TCP and Linux' Pluggable Congestion Control Algorithms

By René Pfeiffer

Many months ago I had a few drinks with some fellow hackers. The discussion touched on the inevitable "my-Linux-is-better-than-your-*BSD-and-vice-versa" topic. I tried to argue in favour of Linux because it knows multiple TCP variants. They corrected me and said that there is only one TCP and that *BSD speaks it very well. Days after the side effects of the drinks were gone, I remembered the discussion and felt a sense of *l'esprit d'escalier*. Of course, there is a clear definition what the famous Tricky Communication Protocol (TCP) looks like, but there are many ways of dealing with situations in Wide Area Networks (WANs) where congestion, round-trip times, and packet loss play a major role. That's what I wanted to say. Enter Linux's pluggable congestion control algorithms.

Transmission Control Protocol (TCP)

Every time you wish to transport data in a reliable way, you probably use TCP (or your car, but let's stick to networks). TCP is able to transport large amounts of data in the correct order over unreliable network links. The protocol keeps track of sent data, detects lost packets, and retransmits them if necessary. This is done by acknowledging every packet to the sender. Basically, this is how TCP works. My article would end here were it not for some parameters of TCP connections that make things very interesting and complicated. (To some, those attributes are the same.) The first perturbation stems from the network itself, and materialises in the form of three parameters that pertain to every TCP connection.

1. Bandwidth
   The bandwidth indicates how many bits per time frame the link can transport. It is usually denoted by *Mbit/S* or *kbit/S* and limited by the hardware. Most people think that this indicates solely the performance of the network link. This is not quite true, as I will explain shortly.
2. Round-Trip Time (RTT)
   Consider a network link between points A and B. The time a packet needs to travel from A to B and then back to A is called the Round-Trip Time. The RTT is highly variable, especially when a node on the way experiences congestion. Typically, you have an RTT from milliseconds to seconds (in the worst case).
3. Packet loss
   Packets of a network transmission can get dropped. The ratio of lost packets to

transported packets is called packet loss. There are many reasons for packet loss. A router might be under heavy load, a frame might get corrupted by interference (wireless networks like to drop packets this way), or an Ethernet switch may detect a wrong checksum.

So, you see that a 1000 Mbit/s link will get you nowhere if it has 25% packet loss and an RTT of several seconds. The overall speed of your transmission depends on all three parameters. Your local Ethernet is (with luck) fine, because it has 100 Mbit/s, 0% packet loss and RTTs below 1 millisecond. As soon as the RTT rises, the management of the packets "in flight", i.e., the ones sent but not yet acknowledged by the receiver, determine the real throughput of the data. RTT and bandwidth are often combined and multiplied into the *bandwidth-delay product*.

How does TCP manage the parameter settings? Fortunately the protocol has some auto-tuning properties. One important parameter is the *TCP window*. The window is the amount of data the sender can send before receiving an acknowledgement. This means that, as long as the window is "not full", the sender can blindly keep sending packets. As soon as the window is filled, the sender stops sending, and waits for acknowledgement packets. This is TCP's main mechanism of flow control, that enables it to detect bottlenecks in the network during a transmission. That's why it's also called the *congestion window*. The default window size varies. Usually Linux starts at 32 kB. The window size is flexible, and can be changed during the transfer. This mechanism is called *window scaling*. Starting with Linux kernel 2.6.17, the window can be up to 4 MB. The window can grow only as long as there is no packet loss. That's one of the reasons why data throughput is reduced on lossy network links.

TCP has some more interesting features, but I won't go into more details. We now know enough, for the features of the Linux kernel I wish to describe.

Category: Protocols

TCP is able to transport large amounts of data in the correct order over unreliable network links. The protocol keeps track of sent data, detects lost packets, and retransmits them if necessary.

