Linux Traffic Control – Next Generation

Werner Almesberger

October 18, 2002

Abstract

Traffic control in the Linux kernel offers a large set of functions for classifying and scheduling network traffic. Unfortunately, properly configuring traffic control is complicated not only by sometimes obscure underlying theoretical concepts, but it is also made difficult by the rather scary "tc" configuration language used for this.

tcng aims to improve this situation. It defines a new, much more human-friendly configuration language, and provides a compiler translating that to a set of lower-level languages, among them C and "tc". tcng also comes with a simulator that uses the actual kernel code to simulate the traffic control subsystem.

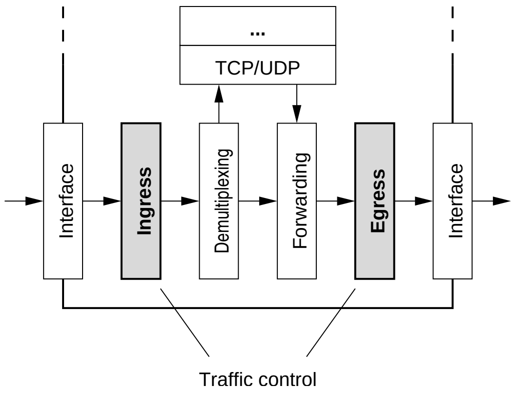

Figure 1: Packet processing in the Linux kernel.

1 Introduction

Traffic control is the set of mechanisms used to condition network traffic, and their application. Typical uses are to give some traffic higher priority than other, to limit the rate at which traffic is sent, or also to block undesirable traffic. The latter example shows that traffic control is in many ways related to firewalling, and that there are cases, where either can partially perform tasks of the other.

When considering the packet processing path shown in figure 1, traffic control is situated next to the network interfaces, at ingress and at egress. At ingress, only a limited set of functions can be performed, such as removal of undesired packets, or preliminary classification. At egress, the full range of traffic control functions is available, including queuing.

In this paper, we briefly introduce the general structure of Linux traffic control in section 2. We then discuss universal usability problems and shortcomings of the current configuration mechanism in section 3, and present a new configuration system (tcng) that is built on top of the existing one in section 4. In section 5, we describe simulators used to test the tcng system, and how they can also be put to interesting uses beyond traffic control. We conclude with a discussion of shortcomings of tcng in section 6.

Configuration of current Linux traffic control is documented by the Linux Advanced Routing and Traffic Control project [1]. A detailed description of the internal structure of traffic control in the Linux kernel can be found in [2], and further material is available on [3].

tcng comes with extensive language and usage documentation. The tcng home page is at `http://tcng.sourceforge.net`

This project started at the beginning of 2001 at EPFL ICA, continued until mid-2002 at Bivio Networks Inc., and has now become one of the author's spare time activities.

2 Linux traffic control overview

The main elements of traffic control are classification, scheduling, and queuing. Classification looks at packet content or at other information related to packets, and attributes them to distinct classes. Packets are then put into queues, and eventually scheduled for transmission. The class of a packet determines in which queue the packet goes, and how it is scheduled. Figure 2 illustrates this process.

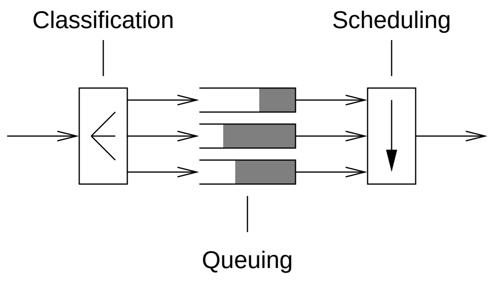

Figure 2: Traffic control functional structure.

Linux traffic control combines queuing and scheduling into so-called "queuing disciplines". Queuing disciplines can be nested, e.g. a queuing discipline implementing a priority scheduler can use queuing disciplines implementing FIFOs[1], for storing the packets.

The design of Linux traffic control is highly modular, with a vast choice of queuing disciplines and classifiers.

