Trickle: A Userland Bandwidth Shaper for Unix-like Systems

Marius A. Eriksen*
Google, Inc.
mae@google.com

Abstract

As with any finite resource, it is often necessary to apply policies to the shared usage of network resources. Existing solutions typically implement this by employing traffic management in edge routers. However, users of smaller networks regularly find themselves in need of nothing more than ad-hoc rate limiting. Such networks are typically unmanaged, with no network administrator(s) to manage complicated traffic management schemes. Trickle bridges this gap by providing a simple and portable solution to rate limit the TCP connections of a given process or group of processes.

Trickle takes advantage of the Unix dynamic loader's preloading functionality to interposition itself in front of the BSD socket API provided by the system's `libc`*. Running entirely in user space, shapes network traffic by delaying and truncating socket I/Os without requiring administrator privileges. Instances of Trickle can cooperate, even across networks allowing for the specification of global rate limiting policies. Due to the prevalence of BSD sockets and dynamic loaders, Trickle enjoys the benefit of portability accross a multitude of Unix-like platforms.*

1 Introduction

Bandwidth shaping is traditionally employed monolithically as part of network infrastructure or in the local operating system kernel which works well for providing traffic management to large networks. Such solutions typically require dedicated administration and privileged access levels to network routers or the local operating system.

Unmanaged network environments without any set bandwidth usage policies (for example home and small office networks) typically do not necessitate mandatory traffic management. More likely, the need for bandwidth shaping is largely ad-hoc, to be employed when and where it is needed. For example,

- bulk transfers may adversely impact an interactive session and the two should receive differentiated services, or
- bulk transfers may need to be prioritized.

Furthermore, such users may not have administrative access to their operating system(s) or network infrastructure in order to apply traditional bandwidth shaping techniques.

Some operating systems provide the ability to shape traffic of local origin (these are usually extensions to the router functionality provided by the OS). This functionality is usually embedded directly in the network stack and resides in the operating system kernel. Network traffic is not associated with the local processes responsible for generating the traffic. Rather, other criteria such as IP, TCP or UDP protocols and destination IP addresses are used in classifying network traffic for shaping. These policies are typically global to the host (thus applying to all users on it). Since these policies are mandatory and global, it is the task of the system administrator to manage the traffic policies.

These are the many burdens that become evident if one would like to employ bandwidth shaping in an ad-hoc manner. While there have been a few attempts to add voluntary bandwidth shaping capabilities to the aforementioned in-kernel shapers[25], there is still a lack of a viable implementation and there is no use of collaboration between multiple hosts. These solutions are also non-portable and there is a lack of any standard application or user interfaces.

We would like to be able to empower any unprivileged user to employ rate limiting on a case-by-case basis, without the need for special kernel support. Trickle addresses precisely this scenario: Voluntary ad-hoc rate

*Work done by the author while at the University of Michigan.

limiting without the use of a network wide policy. Trickle is a portable solution to rate limiting and it runs entirely in user space. Instances of Trickle may collaborate with each other to enforce a network wide rate limiting policy, or they may run independently. Trickle only works properly with applications utilizing the BSD socket layer with TCP connections. We do not feel this is a serious restriction: Recent measurements attribute TCP to be responsible for over 90% of the volume of traffic in one major provider's backbone[16]. The majority of non-TCP traffic is DNS (UDP) – which is rarely desirable to shape anyway.

We strive to maintain a few sensible design criteria for Trickle:

- *Semantic transparency*: Trickle should never change the behavior or correctness of the process it is shaping (Other than the data transfer rates).
- *Portability*: Trickle should be extraordinarily portable, working with any Unix-like operating system that has shared library and preloading support.
- *Simplicity*: No need for excessively expressive policies that confuse users. Don't add features that will be used by only 1 in 20 users. No setup cost, a user should be able to immediately make use of Trickle after examining just the command line options (and there should be very few command line options).

The remainder of this paper is organized as follows: Section 2 describes the linking and preloading features of modern Unix-like systems. Section 3 provides a high-level overview of Trickle. Section 4 discusses the details of Trickle's scheduler. In section 5 we discuss related work. Finally, section 9 concludes.

2 Linking and (Pre)Loading

Dynamic linking and loading have been widely used in Unix-like environments for more than a decade. Dynamic linking and loading allow an application to *refer* to an external symbol which does not need to be resolved to an address in memory until the runtime of the particular binary. The canonical use of this capability has been to implement *shared libraries*. Shared libraries allow an operating system to share one copy of commonly used code among any number of processes. We refer to resolving these references to external objects as *link editing*, and to unresolved external symbols simply as *external symbols*.

After compilation, at link time, the linker specifies a list of libraries that are needed to resolve all external symbols. This list is then embedded in the final executable file. At load time (before program execution), the link editor maps the specified libraries into memory and resolves all external symbols. The existence of any unresolved symbols at this stage results in a run time error.

