Detection of Half-Open (Dropped) TCP/IP Socket Connections

Stephen Cleary, 20 Jun 2009

How to detect a dropped TCP/IP connection

(This post is part of the TCP/IP .NET Sockets FAQ.)

There is a three-way handshake to open a TCP/IP connection, and a four-way handshake to close it. However, once the connection has been established, if neither side sends any data, then no packets are sent over the connection. TCP is an "idle" protocol, happy to assume that the connection is active until proven otherwise.

TCP was designed this way for resiliency and efficiency. This design enables a graceful recovery from unplugged network cables and router crashes. e.g., a client may connect to a server, an intermediate router may be rebooted, and after the router comes back up, the original connection still exists (this is true unless data is sent across the connection while the router was down). This design is also efficient, since no "polling" packets are sent across the network just to check if the connection is still OK (reduces unnecessary network traffic).

TCP does have acknowledgments for data, so when one side sends data to the other side, it will receive an acknowledgment if the connection is still active (or an error if it is not). Thus, broken connections can be detected by sending out data. It is important to note that the act of *receiving* data is completely passive in TCP; a socket that only reads cannot detect a dropped connection.

This leads to a scenario known as a "half-open connection". At any given point in most protocols, one side is expected to send a message and the other side is expecting to receive it. Consider what happens if an intermediate router is suddenly rebooted at that point: the receiving side will continue waiting for the message to arrive; the sending side will send its data, and receive an error indicating the connection was lost. Since broken connections can only be detected by *sending* data, the receiving side will wait forever. This scenario is called a "half-open connection" because one side realizes the connection was lost but the other side believes it is still active.

> Terminology alert: "half-open" is completely different than "half-closed". Half-closed connections are when one side performs a Shutdown operation on its socket, shutting down only the sending (outgoing) stream. See Socket Operations for more details on the Shutdown operation.

Causes of Half-Open Connections

Half-open connections are in that annoying list of problems that one seldomly sees in a test environment but commonly happen in the real world. This is because if the socket is shut down with the normal four-way handshake (or even if it is abruptly closed), the half-open problem will not occur. Some of the common causes of a half-open connection are described below:

- **Process crash.** If a process shuts down normally, it usually sends out a "FIN" packet, which informs the other side that the connection has been lost. However, if a process crashes or is terminated (e.g., from Task Manager), this is not guaranteed. It is possible that the OS will send out a "FIN" packet on behalf of a crashed process; however, this is up to the OS.
- **Computer crash.** If the entire computer (including the OS) crashes or loses power, then there is certainly no notification to the other side that the connection has been lost.
- **Router crash/reboot.** Any of the routers along the route from one side to the other may also crash or be rebooted; this causes a loss of connection if data is being sent at that time. If no data is being sent at that exact time, then the connection is not lost.
- **Network cable unplugged.** Any network cables unplugged along the route from one side to the other will cause a loss of connection without any notification. This is similar to the router case; if there is no data being transferred, then the

connection is not actually lost. However, computers usually will detect if their specific network cable is unplugged and may notify their local sockets that the network was lost (the remote side will not be notified).
- **Wireless devices (including laptops) moving out of range.** A wireless device that moves out of its access point range will lose its connection. This is an often-overlooked but increasingly common situation.

In all of the situations above, it is possible that one side may be aware of the loss of connection, while the other side is not.

Is Detection Necessary?

There are some situations in which detection is not necessary. Even when it is necessary, one must consider how quickly the process needs to detect dropped connections.

The needs of half-open detection can be roughly summarized by three categories:

1. **Unnecessary.** If the application protocol sends data regularly. A "poll" system (as opposed to a "subscription/event" system) already has a timer built in (the poll timer), and sends data across the connection regularly.
2. **Necessary, but not a "hard" requirement.** Server applications need to detect dropped connections, or else those connections will eat up system resources over time; they don't really need to know *immediately*, though. Likewise, client applications waiting for a server response may choose to wait a long time (often resulting in the user retrying the operation).
3. **Necessary with a "soft real-time" requirement.** If the application needs or wants to actively detect the loss of connection even though there is no data flowing across it, then it must actively test the connection. This is also useful if server applications wish to reclaim resources more quickly, or client applications wish to report errors to users more quickly.

The necessity of detection must be considered separately for each side of the communication. e.g., if the protocol is based on a polling scheme, then the side doing the polling does not need explicit keepalive handling, but the side responding to the polling likely does need explicit keepalive handling.

> True Story: I once had to write software to control a serial device that operated through a "bridge" device that exposed the serial port over TCP/IP. The company that developed the bridge implemented a simple protocol: they listened for a single TCP/IP connection (from anywhere), and - once the connection was established - sent any data received from the TCP/IP connection to the serial port, and any data received from the serial port to the TCP/IP connection. Of course, they only allowed one TCP/IP connection (otherwise, there could be contention over the serial port), so other connections were refused as long as there was an established connection.
>
> The problem? No keepalives. If the bridge ever ended up in a half-open situation, it would *never recover*; any connection requests would be rejected because the bridge would believe the original connection was still active. Usually, the bridge was deployed to a stationary device on a physical network; presumably, if the device ever stopped working, someone would walk over and perform a power cycle. However, we were deploying the bridge onto mobile devices on a wireless network, and it was normal for our devices to pass out of and back into access point coverage. Furthermore, this was part of an automated system, and people weren't near the devices to perform a power cycle. Of course, the bridge failed during our prototyping; when we brought the root cause to the other company's attention, they were unable to implement a keepalive (the embedded TCP/IP stack didn't support it), so they worked with us in developing a method of remotely resetting the bridge.
>
> It's important to note that we *did* have keepalive testing on our side of the connection (via a timer), but this was insufficient. **It is necessary to have keepalive testing on both sides of the connection.**
>
> This bridge was in full production, and had been for some time. The company that made this error was a billion-dollar global corporation centered around networking products. The company I worked for had four programmers at the time. This just goes to show that even the big guys can make mistakes.