Always look on the WAN side of life

The right size of the TCP window is critical to efficient transmission. The tricky part is to find that right size. Either a too-small and a too-big window will degrade throughput. A good guess is the use the bandwidth-delay product. Furthermore, you can base estimates on periodically measured RTT or packet loss, and make it dynamic. This is why several researchers have explored algorithms to help TCP tune itself better, under certain circumstances. Beginning with 2.6.13, the Linux kernel supports plugins for the TCP stack, and enables switching between algorithms depending on what the system is connected to. The first strategy on the Internet was TCP Tahoe, proposed in 1988. A later variant was TCP Reno, now widely implemented in network stacks, and its successor TCP NewReno. Currently, the Linux kernel includes the following algorithms (as of kernel 2.6.19.1):

- High Speed TCP
  The algorithm is described in RFC 3649. The main use is for connections with large bandwidth and large RTT (such as Gbit/s and 100 ms RTT).
- H-TCP
  H-TCP was proposed by the Hamilton Institute for transmissions that recover more quickly after a congestion event. It is also designed for links with high bandwidth and RTT.
- Scalable TCP
  This is another algorithm for WAN links with high bandwidth and RTT. One of its design goals is a quick recovery of the window size after a congestion event. It achieves this goal by resetting the window to a higher value than standard TCP.
- TCP BIC
  BIC is the abbreviation for Binary Increase Congestion control. BIC uses a unique window growth function. In case of packet loss, the window is reduced by a multiplicative factor. The window size just before and after the reduction is then used as parameters for a binary search for the new window size. BIC was used as standard algorithm in the Linux kernel.
- TCP CUBIC
  CUBIC is a less aggressive variant of BIC (meaning, it doesn't steal as much throughput from competing TCP flows as does BIC).
- TCP Hybla
  TCP Hybla was proposed in order to transmit data efficiently over satellite links and "defend" the transmission against TCP flows from other origins.
- TCP Low Priority
  This is an approach to develop an algorithm that uses excess bandwidth for TCP flows. It can be used for low priority data transfers without "disturbing" other TCP transmissions (which probably don't use TCP Low Priority).
- TCP Tahoe/Reno
  These are the classical models used for congestion control. They exhibit the typical slow start of transmissions. The throughput increases gradually until it

stays stable. It is decreased as soon as the transfer encounters congestion, then the rate rises again slowly. The window is increased by adding fixed values. TCP Reno uses a multiplicative decrease algorithm for the reduction of window size. TCP Reno is the most widely deployed algorithm.

- TCP Vegas
  TCP Vegas introduces the measurement of RTT for evaluating the link quality. It uses additive increases and additive decreases for the congestion window.
- TCP Veno
  This variant is optimised for wireless networks, since it was designed to handle random packet loss better. It tries to keep track of the transfer, and guesses if the quality decreases due to congestion or random packet errors.
- TCP Westwood+
  Westwood+ addresses both large bandwidth/RTT values and random packet loss together with dynamically changing network loads. It analyses the state of the transfer by looking at the acknowledgement packets. Westwood+ is a modification of the TCP Reno algorithm.

This is only a rough outline of the modules. In case you want to understand what the algorithms do, you should read the authors' publications. I have given links to most of them at the end of this article. While investigating the algorithms and hunting for publications, I also came across CTCP, created by Microsoft Research Asia. I wonder if this will make its way into the Linux kernel, some day.

Switching between the different algorithms can be easily done, by writing text to a `/proc/` entry.

```
nightfall:~# echo "westwood" > /proc/sys/net/ipv4/tcp_congestion_control
nightfall:~# cat /proc/sys/net/ipv4/tcp_congestion_control
westwood
nightfall:~#
```

