23

TCP Keepalive Timer

23.1 Introduction

Many newcomers to TCP/IP are surprised to learn that no data whatsoever flows across an idle TCP connection. That is, if neither process at the ends of a TCP connection is sending data to the other, nothing is exchanged between the two TCP modules. There is no polling, for example, as you might find with other networking protocols. This means we can start a client process that establishes a TCP connection with a server, and walk away for hours, days, weeks or months, and the connection remains up. Intermediate routers can crash and reboot, phone lines may go down and back up, but as long as neither host at the ends of the connection reboots, the connection remains established.

This assumes that neither application—the client or server—has application-level timers to detect inactivity, causing either application to terminate. Recall at the end of Section 10.7 that BGP sends an application probe to the other end every 30 seconds. This is an application timer that is independent of the TCP keepalive timer.

There are times, however, when a server wants to know if the client's host has either crashed and is down, or crashed and rebooted. The *keepalive timer*, a feature of many implementations, provides this capability.

> Keepalives are not part of the TCP specification. The Host Requirements RFC provides three reasons not to use them: (1) they can cause perfectly good connections to be dropped during transient failures, (2) they consume unnecessary bandwidth, and (3) they cost money on an internet that charges by the packet. Nevertheless, many implementations provide the keepalive timer.

The keepalive timer is a controversial feature. Many feel that this polling of the other end has no place in TCP and should be done by the application, if desired. This is one of the *religious issues*, because of the fervor expressed by some on the topic.

The keepalive option can cause an otherwise good connection between two processes to be terminated because of a temporary loss of connectivity in the network joining the two end systems. For example, if the keepalive probes are sent during the time that an intermediate router has crashed and is rebooting, TCP will think that the client's host has crashed, which is not what has happened.

The keepalive feature is intended for server applications that might tie up resources on behalf of a client, and want to know if the client host crashes. Many versions of the Telnet server and Rlogin server enable the keepalive option by default.

A common example showing the need for the keepalive feature nowadays is when personal computer users use TCP/IP to login to a host using Telnet. If they just power off the computer at the end of the day, without logging off, they leave a half-open connection. In Figure 18.16 we showed how sending data across a half-open connection caused a reset to be returned, but that was from the client end, where the client was sending the data. If the client disappears, leaving the half-open connection on the server's end, and the server is waiting for some data from the client, the server will wait forever. The keepalive feature is intended to detect these half-open connections from the server side.

23.2 Description

In this description we'll call the end that enables the keepalive option the server, and the other end the client. There is nothing to stop a client from setting this option, but normally it's set by servers. It can also be set by both ends of a connection, if it's important for each end to know if the other end disappears. (In Chapter 29 we'll see that when NFS uses TCP, both the client and server set this option. But in Chapter 26 with Rlogin and Telnet, only the servers set the option, not the clients.)

If there is no activity on a given connection for 2 hours, the server sends a probe segment to the client. (We'll see what the probe segment looks like in the examples that follow.) The client host must be in one of four states.

1. The client host is still up and running and reachable from the server. The client's TCP responds normally and the server knows that the other end is still up. The server's TCP will reset the keepalive timer for 2 hours in the future. If there is application traffic across the connection before the next 2-hour timer expires, the timer is reset for 2 hours in the future, following the exchange of data.
2. The client's host has crashed and is either down or in the process of rebooting. In either case, its TCP is not responding. The server will not receive a response to its probe and it times out after 75 seconds. The server sends a total of 10 of these probes, 75 seconds apart, and if it doesn't receive a response, the server considers the client's host as down and terminates the connection.
3. The client's host has crashed and rebooted. Here the server will receive a response to its keepalive probe, but the response will be a reset, causing the server to terminate the connection.

4. The client's host is up and running, but unreachable from the server. This is the same as scenario 2, because TCP can't distinguish between the two. All it can tell is that no replies are received to its probes.

The server does not have to worry about the client's host being shut down and then rebooted. (This refers to an operator shutdown, instead of the host crashing.) When the system is shut down by an operator, all application processes are terminated (i.e., the client process), which causes the client's TCP to send a FIN on the connection. Receiving the FIN would cause the server's TCP to report an end-of-file to the server process, allowing the server to detect this scenario.

In the first scenario the server application has no idea that the keepalive probes are taking place. Everything is handled at the TCP layer. It's transparent to the application until one of scenarios 2, 3, or 4 occurs. In these three scenarios an error is returned to the server application by its TCP. (Normally the server has issued a read from the network, waiting for data from the client. If the keepalive feature returns an error, it is returned to the server as the return value from the read.) In scenario 2 the error is something like "connection timed out," and in scenario 3 we expect "connection reset by peer." The fourth scenario may look like the connection timed out, or may cause another error to be returned, depending on whether an ICMP error related to the connection is received. We look at all four scenarios in the next section.

A perpetual question by people discovering the keepalive option is whether the 2-hour idle time value can be changed. They normally want it much lower, on the order of minutes. As we show in Appendix E, the value can usually be changed, but in all the systems described in this appendix, the keepalive interval is a system-wide value, so changing it affects all users of the option.

