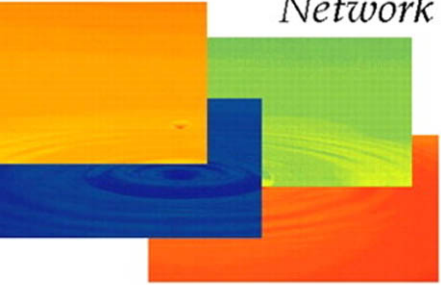
Effective TCP/IP Programming

44 Tips to Improve Your Network Programs



Jon C. Snader

Effective TCP/IP Programming

monitoring in the application, it can be fine-tuned to meet the needs of the application and to work with the application protocol as seamlessly as possible.

Keep-Alives

TCP does, in fact, have a mechanism, called *keep-alives*, for detecting dead connections, but as we shall see it is often not useful to applications. When an application enables the keep-alive mechanism, TCP sends a special segment to its peer when the connection has been idle for a certain interval. If the peer host is reachable and the peer application is still running, the peer TCP responds with an ACK. In this case, the TCP sending the keep-alive resets the idle time to zero, and the application receives no notification that the exchange took place.

If the peer host is reachable but the peer application is not running, the peer TCP responds with an RST, and the TCP sending the keep-alive drops the connection and returns an `ECONNRESET` error to the application. This is normally the result of the peer host crashing and being rebooted, because, as discussed in Tip 9, if the peer application merely terminated or crashed, the peer TCP would have sent a FIN.

If the peer host does not respond with either an ACK or an RST, the TCP sending the keep-alive continues to send keep-alive probes until it decides that its peer is unreachable or has crashed. At that point it drops the connection and informs the application with an `ETIMEDOUT` or, if a router has returned an ICMP host or network unreachable error, with an `EHOSTUNREACH` or `ENETUNREACH` error.

The first problem with keep-alives for applications that need immediate notification of connectivity loss is the time intervals involved. RFC 1122 [Braden 1989] requires that if a TCP implements keep-alives, it must have a default idle time of at least 2 hours before it starts sending keep-alive probes. Then, because the ACK from its peer is not delivered reliably, it must send repeated probes before abandoning the connection. The 4.4BSD implementation sends nine probes spaced 75 seconds apart before dropping the connection.

This is an implementation detail. RFC 1122 does not specify how many or how often probes should be sent before dropping the connection, only that an implementation must not interpret the lack of a response to any single probe as an indication of a dead connection.

This means that BSD-derived implementations using the default values take 2 hours, 11 minutes, 15 seconds to discover that connectivity has been lost. This value makes more sense when we realize that keep-alives are intended to release the resources held by defunct connections. Such connections can come about, for example, when a client connects to a server and the client's host crashes. Without the keep-alive mechanism, the server waits forever for the client's next request, because it never receives a FIN.

This situation is increasingly common due to users of PC-based systems merely turning off the computer or its modem instead of shutting down applications correctly.

Some implementations allow one or both time intervals to be changed, but this is almost always on a systemwide basis. That is, the change affects *all* TCP connections on the system. This is the main reason that keep-alives are not really useful as a connection monitoring mechanism: The default time periods are too long, and if the default is changed, they are no longer useful for their original purpose of cleaning up long-dead connections.

There is a new POSIX socket option, `TCP_KEEPALIVE`, which does allow the time-out interval to be specified on a per-connection basis, but it is not widely implemented.

The other problem with keep-alives is that they don't merely detect dead connections, they drop them. This may or may not be what the application wants.

Heartbeats

The problems with using keep-alives to monitor connectivity are easily solved by implementing a similar mechanism in the application. The best method of doing this depends on the application, and it is here that we see the flexibility that providing this mechanism in the application provides. As examples, let us consider two extreme cases:

1. A client and server that exchange several different types of messages, each having a header that identifies the message type.
2. An application that provides data to its peer as a byte stream with no inherent notion of record or message.

The first case is relatively easy. We introduce a new message type, `MSG_HEARTBEAT`, that one side can send to another. On receipt of the `MSG_HEARBEAT` message, the application merely returns the message to its peer. As we shall see, this allows us great latitude. One or both sides of the connection can monitor it for connectivity, with only one side actually sending the heartbeat.

