets are an abstraction over network interfaces, file descriptors an abstraction over things (not necessarily files) 'that can be written to and read from'. We profited from those well-designed abstractions as well, the TCP communication was almost identical to the UNIX IPC communication and now our fileserver pushing a file over network shares a big deal of code with a program that simply prints a file to the screen—it's the same thing, as far as the abstraction is concerned.

And thus, of course, there exists a function that abstracts IPv4 and IPv6.[1] This function is called **`getaddrinfo(3)`** and is meant to be used as a replacement to filling the `struct sockaddr` structure manually.

[1] Actually even DNS.

We can have many different kinds of connections, listen/server type, IPv4 and IPv6 based connections, TCP or UDP, etc. and combinations thereof. This new function allows us to 'filter' all possible connections by fixing some as constraints. In this exercise we want to allow for IPv4 and IPv6 but restrict ourselves to TCP.

We do that by filling a structure introduced by `getaddrinfo`, called `struct addrinfo`. This structure has members for each 'constraint' we may want to have: `ai_flags` (listen/server mode vs. connect/client), `ai_family` (IPv4 or IPv6), `ai_socktype` (STREAM or DGRAM, i.e. TCP or UDP), and `ai_protocol` (again IPv4 or IPv6).[2]

[2] And some more, but we don't care about them that much.

Each member can now be filled by their respective values (e.g. `SOCK_STREAM` for `ai_socktype`) or left unspecified by setting it to a special value. Quoting from POSIX:

> "A value of `AF_UNSPEC` for `ai_family` means that the caller shall accept any address family. A value of zero for `ai_socktype` means that the caller shall accept any socket type. A value of zero for `ai_protocol` means that the caller shall accept any protocol."

```
struct addrinfo hints = {
    .ai_flags    = /* ... */,
    .ai_family   = /* ... */,
    .ai_socktype = /* ... */,
    .ai_protocol = /* ... */,
    .ai_next     = NULL,
};
```

Listing 1 Setting the hints for `getaddrinfo(3)`

After we specified our 'hints'/constraints, we can ask `getaddrinfo(3)` to lookup all possible configurations which fulfill these, given a specific address and port. E.g. for listening on *any available IP address*, port 5000:[3]

```
struct addrinfo infos;
getaddrinfo(NULL, "5000", &hints, &infos);
```

[3] You can also specify e.g. IPv4 localhost, but this will obviously restrict us to IPv4, regardless of the settings in `hints`.

Now the `infos` structure will contain the first element of a linked-list of entries that all fit the description.

We can then simply iterate over the structure, creating sockets for all those matching descriptions as seen in **Listing 2**.

```
struct addrinfo *p;
for (p = info; p != NULL; p = p->ai_next) {
    /* access p->ai_* */
}
```

Listing 2 Iterating over addrinfo

In this exercise we will, however, it suffices to use the first entry in the list that works. We do that by trying to create a socket and binding it:

```
int s = socket(p->ai_family, p->ai_socktype,
    p->ai_protocol);
bind(s, p->ai_addr, p->ai_addrlen);
```

If either fails (error handling!), we try the next `info` entry.

After we successfully bound a socket to an address we can discard the `info` structure and carry on exactly like we did in all the exercises before. That's it!

We can use the same functions for clients as well, the only difference is that `ai_flags` is set to a different value. Also this makes it quite easy to connect to other services as `getaddrinfo` doesn't only accept IPs as its first argument but also domains! You can do:

```
getaddrinfo("www.hoogle.com", "80", &hints, &infos);
```