Important queuing disciplines include

- the simple drop-tail FIFO[2],
- a "Random Early Detection" (RED) FIFO [4][3]
- the "Token Bucket Filter" (TBF), a shaper that emits packets at a fixed rate,
- a priority scheduler that emits packets in higher priority classes before packets in lower priority classes, and
- the "Hierarchical Token Bucket" that – among other things – allows to assign a certain percentage of the available bandwidth to individual classes.

[1]A FIFO stores and emits packets in the order in which they arrive ("first-in first-out").

[2]A drop-tail FIFO drops newly arriving packets when it has reached its maximum size.

[3]RED starts dropping packets already before reaching the maximum queue size, so that congestion-controlled protocols like TCP can slow down in time.

The most important classifier is called "u32" and it can use any bit patterns in a packet for classification. There are also a few more specialized classifiers, e.g. ones that re-use the results of previous routing or firewalling decisions.

Classification can also be combined with metering. Metering alters the classification decision, based on the rate at which packets arrive. A typical use of metering is to limit the rate of a flow by dropping packets if too many of them arrive in a given time interval.

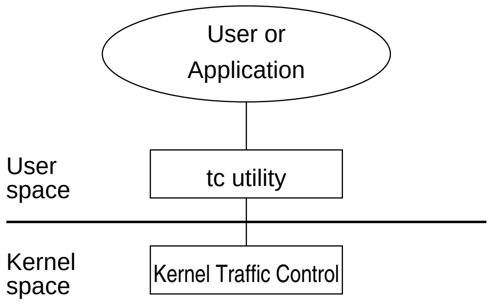

Figure 3: Old traffic control configuration.

As shown in figure 3, the traffic control components in the kernel are configured from user space with a program called "tc", for "traffic control".

3 The problems

It is unfortunate that all this wonderful functionality is fiendishly difficult to use. The reasons for this are manifold: of course, many of the underlying concepts, e.g. the behaviour of TCP when subjected to traffic control, are complicated and need some learning effort to be understood. But there are also several obstacles in the configuration process itself:

First, since all the individual components of a traffic control configuration can be manipulated individually, the location and role of each component needs to be specified in every single configuration command. This redundancy makes configuration scripts very hard to read, and trying to find some small typing error rapidly turns into a search for the proverbial needle in the haystack.

Second, some elements just happen to be complex by design. For example, the u32 classifier uses a pointer plus an offset to walk through a packet, a tree of hash tables to match the data, and a stack to revert to previous states in the classification process if a match has failed. While having full control over all these details allows the construction of highly efficient classifiers, it burdens users with a fair amount of information they need to understand before expecting to accomplish anything, and the sheer number of individual configuration steps is likely to cause mistakes to be made.

A rather vicious twist of the language used by the "tc" tool are unusual naming conventions for units, e.g. "mbps" stands for "2^{20} bytes per second", "kbit" can be "1024 bits" or "1024 bits per second", etc.

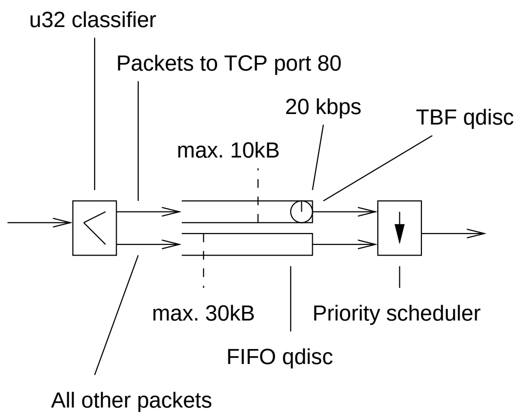

Figure 4: Configuration example.

Figure 4 shows a configuration with a priority scheduler whose high-priority class uses a TBF queue to shape traffic at 20'000 bits per second, and a FIFO for the low-priority class. The "u32" classifier is used to put packets to TCP port 80 into the high-priority class.

These are the corresponding "tc" commands:

```
tc qdisc add dev eth0 root handle 1:0 prio
tc qdisc add dev eth0 parent 1:1 tbf \
  limit 10kB rate 20kbit burst 2kB mtu 1500
tc qdisc add dev eth0 parent 1:2 \
  bfifo limit 30kB
tc filter add dev eth0 parent 1:0 \
  protocol ip u32 \
  match ip protocol 6 ff \
  match tcp dst 50 ffff classid 1:1
```

The language issues used to be complemented by the nearly total absence of documentation, and incomplete and outdated online help. Fortunately, the "Linux Advanced Routing and Traffic Control" project [1] has meanwhile provided much needed relief.