Wrong Methods to Detect Dropped Connections

There are a couple of wrong methods to detect dropped connections. Beginning socket programmers often come up with these incorrect solutions to the half-open problem. They are listed here only for reference, along with a brief description of why they are wrong.

- **A Second socket connection.** A new socket connection cannot determine the validity of an existing connection in all cases. In particular, if the remote side has crashed and rebooted, then a second connection attempt will succeed even though the original connection is in a half-open state.
- **Ping.** Sending a ping (ICMP) to the remote side has the same problem: it may succeed even when the connection is unusable. Furthermore, ICMP traffic is often treated differently than TCP traffic by routers.

Correct Methods to Detect Dropped Connections

There are several correct solutions to the half-open problem. Each one has their pros and cons, depending on the problem domain. This list is in order from best solution to worst solution (IMO):

1. **Add a keepalive message to the application protocol framing (an empty message).** Length-prefixed and delimited systems may send empty messages (e.g., a length prefix of "0 bytes" or a single "end delimiter").
    - *Advantages.* The higher-level protocol (the actual messages) are not affected.
    - *Disadvantages.* This requires a change to the software on both sides of the connection, so it may not be an option if the application protocol is already specified and immutable.
2. **Add a keepalive message to the actual application protocol (a "null" message).** This adds a new message to the application protocol: a "`null`" message that should just be ignored.
    - *Advantages.* This may be used if the application protocol uses a non-uniform message framing system. In this case, the first solution could not be used.
    - *Disadvantages.* (Same as the first solution) This requires a change to the software on both sides of the connection, so it may not be an option if the application protocol is already specified and immutable.
3. **Explicit timer assuming the worst.** Have a timer and assume that the connection has been dropped when the timer expires (of course, the timer is reset each time data is transferred). This is the way HTTP servers work, if they support persistent connections.
    - *Advantages.* Does not require changes to the application protocol; in situations where the code on the remote side cannot be changed, the first two solutions cannot be used. Furthermore, this solution causes less network traffic; it is the only solution that does not involve sending out keepalive (i.e., "are you still there?") packets.
    - *Disadvantages.* Depending on the protocol, this may cause a high number of valid connections to be dropped.
4. **Manipulate the TCP/IP keepalive packet settings.** This is a highly controversial solution that has complex arguments for both pros and cons. It is discussed in depth in Stevens' book, chapter 23. Essentially, this instructs the TCP/IP stack to send keepalive packets periodically on the application's behalf. There are two ways that this can be done:
    1. Set `SocketOptionName.KeepAlive`. The MSDN documentation isn't clear that this uses a 2-hour timeout, which is too long for most applications. This can be changed (system-wide) through a registry key, but changing this system-wide (i.e., for all other applications) is greatly frowned upon. This is the old-fashioned way to enable keepalive packets.
    2. Set per-connection keepalives. Keepalive parameters can be set per-connection only on Windows 2000 and newer, not the old 9x line. This has to be done by issuing I/O control codes to the socket: pass IOControlCode.KeepAliveValues along with a structure to Socket.IOControl; the necessary structure is not covered by the .NET documentation but is described in the unmanaged documentation for WSAIoctl (SIO_KEEPALIVE_VALS).
        - *Advantages.* Once the code to set the keepalive parameters is working, there is nothing else that the application needs to change. The other solutions all have timer events that the application must respond to; this one is "set and forget".
        - *Disadvantages.* RFC 1122, section 4.2.3.6 indicates that acknowledgements for TCP keepalives without data may not be transmitted reliably by routers; this may cause valid connections to be dropped. Furthermore, TCP/IP stacks are not required to support keepalives at all (and many embedded stacks do not), so this solution may not translate to other platforms.

Each side of the application protocol may employ different keepalive solutions, and even different keepalive solutions at different states in the protocol; however, one of the solutions above should always be used. For example, the client side of a request/response style protocol may choose to send "`null`" requests when there is not a request pending, and switch to a timeout solution while waiting for a response.

However, when designing a new protocol, it is best to employ one of the solutions consistently.

(This post is part of the TCP/IP .NET Sockets FAQ.)

License

About the Author

Stephen Cleary
Software Developer (Senior)
United States

Stephen Cleary is a Christian, husband, father, and programmer living in Northern Michigan.

Personal home page (including blog): http://www.stephencleary.com/

Comments and Discussions

4 messages have been posted for this article Visit **https://www.codeproject.com/Articles/37490/Detection-of-Half-Open-Dropped-TCP-IP-Socket-Conne** to post and view comments on this article, or click **here** to get a print view with messages.