To load load its middleware into memory, Trickle uses a feature of link editors in Unix-like systems called preloading. Preloading allows the user to specify a list of shared objects that are to be loaded together the shared libraries. The link editor will first try to resolve symbols to the preload objects (in order), thus selectively bypassing symbols provided by the shared libraries specified by the program. Trickle uses preloading to provide an alternative version of the BSD socket API, and thus socket calls are now handled by Trickle. This feature has been used chiefly for program and systems diagnostics; for example, to match `malloc` to `free` calls, one would provide an alternative of these functions via a preload library that has the additional matching functionality.

In practice, this feature is used by listing the libraries to preload in an environment variable. Preloading does not work for set-UID or set-GID binaries for security reasons: A user could perform privilege elevation or arbitrary code execution by specifying a preload object that defines some functionality that is known to be used by the target application.

We are interested in interpositioning Trickle in between the shaped process and the socket implementation provided by the system. Another way to look at it, is that Trickle acts as a proxy between the two. We need some way to call the procedures Trickle is proxying. The link editor provides this functionality through an API that allows a program to load an arbitrary shared object to resolve any symbol contained therein. The API is very simple: Given a string representation of the symbol to resolve, a pointer to the location of that symbol is returned. A common use of this feature is to provide plug-in functionality wherein plugins are shared objects and may be loaded and unloaded dynamically.

Figure 1 illustrates Trickle's interpositioning.

3 How Trickle Works

We describe a generic rate limiting scheme defining a black-box scheduler. We then look at the practical aspects of how Trickle interpositions its middleware in order to intercept socket calls. Finally we discuss how multiple instances of Trickle collaborate to limit their aggregate bandwidth usage.

3.1 A Simple Rate Limiting Scheme

A process utilizing BSD sockets may perform its own rate limiting. For upstream limiting, the application can

do this by simply limiting the rate of data that is written to a socket. Similarly, for downstream limiting, an application may limit the rate of data it *reads* from a socket. However, the reason why this works is not immediately obvious. When the application neglects to read some data from a socket, its socket receive buffers fill up. This in turn will cause the receiving TCP to advertise a smaller receiver window (`rwnd`), creating back pressure on the underlying TCP connection thus limiting its data flow. Eventually this "trickle-down" effect achieves end-to-end rate limiting. Depending on buffering in all layers of the network stack, this effect may take some time to propagate. More detail regarding the interaction between this scheme and TCP is provided in section 4.

While this scheme is practical, two issues would hinder widespread employment. Firstly, the scheme outlined is deceptively simple. As we will see in section 4, there are many details which make shaping at this level of abstraction complicated. The second issue is that there are no standard protocols or APIs for multiple processes to collaborate.

We also argue that employing bandwidth shaping inside of an application breaks abstraction layers. It is really the task of the operating system to apply policies to bandwidth usage, and it should not need to be a feature of the application. Even if libraries were developed to assist application developers, employing rate limiting in this manner would still put considerable burden on the developers and it should not be expected that every developer would even support it. The socket API provided by the OS provides certain functionality, and it should be the freedom of the application to use it unchanged, and not have to rely on semantics at the lower levels of abstraction in order to limit bandwidth usage.

There are also exceptions to these arguments. For example, certain protocols may benefit from application level semantics to perform shaping. Another example is that some applications may be able to instruct the sending party to limit its rate of outbound traffic[9] which is clearly preferable over relying on TCP semantics to perform traffic shaping.

Trickle provides a bandwidth shaping service without the need to modify applications. Trickle augments the operating system by *interpositioning* its middleware in front of the `libc` socket interface. From there, Trickle applies rate limiting to any dynamically linked binary that uses the BSD socket layer. By providing a standard command line utility, Trickle provides a simple and consistent user interface to specify rate limiting parameters. Communicating with the *trickle daemon*, allows all instances of Trickle to participate in collaborative rate limiting, even across hosts.

In addition to allowing portability, this approach offers several advantages. There is no need for extending the kernel nor configuring such extensions; any user may use and configure Trickle any way she wants, making it ideal for ad-hoc rate limiting. There are also a number of advantages to this approach from the developer's point of view. Furthermore, being entirely contained in userland has made Trickle inherently easier to develop. It is easier to perform experiments and the software is easier to maintain and will be understood by a wider audience.

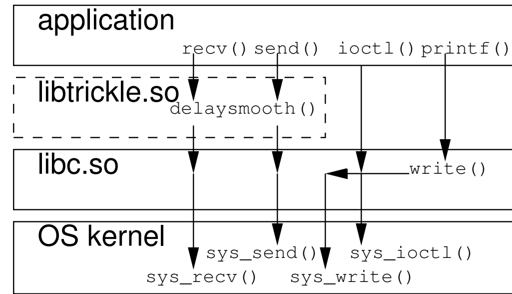

Figure 1: `libtrickle.so` is preloaded in the application's address space, calls to `recv()` and `send()` are handled by `libtrickle.so` and passed down to `libc`.