Since traffic control in general, and classification in particular are highly performance-sensitive areas, it is desirable to be able to optimize the code, or even to use hardware acceleration. The modular design of Linux traffic control makes it quite feasible to add new elements which use such optimizations, but such a replacement is likely to be visible to the user, and hence requires configuration changes.

To summarize, the main difficulties of Linux traffic control are that configuration – much like programming in assembler – focuses almost exclusively on low-level details, and that the low level of abstraction of the interface complicates the integration of performance optimizations.

4 The next generation

The goal of the tcng project is to extend the existing traffic control to become more user-friendly, and to make the interfaces of the configuration system more flexible. This is done by adding another layer of abstraction, as shown in figure 5.

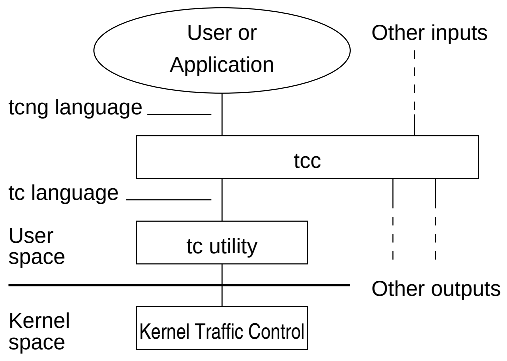

Figure 5: New traffic control configuration.

The traffic control compiler "tcc" takes configuration scripts in the new "tcng" language, translates them to a common internal representation, and then generates commands in the "tc" language from that representation.

Layering tcng on top of tc also allows the use of tcng without requiring any change to the kernel, or to the tc utility.

With this new infrastructure, one can also add modules to configure traffic control subsystems that are now based on "tc". This is described in some more detail in section 4.2.

It would even be possible to add parsers for other languages than "tcng", e.g. XML.

4.1 The "tcng" language

The tcng language is in many ways similar to C and Perl: structure is expressed implicitly with the language syntax, there are variables and arithmetic expressions, classifiers can be expressed in a C-like syntax, and cpp can be used to include files and to write macros.

Below is the tcng version of the example of section 3:

```
dev "eth0" {
    egress {
        class (<$high>) if tcp_dport == 80;
        class (<$low>) if 1;

        prio {
            $high = class {
                tbf(limit 10kB,rate 20kbps,
                  burst 2kB,mtu 1500B);
            }
            $low = class {
                fifo(limit 30kB);
            }
        }
    }
}
```

The configuration begins with the interface name (which could be omitted in the case of eth0) and the role, i.e. ingress or egress. Then, the entire classification is expressed in rules of the form

action `if` *expression*;

where *action* can be the selection of a class, or just `drop` to drop the packet. The boolean expression uses the same syntax and the same precedence rules as C. All the common fields in IP, UDP, TCP, etc. headers are predefined (e.g. `tcp_dport` is the TCP destination port, `ip_src` would be the IPv4 source address, etc.), and users can also easily add their own definitions. IPv4 and IPv6 addresses have been added as first-class data types, and can be written in the usual notation, e.g. `204.152.189.116` or `host"ftp.kernel.org"`. Below is an examples with a more interesting classification expression:

```
ip_off == 1 ||
  (ip_off == 0 &&
    ((ip_proto == IPPROTO_TCP && ip_len < 40) ||
     (ip_proto == IPPROTO_UDP && ip_len < 28) ||
     ip_len < 24))
```

This is a test for so-called "tiny fragments" as described in the [5] and [6]. For convenience, this test is also available as a macro called `ip_is_tiny_fragment`.

Classes are selected with `class (<`*$variable*`>)`, where the variable is later defined in the queuing and scheduling section. That section describes the hierarchy of queuing disciplines, and their configuration. Queuing disciplines with classes are followed by a block (denoted by curly braces, like in C or Perl), which contains all these classes. If a class has a queuing discipline attached to it, that definition is in a block following the class description.