The Host Requirements RFC says that an implementation may provide the keepalive feature, but it must not be enabled unless the application specifically says so. Also, the keepalive interval must be configurable, but it must default to no less than 2 hours.

23.3 Keepalive Examples

We'll now go through scenarios 2, 3, and 4 from the previous section, to see the packets exchanged using the keepalive option.

Other End Crashes

Let's see what happens when the server host crashes and does not reboot. To simulate this we'll do the following steps:

- Establish a connection between a client (our `sock` program on the host `bsdi`) and the standard echo server on the host `svr4`. The client enables the keepalive option with the `-K` option.
- Verify that data can go across the connection.
- Watch the client's TCP send keepalive packets every 2 hours, and see them acknowledged by the server's TCP.

- Disconnect the Ethernet cable from the server, and leave it off until the example is complete. This makes the client think the server host has crashed.
- We expect the client to send 10 keepalive probes, 75 seconds apart, before declaring the connection dead.

Here is the interactive output on the client:

```
bsdi % sock -K svr4 echo                 -K for keepalive option
hello, world                             type this at beginning, to verify connection is up
hello, world                             and see this echoed
                                         disconnect Ethernet cable after 4 hours
read error: Connection timed out         this happens about 6 hours and 11 minutes after start
```

Figure 23.1 shows the `tcpdump` output. (We have removed the connection establishment and the window advertisements.)

```
 1      0.0                  bsdi.1055 > svr4.echo: P 1:14(13) ack 1
 2      0.006105 (   0.0061) svr4.echo > bsdi.1055: P 1:14(13) ack 14
 3      0.093140 (   0.0870) bsdi.1055 > svr4.echo: . ack 14

 4   7199.972793 (7199.8797) arp who-has svr4 tell bsdi
 5   7199.974878 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
 6   7199.975741 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
 7   7199.979843 (   0.0041) svr4.echo > bsdi.1055: . ack 14

 8  14400.134330 (7200.1545) arp who-has svr4 tell bsdi
 9  14400.136452 (   0.0021) arp reply svr4 is-at 0:0:c0:c2:9b:26
10  14400.137391 (   0.0009) bsdi.1055 > svr4.echo: . ack 14
11  14400.141408 (   0.0040) svr4.echo > bsdi.1055: . ack 14

12  21600.318309 (7200.1769) arp who-has svr4 tell bsdi
13  21675.320373 (  75.0021) arp who-has svr4 tell bsdi
14  21750.322407 (  75.0020) arp who-has svr4 tell bsdi
15  21825.324460 (  75.0021) arp who-has svr4 tell bsdi
16  21900.436749 (  75.1123) arp who-has svr4 tell bsdi
17  21975.438787 (  75.0020) arp who-has svr4 tell bsdi
18  22050.440842 (  75.0021) arp who-has svr4 tell bsdi
19  22125.432883 (  74.9920) arp who-has svr4 tell bsdi
20  22200.434697 (  75.0018) arp who-has svr4 tell bsdi
21  22275.436788 (  75.0021) arp who-has svr4 tell bsdi
```

Figure 23.1 Keepalive packets that determine that a host has crashed.

Lines 1, 2, and 3 send the line "hello, world" from the client to the server and back. The first keepalive probe occurs 2 hours (7200 seconds) later on line 4. The first thing we see is an ARP request and an ARP reply, before the TCP segment on line 6 can be sent. The keepalive probe on line 6 elicits a response from the other end (line 7). The same sequence of packets is exchanged 2 hours later in lines 8–11.

If we could see all the fields in the keepalive probes, lines 6 and 10, we would see that the sequence number field is one less than the next sequence number to be sent (i.e., 13 in this example, when it should be 14), but because there is no data in the segment, `tcpdump` does not print the sequence number field. (It only prints the sequence number for empty segments that contain the SYN, FIN, or RST flags.) It is the receipt of this

incorrect sequence number that forces the server's TCP to respond with an ACK to the keepalive probe. The response tells the client the next sequence number that the server is expecting (14).

> Some older implementations based on 4.2BSD do not respond to these keepalive probes unless the segment contains data. Some systems can be configured to send one garbage byte of data in the probe to elicit the response. The garbage byte causes no harm, because it's not the expected byte (it's a byte that the receiver has previously received and acknowledged), so it's thrown away by the receiver. Other systems send the 4.3BSD-style segment (no data) for the first half of the probe period, and if no response is received, switch to the 4.2BSD-style segment for the last half.

We then disconnect the cable and expect the next probe, 2 hours later, to fail. When this next probe takes place, notice that we never see the TCP segments on the cable, because the host is not responding to ARP requests. We can still see that the client sends 10 probes, spaced 75 seconds apart, before giving up. We can see from our interactive script that the error code returned to the client process by TCP gets translated into "Connection timed out," which is what happened.