The primary disadvantage to using this approach is that all usage of Trickle is voluntary – that is, one cannot enforce rate limiting by policy (though some operating systems provide a mechanism for administrators to enforce preload libraries, there are still ways to get around its interpositioning). For its intended usage, this is not a big drawback as ad-hoc bandwidth shaping implies users do so voluntarily. Secondly and with smaller impact, Trickle cannot work with statically linked binaries.

3.2 The Mechanics of Library Interpositioning

With very rare exceptions, network software for Unix-like systems uses the socket abstraction provided by the operating system. In reality, the socket abstraction is entirely contained in the system call layer with corresponding `libc` shims[1]. Thus, with the use of the link editor's preload functionality, we interposition the Trickle middleware at a convenient level of abstraction and we do so entirely in user space.

Using preload objects, we replace the BSD socket abstraction layer provided by `libc`. However, to successfully interposition the Trickle middleware, we must be able to call the original version of the very interface we have replaced. To resolve this issue, we need to take advantage of the second feature of the link editor we discussed: We simply explicitly resolve the `libc` shims and call them as needed. This is done by opening the

actual object file that contains `libc`, and using the link-editor API to resolve the symbols needed. The location of the `libc` shared object is discovered in the configuration/compilation cycle, but could just as easily be discovered dynamically at run time. Figure 1 attempts to illustrate the mechanics of the interpositioning of the Trickle middleware.

In practice, preload objects are specified by the environment variable `LD_PRELOAD`. Trickle's command line utility, `trickle`, sets this environment variable to the object that contains Trickle's middleware. Additionally, it passes any parameters specified by the user in other environment variables in a well defined namespace. These parameters may include upstream or downstream rates to apply, as well as whether or not this instance of Trickle should collaborate with other instances of Trickle.

3.3 The Life of a Socket

New sockets are created with either the `socket()` or `accept()` interfaces. An old socket is aliased with calls to `dup()` or `dup2()`. Any new or duplicated socket is marked by Trickle by keeping an internal table indexing every such socket. File descriptors that are not marked are ignored by Trickle, and relevant calls specifying these as the file descriptor argument are simply passed through to the `libc` shims without any processing. Note that it is also possible for an application to perform file descriptor passing: An application may send an arbitrary file descriptor to another over local inter process communication (IPC), and the receiving application may use that file descriptor as any other. File descriptor passing is currently not detected by Trickle. When a socket is closed, it is unmarked by Trickle. We say that any marked socket is *tracked* by Trickle.

Two categories of socket operations are most pertinent to Trickle: Socket I/O and socket I/O multiplexing. In the following discussion we assume that we possess a black box. This black box has as its input a unique socket identifier (e.g. file descriptor number) and the direction and length of the I/O operation to be performed on the socket. A priority for every socket may also be specified as a means to indicate the wish for differentiated service levels between them. The black box outputs a recommendation to either *delay* the I/O, to *truncate* the length of the I/O, or a combination of the two. We refer to this black box as the Trickle *scheduler* and it discussed in detail in a later section.

The operation of Trickle, then, is quite simple: Given a socket I/O operation, Trickle simply consults the scheduler and delays the operation by the time specified, and when that delay has elapsed, it reads or writes at most the number of bytes specified by the scheduler. If the socket is marked non-blocking, the scheduler will specify the length of I/O that is immediately allowable. Trickle will perform this (possibly truncated) I/O and return immediately, as to not block and violate the semantics of non-blocking sockets. Note that BSD socket semantics allow socket I/O operations to return short counts – that is, an operation is not required to complete in its entirety and it is up to the caller to ensure all data is sent (for example by looping or multiplexing over a calls to `send()` and `recv()`). In practice, this means that the Trickle middleware is also allowed to return short I/O counts for socket I/O operations without affecting the semantics of the socket abstraction. This is an essential property of the BSD socket abstraction that we use in Trickle.

Multiplexing I/O operations, namely calls to `select()` and `poll()`[2] are more complex. The purpose of the I/O multiplexing interface is to, given a set of file descriptors and conditions to watch for each, notify the caller when any condition is satisfied (e.g. file descriptor x is ready for reading). One or more of these file descriptors may be tracked by Trickle, so it is pertinent for Trickle to wrap these interfaces as well. Specifically, `select()` and `poll()` are wrapped, these may additionally wait for a *timeout* event (which is satisfied as soon as the specified timeout value has elapsed).

To simplify the discussion around how Trickle handles multiplexing I/O, we abstract away the particular interface used and assume that we deal only with a set of file descriptors, one or more of which may be tracked by trickle. Also specified is a global timeout. For every file descriptor that is in the set and tracked by Trickle, the scheduler is invoked to see if the file descriptor would be capable of I/O immediately. If it is not, it is removed from the set and added to a holding set. The scheduler also returns the amount of time needed for the file descriptor to become capable of I/O, the holding time. The scheduler calculates this on the basis of previously observed I/O rates on that socket. Trickle now recalculates the timeout to use for the multiplexing call: This is the minimum of the set of holding times and the global timeout.