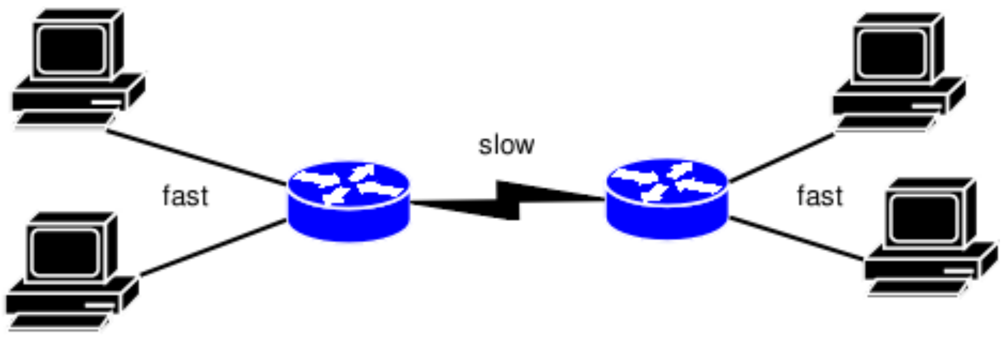
A list of available modules can be found here:

```
nightfall:~# ls /lib/modules/`uname -r`/kernel/net/ipv4/
ip_gre.ko  netfilter   tcp_cubic.ko      tcp_htcp.ko   tcp_lp.ko         tcp_vegas.ko
ipip.ko    tcp_bic.ko  tcp_highspeed.ko  tcp_hybla.ko  tcp_scalable.ko  tcp_veno.ko
nightfall:~#
```

When writing to `/proc/`, you can skip the `tcp_` prefix. If you compile your own kernels, you will find the modules in the *Networking -> Networking options -> TCP: advanced congestion control* section. Since some of the algorithms affect only the sender's side, you may not notice a difference when enabling them. In order to see changed behaviour, you have to create a controlled setup, and measure the parameters of TCP transmissions.

Testing congestion situations

Now that we know how to control the congestion algorithms, all we need is a bottleneck. As long as your Linux box(es) are connected to the local Ethernet with 100 Mbit/s or more, all packets will be sent immediately, and your local queues are always empty. The queues end up on your modem or gateway, which may treat them differently from the way Linux does. Even if you use a Linux gateway, there won't be a queue there, because very often this gateway connects via Ethernet to another device that handles the Internet connection. If you want to know what Linux does with queues, you need to set up a controlled bottleneck between two or more Linux boxes. Of course, you can try to saturate your LAN, but your core switch and your colleagues might not be impressed.



You can use two separate routers and connect them with serial crossover cables. You can set up two Linux boxes as routers, and create the bottleneck link with USB 1.1 crossover cabling, serial links, Bluetooth devices, 802.11b/g WLAN equipment, parallel links, or slower Ethernet cards (10 Mbit/s, for example). You could also do traffic shaping in-between, and reduce the bandwidth or increase the latency. The Linux kernel has a nice tool for doing this. It's called *netem*, which is short for Network Emulator. Recent distributions have it. In case you compile your own kernels, you'll find it here:

```
Networking -->
  Networking Options -->
    QoS and/or fair queuing -->
       Network emulator
```

Additionally, you need to enable the *Advanced Router* config option. netem can be activated by the `tc` tool from the *iproute* package. It allows you to

- emulate WAN links,
- create packet loss,
- create packet duplication,
- reorder packets as they are routed,
- and corrupt packets.

So, let's say you have a Linux router, and wish to simulate a WAN environment by increasing the RTT, adding packet reordering and some corruption of bits. You do this by entering the following commands:

```
tc qdisc add dev eth1 root netem corrupt 2.5% delay 50ms 10ms
tc qdisc add dev eth0 root netem delay 100ms reorder 25% 50%
```

This means that network device `eth1` corrupts 2.5% of all packets passing through it. Additionally, all packets get delayed by 50ms 10ms. Device `eth1` introduces a delay of at least 100 ms for reordered packets. 25% of the packets will be sent immediately. The 50% indicates a correlation between random reordering events. You can combine different netem options. Make sure that you enter only one `tc` command for every network device with all options and parameters attached. `netem` is a network queue, and can only be installed once for every network device. You can remove it any time you want, though.

Creating traffic and measuring TCP

After your new and shiny bottleneck works, you can start moving data from one end to the other. Every program that speaks TCP can be used for that. `iperf`, `netpipe`, any FTP client, `netcat`, or `wget` are good tools. Recording the network flows with `wireshark` and to postprocess them with `tcptrace` is also an option. `wireshark` has some features that allow you to analyse a TCP connection such as creating a graph of RTT values, sequence numbers, and throughput -- though a real evaluation for serious deployment must use a finer resolution and a better statistic than simple progress bars.