Arithmetic expressions can be used for parameters. tcng adjusts units automatically, e.g. instead of ... `10 kB` ..., one could write ... `10Mbps*8ms` ...[4]

Also metering is smoothly integrated into the tcng language: if one wanted to limit the high-priority traffic in the above example by dropping all packets exceeding 20kbps, instead of using a shaper, the configuration would look as follows:

```
dev "eth0" {
    egress {
        $meter = SLB(rate 20kbps,burst 10kB);

        class (<$high>)
            if tcp_dport == 80 &&
              SLB_or_drop($meter);
        class (<$low>) if 1;

        prio {
            $high = class {
                fifo(limit 10kB);
            }
            $low = class {
                fifo(limit 30kB);
            }
        }
```

[4]This is actually just an approximation: in tcng, bit or byte sizes use a multiplier of 1024, while all other units (rates, time, etc.) use a multiplier of 1000.

```
    }
}
```

"SLB" stands for "single leaky bucket" meter. The SLB_or_drop macro yields "true" (1) if the flow is still within the specified rate, or drops the packet if the flow exceeds the rate. Again, all the primitives that tcng uses to implement meters are also available to users.

4.2 Acceleration

tcng allows to bypass the traffic control subsystem in the Linux kernel and to use alternative software implementations, or hardware accelerators. tcng calls the different ways for implementing traffic control "targets".

An alternative software implementation is illustrated with the "C" target, which emits C code for the classifier, and then invokes gcc to build kernel and tc modules.

The so-called external interface uses a more subtle approach and translates classifiers to the "access control list" form typically found in firewall configurations. A third party program can then use this output to configure a hardware accelerator, e.g. a network processor.

The classifier in our example yields the following output at the external interface:

```
block eth0 egress
offset 100 = 0+(0:4:4 << 5)
action 2 = class 1:2
action 1 = class 1:1
match 0:72:8=0x06 100:16:16=0x0050 action 1
match action 2
```

The "offset" line defines how to calculate the position of the TCP header. The matches are simply tests of bit ranges, e.g. 100:16:16=0x0050 tests if the 16 bits at an offset of 16 bits from the beginning of the TCP header have the hexadecimal value 80.

The key aspect of tcng's ability to generate output for distinct targets is of course that no or only very small changes are required in the configuration scripts used as input.

5 Simulators

Since it would be nearly impossibly for users to correct problems in a tcng configuration if they are only detected when issuing the corresponding tc commands, tcc must carefully mirror all checks done by tc (i.e. syntax and consistency inside a single command) and by the kernel (i.e. parameter validity and consistency across elements).

Testing this can be tricky, because the only way to be absolutely sure that a configuration is valid is to implement it. Doing this on a live system would cause all kinds of problems, ranging from disrupting normal network use to kernel crashes by the occasional bug detected when exercising unusual traffic control setups.

Worse yet, also verifying that the configuration is not only formally correct, but that it actually does what was intended, is time-consuming and typically requires a specific hardware setup.

tcng avoids all these problems by simulating the behaviour of traffic control in user space.

5.1 Traffic control simulator

The current simulator of tcng is called "tcsim". In order to ensure that tcsim simulates exactly the behaviour of the tc utility and the traffic control subsystem in the kernel, it simply takes the code from both, and adds a simulation engine, plus some framework for configuration and for tracing. This is shown in figure 6.

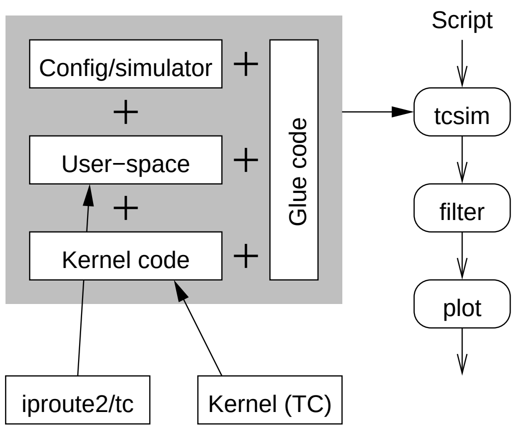

Figure 6: Structure of the "tcsim" simulator.