Trickle then proceeds to invoke the multiplexing call with the new set of file descriptors (that is, the original set minus the holding set) and the new timeout. If the call returns because a given condition has been satisfied, Trickle returns control to the caller. If it returns due to a timeout imposed by Trickle, the process is repeated, with the global timeout reduced by the time elapsed since the original invocation of the multiplexing call (wrapper). In practice, a shortcut is taken here, where only file descriptors from the holding set are examined, and rolled in if ready. The process is repeated until any user specified condition is satisfied by the underlying multiplexing call.

3.4 Collaboration

We have detailed how Trickle works with a set of sockets in a single process, though more often than not it is highly practical to apply *global rate limits* to a set of processes that perform network I/O. These processes do not necessarily reside a single host; it is often useful to apply them on every process that contributes to the network traffic passing through a particular gateway router, or to use a global limit to control the utilization of the local area network. It may also be desirable to apply individual rate limiting policies for processes or classes of processes.

Trickle solves this by running a daemon, `trickled` which coordinates among multiple instances of Trickle. The user specifies to `trickled` the *global* rate limitations which apply across all instances of Trickle. That is, the aggregate I/O rates over all processes shaped by Trickle may not exceed the global rates. Furthermore, the user can specify a priority per instance or type of instance (e.g. interactive applications), allowing her to provide differentiated network services to the various client applications.

By default, `trickled` listens on a BSD domain socket and accepts connections from instances of Trickle running on the local host. These instances then request a bandwidth allocation from `trickled`. The bandwidth allocation is computed using the same black box scheduler described previously. It is used in a slightly different mode where the scheduler simply outputs the current rate the entity is assigned. These rate allocations may change frequently, and so the instances of Trickle get updated allocations with some preset regularity.

It is worth noting that there is a simple denial of service attack should a rogue user exist on the system. This user could simply create as many fake Trickle instances as necessary and, without actually doing any socket I/O, report data transfers to `trickled`. Of course, such a user could, though using more resources to do so, also consume as much network resources as possible, in effect achieving the same result by exploiting TCP fairness.

The same model of collaboration is applied across several hosts. Instead of listening on a Unix domain socket, `trickled` listens on a TCP socket, and can thus schedule network resource usage across any number of hosts. In this scenario, rate allocation updates may start to consume a lot of local network resources, so care must be taken when setting the frequency at which updates are sent.

4 I/O Scheduling With Rate Restrictions

The problem of rate limiting in Trickle can be generalized to the following abstraction: Given a number of entities capable of transmitting or receiving data, a global rate limit must be enforced. Furthermore, entities may have different *priorities* relative to each other as to differentiate their relative service levels. In Trickle, we use this abstraction twice: In a shaped process, a socket is represented as an entity with priority 1. In `trickled` every collaborating process is represented by an entity (the collaborating processes may even reside on different hosts) and every entity is assigned a priority according to a user specified policy.

When an entity is ready to perform some I/O, it must consult the scheduler. The scheduler may then advise the entity to *delay* its request, to *partially complete* the request (i.e. truncate the I/O operation), or a combination of the two. In this capacity, the scheduler is *global* and coordinates the I/O allocation over all entities. In another mode, the scheduler simply outputs the current global rate allocation for the requesting entity.

After an entity has performed an I/O operation, it notifies the Trickle scheduler with the direction (sent or received) and length of the I/O. Trickle then updates a bandwidth statistics structure associated with that entity and direction of data. This structure stores the average data throughput rate for the entire lifetime of that entity as well as a windowed average over a fixed number of bytes. Also, an aggregate statistic covering all entities is updated.

Before an entity performs I/O, it consults the Trickle scheduler to see how much delay it must apply and how much data it is allowed to send or receive after the delay has elapsed. Let us assume for the moment that the scheduler need only decide for how long to delay the requested I/O operation.

4.1 Distribution and allocation

Every entity has an assigned number of *points* inversely proportional to that entity's priority. The global rate *limit* is divided by the total number of points over all entities, and this is the rate allotment per point. If every entity performed I/O with a rate equal to its number of points multiplied by the per point allotment, the total rate over all entities would be at the rate limit and every entity would perform I/O at a rate proportional to their assigned priority. Since Trickle is performing bandwidth shaping, most often the entities has the ability to exceed the transfer rates that they are assigned by the scheduler. The entities only very seldomly behave in any predictable manner: Their data transfer rates may be bursty, they may have consistent transfer rates lower than their alloted rates, or they might be idle. At the same time, the scheduler needs to make sure that the entities in aggregate may transfer data at a rate capped only by the total rate limit: Trickle should never hinder its client applications from fully uti-

lizing the bandwidth allocated to them.

Statistics structures as well as limits are kept independently *per-direction* and thus Trickle is fully asymmetric. Furthermore, it is worthy to note that Trickle makes scheduling decisions based on a *windowed average*. Dealing with instantaneous bursts is discussed later.