You can also do in-kernel measurement by using the module *tcpprobe*. It is available when the kernel is compiled with kernel probes that provide hooks to kernel functions (called Kprobe support). *tcpprobe* can be enabled by loading it with *modprobe* and giving a TCP port as module option. The documentation of *tcpprobe* features a simple example:

```
# modprobe tcpprobe port=5001
# cat /proc/net/tcpprobe >/tmp/data.out &
# pid=$!
# iperf -c otherhost
# kill $pid
```

tcpprobe gets loaded and is bound to port 5001/TCP. Reading `/proc/net/tcpprobe` directly gives access to the congestion window and other parameters of the transmission. It produces one line of date in text format for every packet seen. Using port 0 instead of 5001 allows measuring the window of all TCP connections to the machine. The Linux network people have more tips for testing TCP in their wiki. Also, be sure to familiarise yourself with the methods and tools, in case you plan to do some serious testing.

Category: Protocols

Keep in mind that TCP really is a can of worms that get very complex, and that the developers of all Free operating systems deserve a lot of recognition and thanks for dealing with these issues.

I'm on ADSL. Does it really matter?

Most of the new TCP algorithms incorporated into the Linux kernel were designed for very specific purposes. The modules are no magic bullet. If you are connected to a 56k modem, then no congestion algorithm in the Universe will give you more bandwidth than your modem can handle. However, if multiple TCP flows have to share the same link, then some algorithms give you more throughput than others. The best way to find out is to create a test environment with defined conditions, and make comparisons on what you see. My intention was to give you an overview of what's going on in the kernel, and how flexible Linux is when dealing with variable network environments. Happy testing!

Useful links

No packets were harmed while preparing this article. You might wish to take a look at the following tools and articles suitable to save your network link and deepen the understanding of what the Linux kernel does with your packets.

- Congestion Control in Linux TCP (PDF)
- H-TCP
- High Speed TCP
- iperf
- Linux TCP Probe module
- Linux TCP Tuning Guide
- Netpipe

- Network Emulation functionality of the Linux kernel (netem)
-
- On the Effective Evaluation of TCP
- Scalable TCP
- SG Bandwidth*Delay Product Calculator
- TCP at PERTKB
- TCP BIC
- TCP CUBIC (PDF)
- TCP Low Priority
- TCP Veno
- TCP Westwood+
- TCP Westwood
- tcptrace
- Wireshark traffic analyser

You might even wish to study the kernel's source code. All things IPV4 can be found in the */lib/modules/`uname -r`/build/net/ipv4/* directory. The C files contain valuable comments on what Linux does and how it handles certain situations and packets. You don't have to be a programmer to understand it.

Author's footnote

The only reason that I mentioned the religious Linux/*BSD discussion is the fact of being at a party and talking with friends. I don't wish to imply that one OS is better than the other. You can solve and create all your problems with both systems. Keep in mind that TCP really is a can of worms that get very complex, and that the developers of all Free operating systems deserve a lot of recognition and thanks for dealing with these issues.

Talkback: Discuss this article with The Answer Gang

René was born in the year of Atari's founding and the release of the game Pong. Since his early youth he started taking things apart to see how they work. He couldn't even pass construction sites without looking for electrical wires that might seem interesting. The interest in computing began when his grandfather bought him a 4-bit microcontroller with 256 byte RAM and a 4096 byte operating system, forcing him to learn assembler before any other language.

After finishing school he went to university in order to study physics. He then collected

experiences with a C64, a C128, two Amigas, DEC's Ultrix, OpenVMS and finally GNU/Linux on a PC in 1997. He is using Linux since this day and still likes to take things apart und put them together again. Freedom of tinkering brought him close to the Free Software movement, where he puts some effort into the right to understand how things work. He is also involved with civil liberty groups focusing on digital rights.

Since 1999 he is offering his skills as a freelancer. His main activities include system/network administration, scripting and consulting. In 2001 he started to give lectures on computer security at the Technikum Wien. Apart from staring into computer monitors, inspecting hardware and talking to network equipment he is fond of scuba diving, writing, or photographing with his digital camera. He would like to have a go at storytelling and roleplaying again as soon as he finds some more spare time on his backup devices.

Published in Issue 135 of Linux Gazette, February 2007