The Perils of Closing a Socket with a Non-Empty RecvQ

Consider two peers, A and B, communicating via TCP. If B closes a socket and there is any data in B's receive queue, B sends a TCP RST to A instead of following the standard TCP closing protocol, resulting in an error return value from recv( ).

A		**B**
send( )	data →	
	data →	
	data →	
recv( ) →ERROR	← RST	close( )

When might this situation occur? Consider a simple protocol where A sends 5 prime numbers to B, and B responds with "OK" and closes the connection. B reads one number at a time. When B receives a nonprime number, it assumes A is confused, responds with "ERROR", and closes the connection. If A sends 5 numbers and the second number is not prime, then B will send "ERROR" and call close( ) with data (i.e., 3 numbers) in its receive queue. On some systems, this will cause a TCP RST to be sent to A. Meanwhile, A is blocked on a read, awaiting word from B. The RST from B causes A's attempt to read to fail so A never receives the "ERROR" message from B.

So why not just have B read all 5 prime numbers before closing? Consider the case where A violates the protocol and sends 6 prime numbers. Here B reads the first 5 prime numbers and closes the socket with the 6th prime still in its receive queue, resulting in a RST. In general, we cannot know that an endpoint's receive queue is empty before we call close. We could try to empty the receive queue before calling close; however, we cannot guarantee that more data will not arrive between our queue emptying and close calls.

Aside: You may be wondering why any TCP implementation would send a RST on close if data is still in the receive queue. The reason is that RFC2525 tells them that a correct TCP implementation should do this (see Section on Failure to RST on close with data pending). The argument is that TCP is reliable, and allowing a TCP connection to terminate normally when you know data has not been delivered to the application violates reliability. Note that this RST problem does not occur in all operating systems. Some simply ignore the receive queue.

We can view this RST behavior with a minor modification to the TCP Echo Server. To do this, modify the server such that when a client connects, the server reads and writes a single byte, then closes the connection. First, try the client sending a single byte. It works great! However, if the client sends more than one byte, this new server will read/write the first byte and call close with data in its receive queue, causing a RST to be sent to the client. The client will be blocked on a read, awaiting the response from the server. Upon receiving the RST, the client's read fails with an error message about a reset.

So how do we fix this? We can use shutdown. If we shutdown the write/output direction of the connection before calling close, we avoid the RST behavior. To see this, add a call to shutdown in the write/output direction before the call to close.