Bind: Address Already in Use

Or How to Avoid this Error when Closing TCP Connections

Normal Closure

In order for a network connection to close, both ends have to send `FIN` (final) packets, which indicate they will not send any additional data, and both ends must `ACK` (acknowledge) each other's `FIN` packets. The `FIN` packets are initiated by the application performing a `close()`, a `shutdown()`, or an `exit()`. The `ACK`s are handled by the kernel *after* the `close()` has completed. Because of this, it is possible for the process to complete before the kernel has released the associated network resource, and this port cannot be bound to another process until the kernel has decided that it is done.

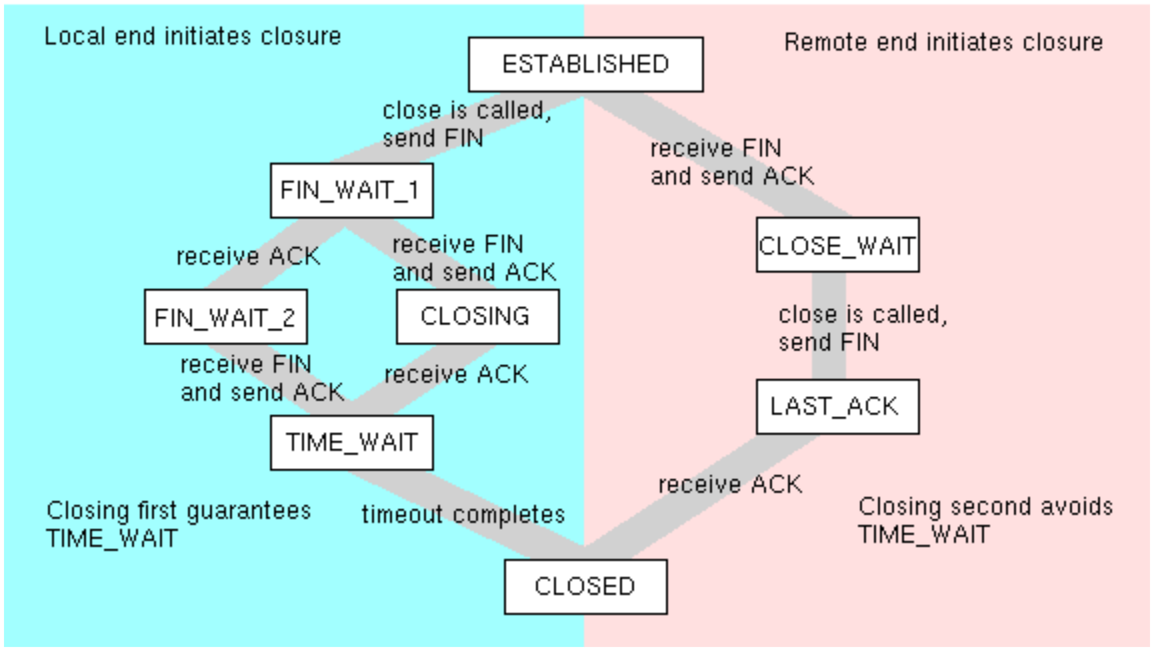

Figure 1

Figure 1 shows all of the possible states that can occur during a normal closure, depending on the order in which things happen. Note that if you initiate closure, there is a `TIME_WAIT` state that is absent from the other side. This `TIME_WAIT` is necessary in case the `ACK` you sent wasn't received, or in case spurious packets show up for other reasons. I'm really not sure why this state isn't necessary on the other side, when the remote end initiates closure, but this is definitely the case. **`TIME_WAIT` is the state that typically ties up the port for several minutes after the process has completed.** The length of the associated timeout varies on different operating systems, and may be dynamic on some operating systems, however typical values are in the range of one to four minutes.

If both ends send a `FIN` before either end receives it, both ends will have to go through `TIME_WAIT`.

Normal Closure of Listen Sockets

A socket which is listening for connections can be closed immediately if there are no connections pending, and the state proceeds directly to `CLOSED`. If connections are pending however, `FIN_WAIT_1` is entered, and a `TIME_WAIT` is inevitable.

Note that it is impossible to completely guarantee a clean closure here. While you can check the connections using a `select()` call before closure, a tiny but real possibility exists that a connection could arrive after the `select()` but before the `close()`.

