TCP Keepalive HOWTO

Fabio Busatto

<fabio.busatto@sikurezza.org>

2007−05−04

Revision History

Revision 1.0	2007−05−04	Revised by: FB
First release, reviewed by TM.		

This document describes the TCP keepalive implementation in the linux kernel, introduces the overall concept and points to both system configuration and software development.

Table of Contents

1. Introduction

Understanding TCP keepalive is not necessary in most cases, but it's a subject that can be very useful under particular circumstances. You will need to know basic TCP/IP networking concepts, and the C programming language to understand all sections of this document.

The main purpose of this HOWTO is to describe TCP keepalive in detail and demonstrate various application situations. After some initial theory, the discussion focuses on the Linux implementation of TCP keepalive routines in the modern Linux kernel releases (2.4.x, 2.6.x), and how system administrators can take advantage of these routines, with specific configuration examples and tricks.

The second part of the HOWTO involves the programming interface exposed by the Linux kernel, and how to write TCP keepalive−enabled applications in the C language. Pratical examples are presented, and there is an introduction to the `libkeepalive` project, which permits legacy applications to benefit from keepalive with no code modification.

1.1. Copyright and License

1.2. Disclaimer

No liability for the contents of this document can be accepted. Use the concepts, examples and information at your own risk. There may be errors and inaccuracies that could be damaging to your system. Proceed with caution, and although this is highly unlikely, the author does not take any responsibility.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark. Naming of particular products or brands should not be seen as endorsements.

1.3. Credits / Contributors

This work is not especially related to any people that I should thank. But my life is, and my knowledge too: so, thanks to everyone that has supported me, prior to my birth, now, and in the future. Really.

A special thank is due to Tabatha, the patient woman that read my work and made the needed reviews.

1.4. Feedback

Feedback is most certainly welcome for this document. Send your additions, comments and criticisms to the following email address: `<fabio.busatto@sikurezza.org>`.

1.5. Translations

There are no translated versions of this HOWTO at the time of publication. If you are interested in translating this HOWTO into other languages, please feel free to contact me. Your contribution will be very welcome.

2. TCP keepalive overview

In order to understand what TCP keepalive (which we will just call keepalive) does, you need do nothing more than read the name: keep TCP alive. This means that you will be able to check your connected socket (also known as TCP sockets), and determine whether the connection is still up and running or if it has broken.

2.1. What is TCP keepalive?

The keepalive concept is very simple: when you set up a TCP connection, you associate a set of timers. Some of these timers deal with the keepalive procedure. When the keepalive timer reaches zero, you send your peer a keepalive probe packet with no data in it and the ACK flag turned on. You can do this because of the TCP/IP specifications, as a sort of duplicate ACK, and the remote endpoint will have no arguments, as TCP is a stream−oriented protocol. On the other hand, you will receive a reply from the remote host (which doesn't need to support keepalive at all, just TCP/IP), with no data and the ACK set.

If you receive a reply to your keepalive probe, you can assert that the connection is still up and running without worrying about the user−level implementation. In fact, TCP permits you to handle a stream, not packets, and so a zero−length data packet is not dangerous for the user program.

This procedure is useful because if the other peers lose their connection (for example by rebooting) you will notice that the connection is broken, even if you don't have traffic on it. If the keepalive probes are not replied to by your peer, you can assert that the connection cannot be considered valid and then take the correct action.

2.2. Why use TCP keepalive?

You can live quite happily without keepalive, so if you're reading this, you may be trying to understand if keepalive is a possible solution for your problems. Either that or you've really got nothing more interesting to do instead, and that's okay too. :)

Keepalive is non−invasive, and in most cases, if you're in doubt, you can turn it on without the risk of doing something wrong. But do remember that it generates extra network traffic, which can have an impact on routers and firewalls.

In short, use your brain and be careful.

In the next section we will distinguish between the two target tasks for keepalive:

- Checking for dead peers
- Preventing disconnection due to network inactivity

2.3. Checking for dead peers

Keepalive can be used to advise you when your peer dies before it is able to notify you. This could happen for several reasons, like kernel panic or a brutal termination of the process handling that peer. Another scenario that illustrates when you need keepalive to detect peer death is when the peer is still alive but the network channel between it and you has gone down. In this scenario, if the network doesn't become operational again, you have the equivalent of peer death. This is one of those situations where normal TCP operations aren't useful to check the connection status.