4.2 Scheduling

Trickle uses a simple but effective algorithm to schedule I/O requests. It is a global scheduler that preserves every requirement outlines above. We maintain a total T, which is initialized with the total number of points alloted over all entities. We also maintain a per-point allotment P which is initially calculated as outlined above. An entity consumes bandwidth less than its allotment if its measured (windowed average) consumption is less than its number of points E_p multiplied by the per-point allotment P.

For every entity that consumes bandwidth less than its allotment, we subtract E_p from T, and then add the difference between E's actual consumption and it's aloted consumption to a free pool, F. After iterating over the entities, the value of the free pool is redistributed amongst the remaining entities. In practice, this is done by inflating the per-point allotment: $P = P + F/T$.

This process is then repeated until there are either *no remaining entities* that have more allotment than their consumption or *all remaining entities* have more allotment than their consumption. This is the stable state. We call the final allotment of each entity after this process the *adjusted allotment*.

After the adjustment process, if the entity being scheduled is currently consuming bandwidth at a rate less than its adjusted allotment, Trickle allows the operation to proceed immediately. If not, it requests the entity to delay the operation by the time it would take to send the requested number of bytes at the adjusted rate.

Figure 2 shows bandwidth consumption at the receiving end of two bulk transfers shaped collaboratively by Trickle, one having a lower priority. The aggregate bandwidth consumption (i.e. the sum of the two) is also shown. Trickle was configured with a global receive limit of 10 kB/s. The lower priority transfer has an average transfer rate of 3,984 bytes/second with a standard deviation of 180 bytes/second. The higher priority transfer averages at 5,952 bytes/sec with a standard deviation of 455 bytes/second.

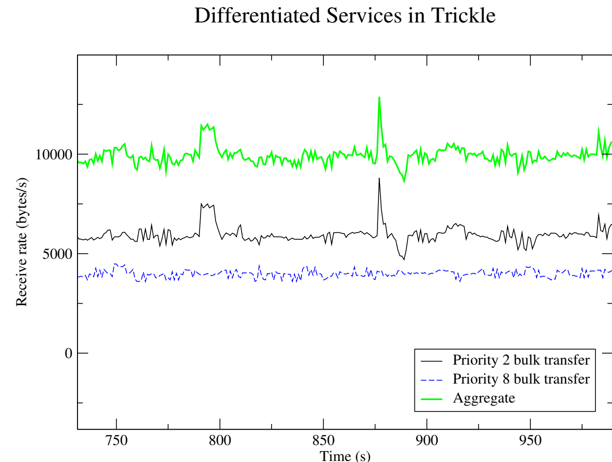

Figure 2: Measuring a windowed-average bandwidth consumption of two bulk transfers with differentiated service. Trickle was configured with a global limit of 10 kB/s.

4.3 Smoothing

This naïve approach of delaying I/O operations tends to result in very bursty network behavior since we are blocking an I/O of any length for some time, and then letting it complete in full. As expected, this behavior is especially prevalent when operations are large. In the short term, burstiness may even result in *over shaping* as network conditions are changing, and the scheduler might have been able allocate more I/O to the stream in question. Figure 4 shows the extremity of this effect, where operations are large and rate limiting very aggressive.

The length of an I/O may also be unpredictable, especially in applications with network traffic driven by user input (e.g. interactive login sessions or games). Such applications are naturally bursty and it would be advantageous for Trickle to dampen these bursts.

Note that even if Trickle considered instantaneous bandwidth consumption in addition to the windowed average as netbrake[5] does, bursty behavior would still be present in many applications. When shaping is based on both instantaneous and average bandwidths, it is the hope that the buffers underneath the application layer will provide dampening. For I/Os (keep in mind that applications are allowed to make arbitrarily large I/O requests to the socket layer) with lengths approaching and exceeding the bandwidth $\times$ delay product, buffering provides little dampening.

Thus, we introduce techniques to *smooth* these bursts. The techniques we introduce are generic and apply equally to both instantaneous and TCP burstiness. Our technique makes use of two parameters to normalize traffic transmitted or received by the socket layer.

In the following discussion, we use the variable `pointvalue` to indicate the value of a point after scheduling, `numpoints` is the number of points allocated to

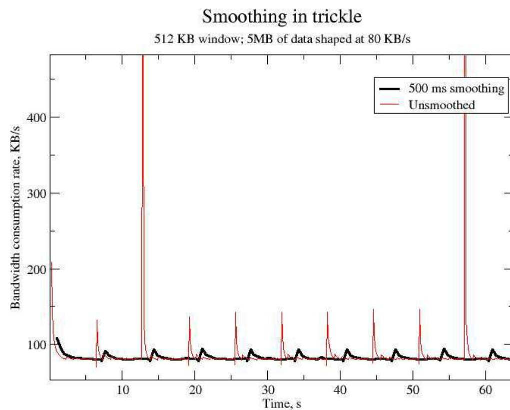

Figure 3: Measuring a windowed-average bandwidth consumption at the receiving end, this plot shows the effects of smoothing in Trickle.