Abnormal Closure

If the remote application dies unexpectedly while the connection is established, the local end will have to initiate closure. In this case `TIME_WAIT` is unavoidable. If the remote end disappears due to a network failure, or the remote machine reboots (both are rare), the local port will be tied up until each state times out. Worse, some older operating systems do not implement a timeout for `FIN_WAIT_2`, and it is possible to get stuck there forever, in which case restarting your server could require a reboot.

If the local application dies while a connection is active, the port will be tied up in `TIME_WAIT`. This is also true if the application dies while a connection is pending.

Strategies for Avoidance

SO_REUSEADDR

You can use `setsockopt()` to set the `SO_REUSEADDR` socket option, which explicitly allows a process to bind to a port which remains in `TIME_WAIT` (it still only allows a single process to be bound to that port). This is the both the simplest and the most effective option for reducing the "address already in use" error.

Oddly, using `SO_REUSEADDR` can actually lead to more difficult "address already in use" errors. `SO_REUSADDR` permits you to use a port that is stuck in `TIME_WAIT`, but you still **can not** use that port to establish a connection to the last place it connected to. What? Suppose I pick local port 1010, and connect to foobar.com port 300, and then close locally, leaving that port in `TIME_WAIT`. I can reuse local port 1010 right away to connect to anywhere **except for** foobar.com port 300.

A situation where this might be a problem is if my program is trying to find a reserved local port (< 1024) to connect to some service which likes reserved ports. If I used `SO_REUSADDR`, then each time I run the program on my machine, I'll keep getting the **same** local reserved port, even if it is stuck in `TIME_WAIT`, and I risk getting a "connect: Address already in use" error if I go back to any place I've been to in the last few minutes. The solution here is to **avoid** `SO_REUSEADDR`.

Some folks don't like `SO_REUSEADDR` because it has a security stigma attached to it. On some operating systems it allows the same port to be used with a different address on the same machine by different processes at the same time. This is a problem because most servers bind to the port, but they don't bind to a specific address, instead they use `INADDR_ANY` (this is why things show up in `netstat` output as `*.8080`). So if the server is bound to *.8080, another malicious user on the local machine can bind to local-machine.8080, which will intercept all of your connections since it is more specific. This is only a problem on multi-user machines that don't have restricted logins, it is **NOT** a vulnerability from outside the machine. And it is easily avoided by binding your server to the machine's address.

Additionally, others don't like that a busy server may have hundreds or thousands of these `TIME_WAIT` sockets stacking up and using kernel resources. For these reasons, there's another option for avoiding this problem.

Client Closes First

Looking at the diagram above, it is clear that `TIME_WAIT` can be avoided if the remote end initiates the closure. So the server can avoid problems by letting the client close first. The application protocol must be designed so that the client knows when to close. The server can safely close in response to an `EOF` from the client, however it will also need to set a timeout when it is expecting an EOF in case the client has left the network ungracefully. In many cases simply waiting a few seconds before the server closes will be adequate.

It probably makes more sense to call this method "Remote Closes First", because otherwise it depends on what you are calling the client and the server. If you are developing some system where a cluster of client programs sit on one machine and contact a variety of different servers, then you would want to foist the responsibility for closure onto the servers, to protect the resources on the client.

For example, I wrote a script that uses `rsh` to contact all of the machines on our network, and it does it in parallel, keeping some number of connections open at all times. `rsh` source ports are arbitrary available ports less than 1024. I initially used "`rsh -n`", which it turns out causes the local end to close first. After a few tests, every single free port less than 1024 was stuck in `TIME_WAIT` and I couldn't proceed. Removing the "`-n`" option causes the remote (server) end to close first (understanding why is left as an exercise for the reader), and should've eliminated the `TIME_WAIT` problem. However, without the -n, rsh can hang waiting for input. And, if you close input at the local end, this can again result in the port going into `TIME_WAIT`. I ended up avoiding the system-installed rsh program, and developing my own implementation in perl. My current implementation, multi-rsh, is available for download

Reduce Timeout

If (for whatever reason) neither of these options works for you, it may also be possible to shorten the timeout associated with `TIME_WAIT`. Whether this is possible and how it should be accomplished depends on the operating system you are using. Also, making this timeout too short could have negative side-effects, particularly in lossy or congested networks.

Fine's Home Send Me Email