Think of a simple TCP connection between Peer A and Peer B: there is the initial three−way handshake, with one SYN segment from A to B, the SYN/ACK back from B to A, and the final ACK from A to B. At this time, we're in a stable status: connection is established, and now we would normally wait for someone to send data over the channel. And here comes the problem: unplug the power supply from B and instantaneously it will go down, without sending anything over the network to notify A that the connection is going to be broken. A, from its side, is ready to receive data, and has no idea that B has crashed. Now restore the power supply to B and wait for the system to restart. A and B are now back again, but while A knows about a connection still active with B, B has no idea. The situation resolves itself when A tries to send data to B over the dead connection, and B replies with an RST packet, causing A to finally to close the connection.

Keepalive can tell you when another peer becomes unreachable without the risk of false−positives. In fact, if the problem is in the network between two peers, the keepalive action is to wait some time and then retry, sending the keepalive packet before marking the connection as broken.

```
    _____                                                     _____
   |     |                                                   |     |
   |  A  |                                                   |  B  |
   |_____|                                                   |_____|
      ^                                                         ^
      |--->--->--->-------------- SYN -------------->--->--->---|
      |---<---<---<------------ SYN/ACK ------------<---<---<---|
      |--->--->--->-------------- ACK -------------->--->--->---|
      |                                                         |
      |                                       system crash ---> X
      |
      |                                     system restart ---> ^
      |                                                         |
      |--->--->--->-------------- PSH -------------->--->--->---|
      |---<---<---<-------------- RST --------------<---<---<---|
      |                                                         |
```

2.4. Preventing disconnection due to network inactivity

The other useful goal of keepalive is to prevent inactivity from disconnecting the channel. It's a very common issue, when you are behind a NAT proxy or a firewall, to be disconnected without a reason. This behavior is caused by the connection tracking procedures implemented in proxies and firewalls, which keep track of all connections that pass through them. Because of the physical limits of these machines, they can only keep a finite number of connections in their memory. The most common and logical policy is to keep newest connections and to discard old and inactive connections first.

Returning to Peers A and B, reconnect them. Once the channel is open, wait until an event occurs and then communicate this to the other peer. What if the event verifies after a long period of time? Our connection has its scope, but it's unknown to the proxy. So when we finally send data, the proxy isn't able to correctly handle it, and the connection breaks up.

Because the normal implementation puts the connection at the top of the list when one of its packets arrives and selects the last connection in the queue when it needs to eliminate an entry, periodically sending packets over the network is a good way to always be in a polar position with a minor risk of deletion.

```
    _____           _____                                     _____
   |     |         |     |                                   |     |
   |  A  |         | NAT |                                   |  B  |
```

```
|_____|         |_____|                                   |_____|
   ^               ^                                         ^
   |--->--->--->---|----------- SYN ------------->--->--->---|
   |---<---<---<---|--------- SYN/ACK -----------<---<---<---|
   |--->--->--->---|----------- ACK ------------->--->--->---|
   |               |                                         |
   |               | <--- connection deleted from table      |
   |               |                                         |
   |--->- PSH ->---| <--- invalid connection                 |
   |               |                                         |
```

3. Using TCP keepalive under Linux

Linux has built−in support for keepalive. You need to enable TCP/IP networking in order to use it. You also need `procfs` support and `sysctl` support to be able to configure the kernel parameters at runtime.

The procedures involving keepalive use three user−driven variables:

`tcp_keepalive_time`
: the interval between the last data packet sent (simple ACKs are not considered data) and the first keepalive probe; after the connection is marked to need keepalive, this counter is not used any further

`tcp_keepalive_intvl`
: the interval between subsequential keepalive probes, regardless of what the connection has exchanged in the meantime

`tcp_keepalive_probes`
: the number of unacknowledged probes to send before considering the connection dead and notifying the application layer

Remember that keepalive support, even if configured in the kernel, is not the default behavior in Linux. Programs must request keepalive control for their sockets using the `setsockopt` interface. There are relatively few programs implementing keepalive, but you can easily add keepalive support for most of them following the instructions explained later in this document.

3.1. Configuring the kernel

There are two ways to configure keepalive parameters inside the kernel via userspace commands:

- `procfs` interface
- `sysctl` interface

We mainly discuss how this is accomplished on the procfs interface because it's the most used, recommended and the easiest to understand. The sysctl interface, particularly regarding the `sysctl`(2) syscall and not the **sysctl**(8) tool, is only here for the purpose of background knowledge.