Other End Crashes and Reboots

In this example we'll see what happens when the client crashes and reboots. The initial scenario is the same as before, but after we verify that the connection is up, we disconnect the server from the Ethernet, reboot it, and then reconnect it to the Ethernet. We expect the next keepalive probe to generate a reset from the server, because the server now knows nothing about this connection. Here is the interactive session:

```
bsdi % sock -K svr4 echo                -K to enable keepalive option
hi there                                type this to verify connection is up
hi there                                and this is echoed back from other end
                                        here server is rebooted while disconnected from Ethernet
read error: Connection reset by peer
```

Figure 23.2 shows the `tcpdump` output. (We have removed the connection establishment and the window advertisements.)

```
1     0.0                   bsdi.1057 > svr4.echo: P 1:10(9) ack 1
2     0.006406 (   0.0064)  svr4.echo > bsdi.1057: P 1:10(9) ack 10
3     0.176922 (   0.1705)  bsdi.1057 > svr4.echo: . ack 10

4  7200.067151 (7199.8902)  arp who-has svr4 tell bsdi
5  7200.069751 (   0.0026)  arp reply svr4 is-at 0:0:c0:c2:9b:26
6  7200.070468 (   0.0007)  bsdi.1057 > svr4.echo: . ack 10
7  7200.075050 (   0.0046)  svr4.echo > bsdi.1057: R 1135563275:1135563275(0)
```

Figure 23.2 Keepalive example when other host has crashed and rebooted.

We establish the connection and send 9 bytes of data from the client to the server (lines 1–3). Two hours later the first keepalive probe is sent by the client, and the response is a reset from the server. The client application prints the error "Connection reset by peer," which makes sense.

Other End Is Unreachable

In this example the client has not crashed, but is not reachable during the 10-minute period when the keepalive probes are sent. An intermediate router may have crashed, a phone line may be temporarily out of order, or something similar.

To simulate this example we'll establish a TCP connection from our host `slip` through our dialup SLIP link to the host `vangogh.cs.berkeley.edu`, and then take the link down. First, here is the interactive output:

```
slip % sock -K vangogh.cs.berkeley.edu echo
testing                                  we type this line
testing                                  and see it echoed
                                         sometime in here the dialup SLIP link is taken down
read error: No route to host
```

Figure 23.3 shows the `tcpdump` output that was collected on the router `bsdi`. (The connection establishment and window advertisements have been removed.)

```
 1      0.0                  slip.1056 > vangogh.echo: P 1:9(8) ack 1
 2      0.277669 (   0.2777) vangogh.echo > slip.1056: P 1:9(8) ack 9
 3      0.424423 (   0.1468) slip.1056 > vangogh.echo: . ack 9

 4   7200.818081 (7200.3937) slip.1056 > vangogh.echo: . ack 9
 5   7201.243046 (   0.4250) vangogh.echo > slip.1056: . ack 9

 6  14400.688106 (7199.4451) slip.1056 > vangogh.echo: . ack 9
 7  14400.689261 (   0.0012) sun > slip: icmp: net vangogh unreachable

 8  14475.684360 (  74.9951) slip.1056 > vangogh.echo: . ack 9
 9  14475.685504 (   0.0011) sun > slip: icmp: net vangogh unreachable

                             14 lines deleted

24  15075.759603 (  75.1008) slip.1056 > vangogh.echo: R 9:9(0) ack 9
25  15075.760761 (   0.0012) sun > slip: icmp: net vangogh unreachable
```

Figure 23.3 Keepalive example when other end is unreachable.

We start the example the same as before: lines 1–3 verify that the connection is up. The first keepalive probe 2 hours later is fine (lines 4 and 5), but before the next one occurs in another 2 hours, we bring down the SLIP connection between the routers `sun` and `netb`. (Refer to the inside front cover for the topology.)

The keepalive probe in line 6 elicits an ICMP network unreachable from the router `sun`. As we described in Section 21.10, this is just a soft error to the receiving TCP on the host `slip`. It records that the ICMP error was received, but the receipt of the error does not take down the connection. Eight more keepalive probes are sent, 75 seconds apart, before the sending host gives up. The error returned to the application generates a different message this time: "No route to host." We saw in Figure 6.12 (p. 82) that this corresponds to the ICMP network unreachable error.

23.4 Summary

As we said earlier, the keepalive feature is controversial. Protocol experts continue to debate whether it belongs in the transport layer, or should be handled entirely by the application.

It operates by sending a probe packet across a connection after the connection has been idle for 2 hours. Four different scenarios can occur: the other end is still there, the other end has crashed, the other end has crashed and rebooted, or the other end is currently unreachable. We saw each of these scenarios with an example, and saw different errors returned for the last three conditions.

In the first two examples that we looked at, had this feature not been provided, and without any application-level timer, our client would never have known that the other end had crashed, or crashed and rebooted. In the final example, however, nothing was wrong with the other end, the connection between them was temporarily down. We must be aware of this limitation when using keepalives.

Exercises

23.1 List some advantages of the keepalive feature.

23.2 List some disadvantages of the keepalive feature.