How Expensive is TCP's Keep-Alive Feature?

Gene Michael Stover

created Saturday, 25 October 2003
updated Thursday, 30 October 2003

Contents

1 Introduction

A friend & programmer once told me that TCP's Keep Alive feature consumed too much bandwidth to be applicable to the application we were writing at the time. I didn't have any performance statistics on TCP's Keep Alive feature at the time, but I've finally gotten around to doing some experiments of my own. Here's what I did & the results I found.

2 The Test Procedure

To compare the performance of TCP with & without Keep Alive enabled, I wrote two programs: a client & a server.

The server opens two sockets on randomly selected TCP ports. It prints those ports to standard output. On the first port, it explicitly disables Keep Alive, which probably doesn't alter the socket's behaviour from the default. On the second port, it explicitly enables Keep Alive.

The client program needs to know the port number for the server, the number of sockets to create, & whether or not to enable Keep Alive before connecting to the server. It learns these things from command line options. The client creates all the sockets, enables Keep Alive on all of them if the command line options indicate to do so, & then connects to the server. Then it sends a bunch of packets over one of the sockets, allowing the others to idle so they will send TCP Keep Alive packets. The client counts the packets & tracks the time, then prints the performance statistics.

The client exits after sending all its packets, but the same server can be used for many client runs.

I ran the client for many combinations of "number of sockets" and "amount of time to run". I ran them from a shell script called `bin/speed`, saved the output to a temporary file, & then copied the temporary file into this document.[1]

The source code for the server, the client, & the shell script is in

- keepalive-1.cpio.bz2[2], which can be extracted by piping it through "`bzcat |cpio`", or
- keepalive-1.tar.gz[3], which can be extracted by piping it through "`gzcat |tar xf -`".

You just need one of those archives. Their conents are identical; only the archive formats differ.

I ran the client on Linux 2.2 on a 450 MHz Pentium something-or-other. I ran the server on Linux 2.2 on a 750 MHz Pentium something-or-other. The two computers were on a 100 megabit-per-second Ethernet LAN. Neither computer was doing anything strenuous at the time other than this test.

3 Raw Data

Here are the raw, unsorted, unanalyzed results from a run of the `bin/speed` program. From its output, I removed the lines that were not table rows for LaTeX. I also numbered them. Other than that, it's just a table of what happened, unaltered, unsorted, un-anything. We'll analyze it next.

[1] I had to do some minor edits to massage the output into a LaTeX table. To do some operations on the data, I had to edit it again into a data structure for Lisp. If I did it again, I think it would have been easier for the client program to write data in a format that Lisp could read trivially. I'd copy the output into Lisp & use some functions to produce the different tables for LaTeX.

[2] http://lisp-p.org/tka/keepalive-1.cpio.bz2

[3] http://lisp-p.org/tka/keepalive-1.tar.gz

trial id	keep-alive?	sockets	packets written	seconds	packet/second	select max
1	no	1	197918	17	1.164e+04	1
2	yes	1	194373	17	1.143e+04	1
3	no	1	195081	17	1.148e+04	1
4	yes	1	195130	17	1.148e+04	1
5	no	1	190313	17	1.119e+04	1
6	yes	1	192565	17	1.133e+04	1
7	no	7	189586	17	1.115e+04	1
8	yes	7	200820	17	1.181e+04	1
9	no	7	206591	17	1.215e+04	1
10	yes	7	204046	17	1.200e+04	1
11	no	7	204298	17	1.202e+04	1
12	yes	7	200702	17	1.181e+04	1
13	no	17	199213	17	1.172e+04	1
14	yes	17	192651	17	1.133e+04	1
15	no	17	198799	17	1.169e+04	1
16	yes	17	194221	17	1.142e+04	1
17	no	17	190112	17	1.118e+04	1
18	yes	17	195298	17	1.149e+04	1
19	no	51	172105	17	1.012e+04	1
20	yes	51	134480	17	7.911e+03	1
21	no	51	172830	17	1.017e+04	1
22	yes	51	131408	17	7.730e+03	1
23	no	51	176926	17	1.041e+04	1
24	yes	51	133850	17	7.874e+03	1
25	no	1	1521564	125	1.217e+04	1
26	yes	1	1504186	125	1.203e+04	1
27	no	1	1482259	125	1.186e+04	1
28	yes	1	1510831	125	1.209e+04	1
29	no	1	1516938	125	1.214e+04	1
30	yes	1	1518025	125	1.214e+04	1
31	no	7	1486686	125	1.189e+04	1
32	yes	7	1442189	125	1.154e+04	1
33	no	7	1458905	125	1.167e+04	1
34	yes	7	1459590	125	1.168e+04	1
35	no	7	1443856	125	1.155e+04	1
36	yes	7	1466546	125	1.173e+04	1
37	no	17	1418704	125	1.135e+04	1
38	yes	17	1375353	125	1.100e+04	1
39	no	17	1400023	125	1.120e+04	1
40	yes	17	1381440	125	1.105e+04	1
41	no	17	1444769	125	1.156e+04	1
42	yes	17	1363319	125	1.091e+04	1
43	no	51	1292572	125	1.034e+04	1
44	yes	51	1156970	125	9.256e+03	1
45	no	51	1292853	125	1.034e+04	1
46	yes	513	1001461	125	8.012e+03	1
47	no	51	1281842	125	1.025e+04	1
48	yes	51	1004165	125	8.033e+03	1
49	no	1	4293899	361	1.189e+04	1
50	yes	1	4303670	361	1.192e+04	1
51	no	1	4265132	361	1.181e+04	1
52	yes	1	4228313	361	1.171e+04	1
53	no	1	4279627	361	1.185e+04	1
54	yes	1	4218255	361	1.168e+04	1
55	no	7	4278727	361	1.185e+04	1
56	yes	7	4305502	361	1.193e+04	1
57	no	7	4268443	361	1.182e+04	1