3.1.1. The `procfs` interface

This interface requires both `sysctl` and `procfs` to be built into the kernel, and `procfs` mounted somewhere in the filesystem (usually on `/proc`, as in the examples below). You can read the values for the actual parameters by "catting" files in `/proc/sys/net/ipv4/` directory:

```
# cat /proc/sys/net/ipv4/tcp_keepalive_time
  7200

  # cat /proc/sys/net/ipv4/tcp_keepalive_intvl
  75

  # cat /proc/sys/net/ipv4/tcp_keepalive_probes
  9
```

The first two parameters are expressed in seconds, and the last is the pure number. This means that the keepalive routines wait for two hours (7200 secs) before sending the first keepalive probe, and then resend it every 75 seconds. If no ACK response is received for nine consecutive times, the connection is marked as broken.

Modifying this value is straightforward: you need to write new values into the files. Suppose you decide to configure the host so that keepalive starts after ten minutes of channel inactivity, and then send probes in intervals of one minute. Because of the high instability of our network trunk and the low value of the interval, suppose you also want to increase the number of probes to 20.

Here's how we would change the settings:

```
# echo 600 > /proc/sys/net/ipv4/tcp_keepalive_time

  # echo 60 > /proc/sys/net/ipv4/tcp_keepalive_intvl

  # echo 20 > /proc/sys/net/ipv4/tcp_keepalive_probes
```

To be sure that all succeeds, recheck the files and confirm these new values are showing in place of the old ones.

Remember that `procfs` handles special files, and you cannot perform any sort of operation on them because they're just an interface within the kernel space, not real files, so try your scripts before using them, and try to use simple access methods as in the examples shown earlier.

You can access the interface through the **sysctl**(8) tool, specifying what you want to read or write.

```
# sysctl \
  > net.ipv4.tcp_keepalive_time \
  > net.ipv4.tcp_keepalive_intvl \
  > net.ipv4.tcp_keepalive_probes
  net.ipv4.tcp_keepalive_time = 7200
  net.ipv4.tcp_keepalive_intvl = 75
  net.ipv4.tcp_keepalive_probes = 9
```

Note that `sysctl` names are very close to `procfs` paths. Write is performed using the `-w` switch of **sysctl** (8):

```
# sysctl -w \
  > net.ipv4.tcp_keepalive_time=600 \
  > net.ipv4.tcp_keepalive_intvl=60 \
  > net.ipv4.tcp_keepalive_probes=20
  net.ipv4.tcp_keepalive_time = 600
  net.ipv4.tcp_keepalive_intvl = 60
  net.ipv4.tcp_keepalive_probes = 20
```

Note that **sysctl** (8) doesn't use `sysctl`(2) syscall, but reads and writes directly in the `procfs` subtree, so you will need `procfs` enabled in the kernel and mounted in the filesystem, just as you would if you directly accessed the files within the `procfs` interface. **Sysctl**(8) is just a different way to do the same thing.

3.1.2. The `sysctl` interface

There is another way to access kernel variables: `sysctl`(2) syscall. It can be useful when you don't have `procfs` available because the communication with the kernel is performed directly via syscall and not through the `procfs` subtree. There is currently no program that wraps this syscall (remember that **sysctl**(8) doesn't use it).

For more details about using `sysctl`(2) refer to the manpage.

3.2. Making changes persistent to reboot

There are several ways to reconfigure your system every time it boots up. First, remember that every Linux distribution has its own set of init scripts called by **init** (8). The most common configurations include the `/etc/rc.d/` directory, or the alternative, `/etc/init.d/`. In any case, you can set the parameters in any of the startup scripts, because keepalive rereads the values every time its procedures need them. So if you change the value of `tcp_keepalive_intvl` when the connection is still up, the kernel will use the new value going forward.

There are three spots where the initialization commands should logically be placed: the first is where your network is configured, the second is the `rc.local` script, usually included in all distributions, which is known as the place where user configuration setups are done. The third place may already exist in your system. Referring back to the **sysctl** (8) tool, you can see that the `-p` switch loads settings from the `/etc/sysctl.conf` configuration file. In many cases your init script already performs the **sysctl** `-p` (you can "grep" it in the configuration directory for confirmation), and so you just have to add the lines in `/etc/sysctl.conf` to make them load at every boot. For more information about the syntax of `sysctl.conf`(5), refer to the manpage.

