Binding On Port 0

December 1, 2015

A commonly occurring problem is that you you have a program that needs to bind on some TCP or UDP port, but you don't actually care what port is chosen. The most obvious way to solve this would be to pick a port at random, but this is a poor solution because you could randomly pick a port that's already in use. Another solution here is to bind on port 0. If you do this the kernel will select an unused port from the ephemeral port range. You can subsequently use getsockname(2) to determine which port was actually chosen.

There's nothing wrong with doing this *per se*. However, frequently this pattern is abused when program A is trying to decide on a port that program B should use. One common case I've seen this come up in is with test suites. There are two common variations here. The first is where the program you're testing listens on a TCP (or UDP) port and you're trying to do full end-to-end tests. In this case your test program needs to launch the binary, connect to it over the port, and then interact with it. An example of this (which I admittedly had my hand in) is the tests for statsrelay. The other case is where you've got some regular program that doesn't listen on a TCP port, but communicates over TCP with another program like Redis or MySQL. In this case your test suite might launch an instance of Redis or MySQL and subsequently needs to know what TCP port it should use to communicate with this "sidecar" process.

In both cases, frequently what you'll see is a pattern like this:

- the test suite creates a socket and binds it to port 0
- the port that was bound on is discovered using `getsockname()`
- the socket is closed
- the second program is launched with an argument telling it to use the recently allocated port
- the second program allocates the socket with SO_REUSEADDR and then tries to bind with the specified port

This will usually work but it does have a few problems.

The most important problem here is that the port isn't guaranteed to still be available when the second program tries to use it. In practice under normal circumstances it almost always will be available for two reasons. The first is that in the common case that you're using a TCP port your TCP stack will put the port into the so-called TIME_WAIT state. When this happens the kernel will reserve the port (actually, the interface/port tuple) for some amount of time before allowing another program to use it. On Linux this defaults to two minutes, so if you can get the second program to bind in under two minutes you're usually in the clear. The second reason is that in practice the kernel on most systems—and at the very least on Linux—will try to allocate ports in this way pseudo-sequentially. What this means is that if you bind on port 0 and actually bind to port 55441, then the next process that tries to bind to port 0 will probably get port 55442 (if it's available), and the next process will get port 55443, and so on. Eventually this will wrap around which means that two processes that bind to port 0 could eventually get allocated the same port. In practice unless the number of free ports is extremely small usually it will take a long time to wrap around and therefore this isn't usually an issue.

The second issue here is that for this strategy to work you need to bind the second time with SO_REUSEADDR. This isn't really a big deal, but it could be annoying if the program you're trying to invoke doesn't already use this socket option. It also means that you're not shielded by the TIME_WAIT feature I mentioned above, meaning that if a lot of processes are doing this you're much more likely to get dreaded port collisions.

The fact that this could happen at all becomes an issue once you have a big test suite with lots of jobs running. Furthermore it's possible to induce the system into a state where there are very few free ports which can make port collisions very common. For instance, just establishing an outbound TCP connection uses an ephemeral port, so a program that creates many TCP sockets rapidly can easily exhaust the ephemeral port range.

For programs that operate over TCP there's actually a pretty elegant solution to this problem that avoids the issues I mentioned above. In their infinite wisdom the designers of the Berkeley sockets API decided that sockets have "families" and "types". The type of a stream-oriented socket like a TCP socket is SOCK_STREAM (and the family is AF_INET). You can also create an AF_UNIX socket (commonly called a "Unix domain socket") that is of type SOCK_STREAM. Because of this it's usually pretty easy to take a program that operates over TCP streams and make it work with AF_UNIX sockets. The basic programming interface of the two is exactly the same—only programs that are doing advanced things will notice the difference between the two.

If you can use an AF_UNIX socket with your program there's an easy way to safely create a socket to use between two programs:

- Use mkdtemp(3) to safely create a new unique directory without race conditions.
- Create the socket with some well known name like `THEDIR/mysocket.sock` and pass it as a command line argument to the subprocess.
- Have the subprocess bind to the named socket.

This avoids all race conditions. It also works with a much greater concurrency level because while TCP is limited to 64k sockets, your filesystem can have many more directories.

The one caveat with this approach is that you will need to make sure that you delete the temporary directory later. Usually this is easy to do with a context manager, a bash trap, etc. However there can still be cases where the cleanup doesn't happen, for instance if your process is terminated by SIGKILL. If you're on Linux one interesting (and non-portable!) solution to this problem is to create an "abstract Unix domain socket". The way this works is that when you create the address to bind the AF_UNIX socket on you can fill out the struct sockaddr_un so that the `sun_path` field stats with a null byte. If you do this then the path is treated as an abstract path that doesn't correspond to an actual path on the local filesystem. The kernel will then track references to the socket and clean the socket up when no further references to it remain.