The output of tcsim is simply a chronological trace of all the events that have happened in the simulation. The output can then be filtered, and counted or plotted.

In order to test our example configuration, we can use the following simulation:

```
dev "eth0" 56kbps {
    #include "example.tc"
}

// (40+85)*8 = 1000 bits
every 20ms
  send TCP_PCK($tcp_dport=25) 0 x 85
time 10s
every 100ms until 20s
  send TCP_PCK($tcp_dport=80) 0 x 85
time 20s
every 25ms
  send TCP_PCK($tcp_dport=80) 0 x 85
time 30s
```

In this example, we first generate low-priority traffic at 50 kbps by sending a packet of 1000 bits every 20 milliseconds.[5] After ten seconds, we add high-priority traffic at 10 kbps. Since the total link speed is only 56 kbps, some of the low-priority traffic is lost now. Ten seconds later, we try to increase the rate of the high-priority traffic to 40 kbps. Since the shaper limits high-priority traffic to only 20 kbps, now half of this traffic is lost.

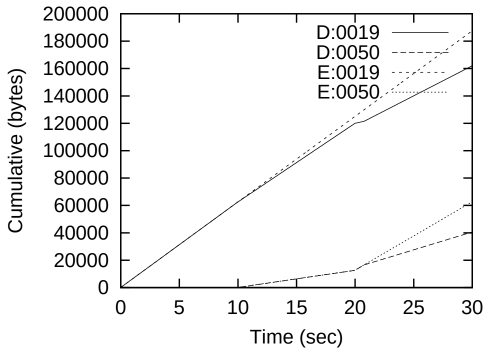

Figure 7: Simulation result.

Figure 7 shows the cumulative traffic that is enqueued and dequeued for destination ports 25 and 80, e.g. "D:0050" is the sum of the dequeued traffic to port 80. This plot was generated directly with the tools available for tcsim.

tcng version 8z has more than 1500 regression tests, and about a third of them use tcsim.

5.2 UML-based simulator

tcsim is used extensively for regression tests in the tcng system, and it is also quite useful to debug configurations. One major drawback of the approach of extracting code fragments and gluing them together in the simulator is that the build procedure for tcsim is fairly fragile and requires frequent adjustments when the kernel changes. Also, 2.4 and recent 2.5 kernels differ sufficiently that the whole extraction process would have to be re-designed and re-written for 2.5.

Fortunately, a much less invasive alternative is already available: "User-Mode Linux" (UML [7]) runs the Linux kernel in user space, and it has recently been added to 2.5.

As an added bonus, UML is a complete Linux system, so the entire networking stack, including TCP, routing, etc. is available, and arbitrary applications can be used to generate traffic. This offers fantastic possibilities far beyond just experimenting with traffic control. For example, one could use this as a starting point for implementing a comprehensive network simulator similar to ns-2 [8].

In order to use UML for simulations, mainly two changes are necessary: the concept of "time" in UML must be replaced with the virtual time of an event-driven simulator. In order to do this, the simulator must know when a timer-driven activity is due, and advance the kernel timers accordingly. An efficient implementation of this would also avoid executing unnecessary ticks of the 100 or 1000 Hz kernel timer.

The second change is to introduce mechanisms for executing simulation actions (e.g. enqueuing of a packet), and for capturing the results of such actions (e.g. receiving a notification when a packet gets sent).

By now, the conversion of UML to use simulation time has been completed in the "umlsim" project [9], while instrumentation of the UML simulator still remains to be done.

A UML kernel running with virtual time will also be interesting for general regression testing, e.g. some of the tests regularly performed in the FreeS/WAN project [10] consist of waiting for timeouts, so the test machines just sit there for minutes, waiting for time to pass. With a virtual time base, such tests could be completed in seconds instead.

[5]The packet consists of 20 bytes for the IP header, 20 bytes for the TCP header, and the 85 zero bytes generated with `0 x 85`.

6 Remaining issues

While tcng contributes much towards making traffic control more accessible, there are still some problems left, and it also introduces some new ones of its own.