First we look at the header file that is used by both client and server (Figure 2.46).

heartbeat.h
```
 1 #ifndef __HEARTBEAT_H__
 2 #define __HEARTBEAT_H__

 3 #define MSG_TYPE1       1       /* application specific msg */
 4 #define MSG_TYPE2       2       /* another one */
 5 #define MSG_HEARTBEAT   3       /* heartbeat message */

 6 typedef struct                  /* message structure */
 7 {
 8     u_int32_t type;             /* MSG_TYPE1, ... */
 9     char data[ 2000 ];
10 } msg_t;

11 #define T1              60      /* idle time before heartbeat */
12 #define T2              10      /* time to wait for response */

13 #endif  /* __HEARTBEAT_H__ */
```
heartbeat.h

Figure 2.46 The heartbeat header file

3–5 These manifest constants define the various message types that the client and server exchange. Only the `MSG_HEARTBEAT` message is meaningful in the example.

6–10 This typedef defines the structure of the messages passed between the client and the server. Again, only the `type` field is used in the example. A real application would customize this structure to fit its needs. See the remarks discussing Figure 2.31 for the meaning of `u_int32_t` and the dangers of making assumptions about the packing of structures.

11 This constant defines the amount of time that the connection can be idle before the client sends a heartbeat to its peer. We have arbitrarily chosen 60 seconds, but a real application would have to choose a value based on its needs and on the type of network.

12 The other timer value is the amount of time that the client waits for a response to its heartbeat message.

Next, we show the client side (Figure 2.47), which initiates the heartbeat. This choice is completely arbitrary and we could have just as easily chosen the server as the initiator.

hb_client.c

```
 1 #include "etcp.h"
 2 #include "heartbeat.h"

 3 int main( int argc, char **argv )
 4 {
 5     fd_set allfd;
 6     fd_set readfd;
 7     msg_t msg;
 8     struct timeval tv;
 9     SOCKET s;
10     int rc;
11     int heartbeats = 0;
12     int cnt = sizeof( msg );

13     INIT();
14     s = tcp_client( argv[ 1 ], argv[ 2 ] );
15     FD_ZERO( &allfd );
16     FD_SET( s, &allfd );
17     tv.tv_sec = T1;
18     tv.tv_usec = 0;
19     for ( ;; )
20     {
21         readfd = allfd;
22         rc = select( s + 1, &readfd, NULL, NULL, &tv );
23         if ( rc < 0 )
24             error( 1, errno, "select failure" );
25         if ( rc == 0 )      /* timed out */
26         {
27             if ( ++heartbeats > 3 )
28                 error( 1, 0, "connection dead\n" );
29             error( 0, 0, "sending heartbeat #%d\n", heartbeats );
30             msg.type = htonl( MSG_HEARTBEAT );
31             rc = send( s, ( char * )&msg, sizeof( msg ), 0 );
32             if ( rc < 0 )
33                 error( 1, errno, "send failure" );
34             tv.tv_sec = T2;
35              continue;
36          }
37          if ( !FD_ISSET( s, &readfd ) )
38              error( 1, 0, "select returned invalid socket\n" );
39          rc = recv( s, ( char * )&msg + sizeof( msg ) - cnt,
40              cnt, 0 );
41          if ( rc == 0 )
42              error( 1, 0, "server terminated\n" );
43          if ( rc < 0 )
44              error( 1, errno, "recv failure" );
45          heartbeats = 0;
46          tv.tv_sec = T1;
47          cnt -= rc;                  /* in-line readn */
48          if ( cnt > 0 )
49              continue;
50          cnt = sizeof( msg );

51          /* process message */
52      }
53  }
```
hb_client.c

Figure 2.47 A message-based client with a heartbeat

Initialization

13–14 We perform our standard initialization and connect to the server at the host and port specified on the command line.

15–16 We set up the select mask for our connect socket.

17–18 We initialize our timer to T1 seconds. If we don't receive a message within T1 seconds, `select` returns with a timer expiration.

21–22 We set the read select mask and then block in the call to `select` until we have data on the socket or until the timer expires.

Handle Timeout

27–28 If we have sent more than three consecutive heartbeats without an answer, we declare the connection dead. In this example, we merely quit, but a real application could take whatever steps it deemed appropriate.

29–33 If we haven't yet exhausted our heartbeat probes, we send our peer a heartbeat.