keep-alive?	sockets	packets	seconds	rate (p/s)
NO	1	42353431	3519	1.204E+4
YES	1	42128709	3519	1.197E+4
NO	7	41571530	3519	1.181E+4
YES	7	41869223	3519	1.190E+4
NO	17	40338885	3519	1.146E+4
YES	17	40112719	3519	1.140E+4
NO	51	36194404	3519	1.029E+4
YES	51	30577331	3519	8.689E+3

Figure 1: A table of the trials, reduced according to keep-alive setting & number of sockets.

4 Analysis

4.1 Consolidate the trials

For each combination of keep-alive, sockets, & seconds, I ran three trials. Maybe that was over-kill. To reduce the number of rows, let's consolidate them according to keep-alive, & sockets. The new table will have a column for the total number of packets sent by all trials for that keep-alive setting & that number of sockets. It'll also have a column for the entire number of seconds spent by those trials & for the cumulative rate in packets-per-second, of all those trials.

For example, we take all the trials for no keep-alive & one socket, & we create one row for them. The trials with no keep-alive & one socket are 1, 3, 5, 25, 27, 29, 49, 51, 53, 73, 75, & 77. We sum those trials to produce a row for no keep-alive, one socket, in which the total number of packets sent is 42353431, the total number of seconds spent sending them is 3519 second, & the rate is $1.204 \times 10^4 \frac{packet}{s}$.

The new table is in Figure 1.

Notice that the rate with keep-alive enabled is almost always slower than the rate without it, but with seven sockets, the rate with keep-alive was actually a little faster. I presume keep-alive didn't actually increase the rate. This suggests that the bandwidth consumed by keep-alive data was less than the bandwidth consumed by some factor I didn't measure. If so, that means that, with seven or so sockets, keep-alive will affect an application's performance a lot less than other inefficiencies in your application might. In other words, it's an argument that keep-alive is fast enough, at least for about seven sockets or less.

4.2 Compare the rates

Let's make a new table to compare the rates between the keep-alive settings for a given number of sockets. The new table has a row for each number of sockets (1, 7, 17, & 51). It shows the number of packets sent by all the non-keep-alive trials for that number of sockets, the number of packets sent by the keep-alive trials for that number of sockets, & the ratio between the keep-alive packet

sockets	packets with KA	packets w/o KA	relative w/o
1	42353431	42128709	0.995
7	41571530	41869223	1.007
17	40338885	40112719	0.994
51	36194404	30577331	0.845

Figure 2: A table comparing the rate of sending packets when keep-alive is enabled to the rate of sending packets when keep-alive is disabled.

count & the non-keep-alive packet count. (We can drop the number of seconds because it is always the same.) The new table is in Figure 2.

From Figure 2, it looks like the cost of keep-alive is negligible until somewhere after 17 sockets. Remember that these tests were a sort of worst case in that one socket was writing continually while the other sockets idled. If the other sockets had been transmitting data, they might not have been sending keep-alive packets, so the cost of keep-alive in that case would have been less.