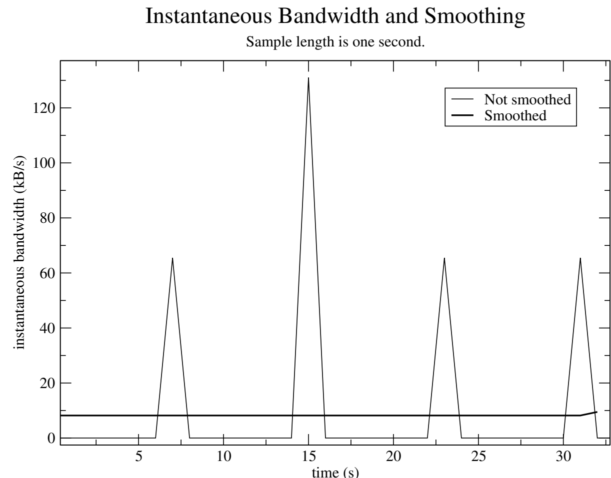

Figure 4: When dealing with large I/O operations, smoothing helps amortize bandwidth consumption.

the entity in question and length refers to the (original) length of the I/O being scheduled.

We first introduce a *time smoothing* parameter. We set the delay imposed on a socket to the minimum of the time smoothing parameter and the delay requested (by the process outlined in the previous subsection). If the time smoothing delay is the smaller of the two, the length is truncated so that the entity meets its adjusted rate allotment. This is called the *adjusted length*: $\mathsf{adjlen} = \mathsf{pointvalue} * \mathsf{numpoints} * \mathsf{timesmoothingparam}$. The purpose of the time smoothing parameter is to introduce a certain continuity in the data transfer.

We must be able to handle the case where the adjusted length is 0. That is, the time smoothing parameter is too small to send even one byte. To mitigate this situation, we introduce a *length smoothing* parameter. When the adjusted length is 0, we simply set the length to the length smoothing parameter, and adjust the delay accordingly: $\mathsf{delay} = \mathsf{length}/(\mathsf{pointvalue} * \mathsf{numpoints})$.

The effect of smoothing is illustrated in figure 3. Here, Iperf[27], a network measurement tool, was run for 60 seconds. The source node was rate limited with Trickle, which was configured to rate limit at 80 KB/s. The thin line indicates the resulting Iperf behavior without any smoothing applied. The thick line applies a time smoothing parameter of 500 ms. The transfer rates shown are instantaneous with a sampling rate once per second.

In practice, the scheduler deployed as follows: In a single Trickle instance, the entities are sockets, all with priority 1 and the global limit is either user specified or by `trickled`. `trickled` again uses the same scheduler: Here the entities are the instances of Trickle, and the global limit is specified by the user. Note that in this case, the scheduler does not need to apply delay or smoothing, it simply needs to report back to each instance of Trickle what its allotment is at that point in time.

4.4 Streams vs. Packets

One big difference between Trickle and in-kernel rate limiters is that Trickle only has access to much higher levels of the OSI layers. The only context available to Trickle are BSD sockets, while in-kernel rate limiters typically schedule discrete packets and reside in or around the network layer.

Trickle can only provide bandwidth shaping by delaying and truncating I/O on a socket (e.g. TCP/IP stream), and must rely on the underlying semantics in order to be effective. Furthermore, Trickle is affected by local buffering in every OSI[24] layer, and this effectively reduces the best reaction time Trickle can provide. Together, these conditions severely reduces the granularity at which Trickle can operate.

Since in-kernel rate limiters reside so low in the network stack, they may exercise complete control of the actual outgoing date rates. These rate limiters typically provide ingress rate limiting by a technique called “policing”. Policing amounts to simply dropping matching incoming packets even though there is no real local contention that would otherwise prevent them from being delivered. Note that policing is a different strategy for shaping incoming traffic. When a policing router drops a packet, it creates congestion in the view of the sending TCP as it will need to retransmit the lost packet(s). When detecting congestion, the sending TCP will reduce its congestion window, effectively throttling its rate of

transmission. After shrinking its congestion window, the sending TCP has to expand its congestion window again by the slow start and congestion avoidance strategies. Trickle's approach avoids artificial congestion by shrinking the advertised TCP receiver window (`rwnd`), causing the sending TCP to artificially limit the amount of data it can send. One advantage of this technique is that policing makes TCP more volatile in a somewhat subtle way. In its steady-state, TCP is self-clocking, that is, it tries to inject a packet into the network for every packet that leaves the network. Thus, as networks get more congested, router queues fill up and round trip times (RTTs) increase, and thus the sending TCP slows its transmission rate. When policing, packets are dropped indiscriminately, and TCP has no chance to avoid the congestion by observing increasing RTTs. This results in a more volatile TCP state as the sending TCP will have to enter fast retransmit/recovery mode more frequently.

4.5 The Interactions of Delay, Smoothing and TCP

To recapitulate, Trickle shapes network traffic by delaying and truncating I/O requests according to a few simple parameters. Trickle attempts to reduce burstiness by smoothing, which in effect introduces some time and length normalization for the emitted I/O operations. We now explore how our shaping techniques interact with TCP.