First of all, tcng does not help at all against the inherent complexity of traffic management. In order to allow users to set up traffic control without needing to spend weeks reading books and research papers, cookbook solutions are needed. A very nice example of such a cookbook solution is the so-called "Wonder Shaper" [11] that implements intelligent bandwidth sharing for the typical domestic Internet access, and that requires users only to configure those parameters they can be expected to understand.

6.1 Performance

The perhaps biggest restriction of tcng is currently the rather poor performance of the algorithms that translate a classifier expression into something suitable for the u32 classifier, or also the external interface. First of all, it can take tcc a long time (e.g. minutes, or worse) to re-arrange sufficiently hairy expressions. Second, the output of tcc is not always as compact as it could be. Third, tcc does not make use of all the mechanisms offered by u32, so the resulting u32 classifiers are comparably slow.

Among these performance issues, the potentially large amount of internal processing in tcc is the most critical one. The author is currently experimenting with algorithms that translate expressions into finite state machines (which are structurally much simpler), and then work on these.

Instead of trying to make optimal use of u32, it is probably better to write a new classifier that is optimized for what output tcc can generate easily.

6.2 Dynamic reconfiguration

One area where tcng is still lagging behind tc is dynamic reconfiguration. tcng configurations are currently self-contained and describe the whole setup of an network interface or a set of interfaces. If, for example, a single rule needs to be added to a classifier containing a few thousand rules, the whole configuration will have to be re-processed. While this may be acceptable in many cases, applications requiring frequent small changes are severely penalized by this design.

A possible solution is to simply make processing of the entire configuration very fast.

Another approach would be to add constructs to the tcng language to express highly regular patters, e.g. a set construct. If all else fails, one can still create optimized classifiers, or introduce "magic" elements in regular classifiers, which can then be directly manipulated at run time, without tcc's knowledge.

6.3 Extensibility

Last but not least, it is currently hard to add new traffic control elements to tcng. Partially, this is plainly the result of the need for very detailed consistency checks, which are hardcoded into tcc, but simpler items like parameter naming and range checks could be improved and generalized.

7 Conclusion

Traffic control in Linux is wonderfully versatile but suffers today from a number of problems that make it unnecessarily hard to use. We have described these problems, and we have presented a new configuration system, called tcng, that builds on top of the existing architecture, that provides a configuration language that is intuitive to most programmers, and that – by abstracting its interfaces from the actual implementation – makes it easier to interface with the traffic control subsystem. Furthermore, we have briefly introduced the simulation environment of tcng, and its uses for other applications.

Last but not least, we have listen several problems that need to be addressed before tcng can be considered to be in every respect a more then adequate replacement for tc, and have sketched directions for further work.

References

[1] Hubert, Bert; et al. *Linux Advanced Routing & Traffic Control*, `http://lartc.org/`

[2] Almesberger, Werner. *Linux Traffic Control — Implementation Overview*, ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.gz, Technical Report SSC/1998/037, EPFL, November 1998.

[3] *Differentiated Services on Linux*, http://diffserv.sourceforge.net

[4] Floyd, Sally; Jacobson, Van. *Random Early Detection Gateways for Congestion Avoidance*, IEEE/ACM Transactions on Networking, August 1993.

[5] Ziemba, G. Paul; Reed, Darren; Traina, Paul. *Security Considerations for IP Fragment Filtering*, IETF, October 1995.

[6] Miller, Ian. *Protection Against a Variant of the Tiny Fragment Attack*, IETF, June 2001.

[7] Dike, Jeff; et al. *The User-mode Linux Kernel Home Page*, http://user-mode-linux.sourceforge.net/

[8] USC/UCB/LBNL/VINT. *The Network Simulator – ns-2*, http://www.isi.edu/nsnam/ns/

[9] Almesberger, Werner. *umlsim – Event-driven simulation for User-Mode Linux*, http://www.almesberger.net/umlsim/

[10] Richardson, Michael C. *Automatic Regression testing of network code: User-Mode Linux and FreeSWAN*, Ottawa Linux Symposium, June 2002, http://www.sandelman.ottawa.on.ca/SSW/freeswan/fsumltesting/

[11] Hubert, Bert. *The Wonder Shaper*, http://lartc.org/wondershaper/