Nevertheless, once we reach 51 sockets, with one of them sending & the other 50 idling, the cost of keep-alive is over 15 percent, which could be significant for many applications.

5 Alternatives to Keep Alive

Is TCP's Keep Alive feature ever necessary? No. In normal use, even without keep-alive enabled, you'll be notified that a connection is closed or broken if you try to write to your socket for that connection. So if you are going to write to a socket continuously or frequently during the life of the connection, you don't need keep-alive on that socket at all.

What if you are only going to read from the socket? If the connection breaks or the other process closes its end for whatever reason, you won't learn about it unless you have keep-alive enabled or you try to write to that socket.

You can implement a keep-alive feature at the application level in a way that is more acceptable to your application's performance requirements. It works well if your appliction protocol over the TCP connection has a concept of packets (or message or frame or whatever you want to call it). Create a Null packet type that the receiver is supposed to ignore & can accept at any point in the conversation. When one of the conversants notices that it hasn't received any packets from the other end in T time, it should send a Null packet. If the connection has been lost, that process will hear about it soon after that; otherwise, the connection is up, the process which receives the Null packet will happily ignore it, & communication can continue. You can choose the size of T to suit your application. It could be seconds, minutes, fortnights, or whatever else is appropriate. In fact, the keep-alive feature simply does this for you at the TCP level so you don't need to think about it, & with a timer of about 45 seconds (**??**, page 40).

1. Send your Null packet over the socket.

2. In your event loop, include the socket in the set of readers you give to `select`.

3. If the socket is closed, `select` will soon give you a *read* indication on it. When you try to read from that socket, you'll get 0 bytes. (That read operation will be non-blocking.) You can close the socket.

Figure 3: Pseudocode that integrates a connection test with an event loop

You could turn the problem inside-out & use a lazy approach. Instead of testing connections at regular intervals, decide how many extant sockets your program can afford. Call this number N, the maximum Number of extant sockets. It might be ten or 50 or 100 or whatever floats your boat as long as it is less than the maximum number of sockets your operating system can deliver to a single process. Whenever a server accepts a new connection & the number of sockets in use is at least N, scan the list of existing sockets & send Null packets to test them. Close any that are no longer connected. The client (or whatever you want to call the program that originates the connections) might still need to use a timer.

There are twists on this technique. When it's time to test the connections, instead of testing all sockets, the server might test only those that haven't been used in T time. Another way to avoid scanning the entire list of sockets is to test & close until the number of extant sockets is less than N. If there is a chance you won't scan the entire list of sockets, it might be a good idea to sort them, starting from least recently used or the oldest. Otherwise, you might end up with a socket whose file descriptor is very large & whose connection is dead, but you never scan it because it's last on your list of sockets to scan, & you always manage to get the number of extant sockets to less than N before you read this high-numbered socket.

Testing a socket for a dead connection is not a very expensive operation. Send your Null packet over the socket, then try to read from the socket. If the connection is closed or dropped, `read` will return 0, & you can close the socket.[4] Pseudocode for a more event-driven model is in Figure 3.

6 Conclusion

Is TCP's Keep Alive feature too slow? That's impossible to answer without defining "too slow" for a particular application, but it looks like at about ten connections or less, it consumes less than five percent of the bandwidth, but at about 50 connections, it consumes over 15 percent. My gut feeling is that five

[4] Closed connections, dropped connections, end-of-file, & a few errors are the only situations in which `read` returns zero.

percent isn't too much to ask from most applications, but fifteen percent is a lot. Does your application expect ten simultanous connections or fifty?

An alternative to TCP's Keep Alive feature is a similar feature implemented at the application level. Since it is simple to implement for many applications, & its performance cost can be tailored to the application's performance requirements, & the performance cost can be very low (by using a lazy technique), a Keep Alive feature at the application level may be a better choice than TCP's Keep Alive feature.

7 Other File Formats

This document is available online in several formats:

- HTML is at http://lisp-p.org/tka/.
- PostScript is http://lisp-p.org/tka/tka.ps.
- DVI is http://lisp-p.org/tka/tka.dvi.

There are no plans to make it available in Pointless Document Format (PDF).

The source files are also available online:

- http://lisp-p.org/tka/keepalive-1.cpio.bz2, 117982 bytes
- http://lisp-p.org/tka/keepalive-1.tar.gz, 176142 bytes

References