For ingress traffic, this should result in a less volatile`rwnd` in the receiving TCP since the utilization of socket receive buffers have smaller variance.

Smoothing is also beneficial for the transmitting TCP. Because the data flow from the application layer is less bursty, the TCP does not have to deal with long idle times which may reduce responsiveness: It is standard practice to reduce the congestion window (`cwnd`) and perform slow start upon TCP idleness beyond one retransmission timeout (`RTO`)[20].

Smoothing may also be used for adapting to interactive network protocols. For example, a smaller time smoothing parameter should cause data to be sent in a more continuous and consistent manner, whereas the lack of smoothing would likely cause awkward pauses in user interactions.

Another tradeoff made when smoothing is that you are likely to loose some accuracy because of timing. When using timers in userland, the value used is the floor of the actual timeout you will get, and thus when sleeping on a timer just once for some I/O, the inaccuracy is amortized over the entirety of that I/O. However, smoothing is likely to break this I/O up into many smaller I/Os, and the timer inaccuracies may be more pronounced. The ultimate compromise here would be to use the effects of buffering whenever you can, and to use smoothing whenever you have to. This is left for future work.

5 Related Work

There are a number of generic rate limiting software solutions and one is included in nearly every major open source operating system. These operate in a mostly traditional manner (defining discrete packet queues and applying policies on these queues). What differentiates these utilities are what operating system(s) they run on and how expressive their policies are. Examples are AltQ[14], Netnice[21], Dummynet[23] and Netfilter[26].

Several network client and server applications incorporate rate limiting as a feature. For example, OpenSSH[8]'s has the ability to rate limit `scp` file transfers. `rsync`[9] too, features rate limiting. Both use a simple scheme that sleeps whenever the average or instantaneous bandwidth rates go beyond their threshold(s). `rsync` has the additional advantage that they may control the sender as well (their protocol is proprietary) and so bandwidth is shaped at the sender, which is easier and more sensible.

There are a few modules for Apache[1] which incorporate more advanced bandwidth shaping. These modules are typically application layer shapers: they exploit additional context within Apache and the HTTP protocol. For example, such modules could have the ability to perform rate limiting by particular cookies, or on CPU intensive CGI scripts.

Many peer-to-peer applications also offer traffic shaping. Again, here there is great opportunity to use application level knowledge to apply policies for shaping[2, 4].

Netbrake[5] is another bandwidth shaper that uses shared library preloading.[3] Like Trickle, it delays I/Os on sockets. Netbrake does not have the ability to coordinate bandwidth allocation amongst several instances. Netbrake calculates aggregate bandwidth usage for all sockets in a process, and shapes only according to this: That is, if a given socket I/O causes the global rate to exceed the specified limit, that socket is penalized with a delay as for bandwidth consumption to converge to the limit, and there no equivalent to the smoothing in Trickle. Thus, Netbrake does not retain a sense of "fairness" among sockets: One "overzealous" socket could cause delays in other sockets performing I/O at lower rates. This does not retain the TCP fairness semantics (nor does it attempt to), and could cause uneven application performance, one example being an application that uses different streams for control and data. Netbrake also does not distinguish between the two directions of data; incoming data will add to the same observed rate as outgoing data.

Netbrake is not semantically transparent (nor does it aim to be);

- it does not handle non-blocking I/O nor
- I/O multiplexing (`select()`, `poll()`, etc.) nor
- socket aliasing (`dup()`, etc.).

Trickle provides semantic transparency and the ability to provide fairness or managed differentiated bandwidth usage to different sockets or applications. Trickle also allows applications to cooperate as to retain (a) global bandwidth limit(s).

6 Future Work

Trickle does not have much control over how the lower layers of the network stack behave. A future area of exploration is to dynamically adjust any relevant socket options. Especially interesting is to adjust the socket send and receive buffers as to lessen the reaction time of Trickle's actions. Another area of future work is dynamic adjustment of smoothing settings, parameterized by various observed network characteristics and usage patterns (e.g. interactive, bulk transfers) of a particular socket.

There also exists a need for Trickle to employ more expressive and dynamic policies, for example adding the ability to shape by remote host or by protocol.

There are a few new and disparate interfaces for dealing with socket I/O multiplexing. In the BSD operating systems, there is the `kqueue`[18] event notification layer, Solaris has `/dev/poll`[13] and Linux `epoll`[19]. Trickle stands to gain from supporting these interfaces as they are becoming more pervasive.

By using a system call filter such as Systrace[22] or Ostia[17], Trickle could address its two highest impact issues. By using such a system call filter, Trickle could interposition itself in the system call layer, while still running entirely in userland, hence gaining the ability to work with statically linked binaries. Furthermore, these tools provide the means to actually enforce the usage of Trickle, thus enforcing bandwidth policies.

In order to do collaborative rate limiting when joining a new network, a user would have to manually find which host (if any) is running `trickled`. Trickle would thus benefit from some sort of service discovery protocol akin to DHCP[15]. Using Zeroconf[12] technologies could potentially prove beneficial.