4. Programming applications

This section deals with programming code needed if you want to create applications that use keepalive. This is not a programming manual, and it requires that you have previous knowledge in C programming and in networking concepts. I consider you familiar with sockets, and with everything concerning the general aspects of your application.

4.1. When your code needs keepalive support

Not all network applications need keepalive support. Remember that it is TCP keepalive support. So, as you can imagine, only TCP sockets can take advantage of it.

The most beautiful thing you can do when writing an application is to make it as customizable as possible, and not to force decisions. If you want to consider the happiness of your users, you should implement keepalive and let the users decide if they want to use it or not by using a configuration parameter or a switch on the command line.

4.2. The `setsockopt` function call

All you need to enable keepalive for a specific socket is to set the specific socket option on the socket itself. The prototype of the function is as follows:

```
  int setsockopt(int s, int level, int optname,
                 const void *optval, socklen_t optlen)
```

The first parameter is the socket, previously created with the `socket`(2); the second one must be `SOL_SOCKET`, and the third must be `SO_KEEPALIVE` . The fourth parameter must be a boolean integer value, indicating that we want to enable the option, while the last is the size of the value passed before.

According to the manpage, 0 is returned upon success, and −1 is returned on error (and `errno` is properly set).

There are also three other socket options you can set for keepalive when you write your application. They all use the `SOL_TCP` level instead of `SOL_SOCKET`, and they override system−wide variables only for the current socket. If you read without writing first, the current system−wide parameters will be returned.

- `TCP_KEEPCNT`: overrides `tcp_keepalive_probes`
- `TCP_KEEPIDLE`: overrides `tcp_keepalive_time`
- `TCP_KEEPINTVL`: overrides `tcp_keepalive_intvl`

4.3. Code examples

This is a little example that creates a socket, shows that keepalive is disabled, then enables it and checks that the option was effectively set.

```
            /* --- begin of keepalive test program --- */

#include <stdio.h>
```

```
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void);

int main()
{
   int s;
   int optval;
   socklen_t optlen = sizeof(optval);

   /* Create the socket */
   if((s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
      perror("socket()");
      exit(EXIT_FAILURE);
   }

   /* Check the status for the keepalive option */
   if(getsockopt(s, SOL_SOCKET, SO_KEEPALIVE, &optval, &optlen) < 0) {
      perror("getsockopt()");
      close(s);
      exit(EXIT_FAILURE);
   }
   printf("SO_KEEPALIVE is %s\n", (optval ? "ON" : "OFF"));

   /* Set the option active */
   optval = 1;
   optlen = sizeof(optval);
   if(setsockopt(s, SOL_SOCKET, SO_KEEPALIVE, &optval, optlen) < 0) {
      perror("setsockopt()");
      close(s);
      exit(EXIT_FAILURE);
   }
   printf("SO_KEEPALIVE set on socket\n");

   /* Check the status again */
   if(getsockopt(s, SOL_SOCKET, SO_KEEPALIVE, &optval, &optlen) < 0) {
      perror("getsockopt()");
      close(s);
      exit(EXIT_FAILURE);
   }
   printf("SO_KEEPALIVE is %s\n", (optval ? "ON" : "OFF"));

   close(s);

   exit(EXIT_SUCCESS);
}

            /* ---  end of keepalive test program  --- */
```

5. Adding support to third−party software

Not everyone is a software developer, and not everyone will rewrite software from scratch if it lacks just one feature. Maybe you want to add keepalive support to an existing application because, though the author might not have thought it important, you think it will be useful.

First, remember what was said about the situations where you need keepalive. Now you'll need to address connection−oriented TCP sockets.

Since Linux doesn't provide the functionality to enable keepalive support via the kernel itself (as BSD−like operating systems often do), the only way is to perform the `setsockopt` (2) call after socket creation. There are two solutions:

- source code modification of the original program
- `setsockopt` (2) injection using the library preloading technique

5.1. Modifying source code

Remember that keepalive is not program−related, but socket−related, so if you have multiple sockets, you can handle keepalive for each of them separately. The first phase is to understand what the program does and then search the code for each socket in the program. This can be done using **grep**(1), as follows:

```
# grep 'socket *(' *.c
```

This will more or less show you all sockets in the code. The next step is to select only the right ones: you will need TCP sockets, so look for `PF_INET` (or `AF_INET`), `SOCK_STREAM` and `IPPROTO_TCP` (or more commonly, `0`) in the parameters of your socket list, and remove the non−matching ones.