34–35 We set the timer to T2 seconds. If we don't receive a message from our peer within this amount of time, we either send another heartbeat or declare the connection dead depending on the value of `heartbeats`.

Process Message

37–38 If `select` returns any socket other than the one connected to our peer, we quit with a fatal error.

39–40 We call `recv` to read up to one message. These lines and the code below that housekeeps `cnt` are essentially an in-line version of `readn`. We can't call `readn` directly because it could block indefinitely disabling our heartbeat timing.

41–44 If we get either an EOF or a read error, we output a diagnostic and quit.

45–46 Because we have just heard from our peer, we set the heartbeat count back to zero and reset the timer for T1 seconds.

47–50 This code completes the in-line readn. We decrement cnt by the amount of data just read. If there is more data to be read, we continue at the call to select. Otherwise, we reset cnt for a full message and finish processing the message we have just read.

Figure 2.48 shows the server for our example. We have chosen to have the server monitor the connection also, but that is not strictly necessary.

hb_server.c

```
#include "etcp.h"
#include "heartbeat.h"

int main( int argc, char **argv )
{
    fd_set allfd;
    fd_set readfd;
    msg_t msg;
    struct timeval tv;
    SOCKET s;
    SOCKET s1;
    int rc;
    int missed_heartbeats = 0;
    int cnt = sizeof( msg );

    INIT();
    s = tcp_server( NULL, argv[ 1 ] );
    s1 = accept( s, NULL, NULL );
    if ( !isvalidsock( s1 ) )
        error( 1, errno, "accept failed" );
    tv.tv_sec = T1 + T2;
    tv.tv_usec = 0;
    FD_ZERO( &allfd );
    FD_SET( s1, &allfd );
    for ( ;; )
    {
        readfd = allfd;
        rc = select( s1 + 1, &readfd, NULL, NULL, &tv );
        if ( rc < 0 )
            error( 1, errno, "select failure" );
        if ( rc == 0 )      /* timed out */
        {
            if ( ++missed_heartbeats > 3 )
                error( 1, 0, "connection dead\n" );
            error( 0, 0, "missed heartbeat #%d\n",
                missed_heartbeats );
            tv.tv_sec = T2;
            continue;
        }
        if ( !FD_ISSET( s1, &readfd ) )
            error( 1, 0, "select returned invalid socket\n" );
        rc = recv( s1, ( char * )&msg + sizeof( msg ) - cnt,
            cnt, 0 );
        if ( rc == 0 )
            error( 1, 0, "client terminated\n" );
        if ( rc < 0 )
            error( 1, errno, "recv failure" );
46          missed_heartbeats = 0;
47          tv.tv_sec = T1 + T2;
48          cnt -= rc;              /* in-line readn */
49          if ( cnt > 0 )
50              continue;
51          cnt = sizeof( msg );
52          switch ( ntohl( msg.type ) )
53          {
54              case MSG_TYPE1 :
55                  /* process type1 message */
56                  break;

57              case MSG_TYPE2 :
58                  /* process type2 message */
59                  break;

60              case MSG_HEARTBEAT :
61                  rc = send( s1, ( char * )&msg, sizeof( msg ), 0 );
62                  if ( rc < 0 )
63                      error( 1, errno, "send failure" );
64                  break;

65              default :
66                  error( 1, 0, "unknown message type (%d)\n",
67                      ntohl( msg.type ) );
68          }
69      }
70      EXIT( 0 );
71  }
```
hb_server.c

Figure 2.48 A message-based server with a heartbeat

Initialization

14–18 We perform our standard initializations and accept a connection from our peer.

19–20 We set our initial timer value to T1 + T2 seconds. Because our peer sends a hearbeat after T1 seconds of inactivity, we allow a little extra time by increasing our timeout value by T2 seconds.

21–22 Next, we initialize our select mask for readability on the socket connected to our peer.

25–28 We call `select` and deal with any error returns.

Handle Timeout

31–32 If we have missed more than three consecutive heartbeats, we declare the connection dead and quit. As with the client, the server could take any other appropriate action at this point.

35 The timer is reset to T2 seconds. By now, the client should be sending us heartbeats every T2 seconds, and we time out after that amount of time so that we can count the missed heartbeats.

Process Message

38–39 We make the same invalid socket check that we did in the client.