Another way to create a socket is through `accept`(2). In this case, follow the TCP sockets identified and check if any of these is a listening socket: if positive, keep in mind that `accept`(2) returns a socket descriptor, which must be inserted in your socket list.

Once you've identified the sockets you can proceed with changes. The most fast & furious patch can be done by simply adding the `setsockopt`(2) function just after the socket creation block. Optionally, you may include additional calls in order to set the keepalive parameters if you don't like the system defaults. Please be careful when implementing error checks and handlers for the function, maybe by copying the style from the original code around it. Remember to set the `optval` to a non−zero value and to initialize the `optlen` before invoking the function.

If you have time or you think it would be really cool, try to add complete keepalive support to your program, including a switch on the command line or a configuration parameter to let the user choose whether or not to use keepalive.

5.2. libkeepalive: library preloading

There are often cases where you don't have the ability to modify the source code of an application, or when you have to enable keepalive for all your programs, so patching and recompiling everything is not recommended.

The libkeepalive project was born to help add keepalive support for applications since the Linux kernel doesn't provide the ability to do the same thing natively (like BSD does). The libkeepalive project homepage is http://libkeepalive.sourceforge.net/

It consists of a shared library that overrides the socket system call in most binaries, without the need to recompile or modify them. The technique is based on the *preloading* feature of the **ld.so**(8) loader included in Linux, which allows you to force the loading of shared libraries with higher priority than normal. Programs usually use the `socket`(2) function call located in the `glibc` shared library; with libkeepalive you can wrap it and inject the `setsockopt` (2) just after the socket creation, returning a socket with keepalive already set to the main program. Because of the mechanisms used to inject the system call, this doesn't work when the socket function is statically compiled into the binary, as in a program linked with the **gcc**(1) flag `-static`.

After downloading and installing libkeepalive, you will able to add keepalive support to your programs without the prerequisite of being `root`, simply setting the `LD_PRELOAD` environment variable before executing the program. By the way, the superuser can also force the preloading with a global configuration, and the users can then decide to turn it off by setting the `KEEPALIVE` environment variable to `off`.

The environment is also used to set specific values for keepalive parameters, so you have the ability to handle each program differently, setting `KEEPCNT`, `KEEPIDLE` and `KEEPINTVL` before starting the application.

Here's an example of libkeepalive usage:

```
$ test
  SO_KEEPALIVE is OFF

  $ LD_PRELOAD=libkeepalive.so \
  > KEEPCNT=20 \
  > KEEPIDLE=180 \
  > KEEPINTVL=60 \
  > test
  SO_KEEPALIVE is ON
  TCP_KEEPCNT   = 20
  TCP_KEEPIDLE  = 180
  TCP_KEEPINTVL = 60
```

And you can use **strace** (1) to understand what happens:

```
$ strace test
  execve("test", ["test"], [/* 26 vars */]) = 0
  [..]
  open("/lib/libc.so.6", O_RDONLY)        = 3
  [..]
  socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
  getsockopt(3, SOL_SOCKET, SO_KEEPALIVE, [0], [4]) = 0
  close(3)                                = 0
  [..]
  _exit(0)                                = ?

  $ LD_PRELOAD=libkeepalive.so \
  > strace test
  execve("test", ["test"], [/* 27 vars */]) = 0
  [..]
  open("/usr/local/lib/libkeepalive.so", O_RDONLY) = 3
```

```
[..]
open("/lib/libc.so.6", O_RDONLY)        = 3
[..]
open("/lib/libdl.so.2", O_RDONLY)       = 3
[..]
socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
setsockopt(3, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
setsockopt(3, SOL_TCP, TCP_KEEPCNT, [20], 4) = 0
setsockopt(3, SOL_TCP, TCP_KEEPIDLE, [180], 4) = 0
setsockopt(3, SOL_TCP, TCP_KEEPINTVL, [60], 4) = 0
[..]
getsockopt(3, SOL_SOCKET, SO_KEEPALIVE, [1], [4]) = 0
[..]
getsockopt(3, SOL_TCP, TCP_KEEPCNT, [20], [4]) = 0
[..]
getsockopt(3, SOL_TCP, TCP_KEEPIDLE, [180], [4]) = 0
[..]
getsockopt(3, SOL_TCP, TCP_KEEPINTVL, [60], [4]) = 0
[..]
close(3)                                = 0
[..]
_exit(0)                                = ?
```

For more information, visit the libkeepalive project homepage: http://libkeepalive.sourceforge.net/