7 Acknowledgments

The author would like to thank the following people for their sharp minds and eyes: Evan Cooke, Crispin Cowan (our shepherd), Kamran Kashef, Ken MacInnis, Joe McClain, Niels Provos (also for pushing and prodding to submit this paper), Andrew de los Reyes, Cynthia Wong, as well as the anonymous reviewers.

8 Availability

Trickle source code, documentation and other information is available under a BSD style license from

`http://monkey.org/˜marius/trickle/`

9 Summary and Conclusion

Trickle provides a practical and portable solution to ad-hoc rate limiting which runs entirely in userland. It has been shown to work extremely well in practice, and none of its inherent limitations seem to be a problem for its target set of users.

Since the time Trickle was released in March, 2003, it has enjoyed a steady user base. It is widely used, especially by home users in need of ad-hoc rate limiting. Trickle has also been used in research.

Trickle works by interpositioning its middleware at the BSD socket abstraction layer which it can do this entirely in userland by preloading itself using the link editor present in Unix-like systems. Trickle has been reported to work on a wide variety of Unix-like operating systems including OpenBSD[7], NetBSD[6], FreeBSD[3], Linux[10] and Sun Solaris[11], and is by its very nature also architecture agnostic.

At the socket layer the number of ways an operation can be manipulated is limited. Furthermore, we gain no access to lower layers in the network stack at which rate limiters typically reside, and so we have to rely on the semantics of TCP to cause reductions in bandwidth consumption. We developed several techniques including *smoothing* that help normalize the behavior observed in the lower network layers and avoids bursty throughput.

There are many venues to explore in the future development of Trickle, and we believe it will remain a very useful utility for ad-hoc rate limiting. Furthermore, we expect that this is the typical usage case for rate limiting by end users requiring occasional service differentiation.

References

[1] Apache. `http://www.apache.org/`.

[2] Azureus - java bittorrent client. `http://azureus.sourceforge.net/`.

[3] FreeBSD: An advanced operating system. `http://www.freebsd.org/`.

[4] Kazaa 3.0. `http://www.kazaa.com/`.

[5] Netbrake. http://www.hping.org/netbrake/.

[6] NetBSD: the free, secure, and highly portable Unix-like Open Source operating system. http://www.netbsd.org/.

[7] OpenBSD: The proactively secure Unix-like operating system. http://www.openbsd.org/.

[8] OpenSSH. http://www.openssh.com/.

[9] Rsync. http://samba.anu.edu.au/rsync/.

[10] The Linux Kernel. http://www.kernel.org/.

[11] The Sun Solaris Operating System. http://wwws.sun.com/software/solaris/.

[12] The Zeroconf Working Group. http://www.zeroconf.org/.

[13] ACHARYA, S. Using the devpoll (/dev/poll) interface. http://access1.sun.com/techarticles/devpoll.html.

[14] CHO, K. Managing traffic with ALTQ. In *Proceedings of the USENIX 1999 Annual Technical Conference* (June 1999), pp. 121–128.

[15] DROMS, R. Dynamic Host Configuration Protocol. RFC 2131, 1997.

[16] FRALEIGH, C., MOON, S., LYLES, B., COTTON, C., KHAN, M., MOLL, D., ROCKELL, R., SEELY, T., AND DIOT, C. Packet-level traffic measurements from the sprint IP backbone. *IEEE Network* (2003).

[17] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proc. Network and Distributed Systems Security Symposium* (February 2004).

[18] LEMON, J. Kqueue - a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference* (2001), USENIX Association, pp. 141–153.

[19] LIBENZI, D. /dev/epoll home page. http://www.xmailserver.org/linux-patches/nio-improve.html.

[20] M. ALLMAN, V. PAXSON, W. S. TCP Congestion Control. RFC 2581, Apr 1999.

[21] OKUMURA, T., AND MOSSÉ, D. Virtualizing network i/o on end-host operating system: Operating system support for network control and resource protection. *IEEE Transactions on Computers* (2004).

[22] PROVOS, N. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium* (Aug. 2003), USENIX, pp. 257–272.

[23] RIZZO, L. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev. 27*, 1 (1997), 31–41.

[24] ROSE, M. T. *The Open Book: A Practical Perspective on OSI.* Prentice Hall, 1990.

[25] S. BOTTOMS, T. L., AND WASH, R. nlimit: A New Voluntary Bandwidth Limiting Facility. Unpublished manuscript, 2002.

[26] TEAM, T. N. C. The Netfilter Project: Packet Mangling for Linux 2.4. http://www.netfilter.org/, 1999.

[27] TIRUMALA, A. End-to-end bandwidth measurement using iperf. In *SC'2001 Conference CD* (Denver, Nov. 2001), ACM SIGARCH/IEEE. National Laboratory for Applied Network Research (NLANR).

Notes

[1] A small software component used to provide an interface to another software component (a trivial “adapter”)

[2] Modern operating systems have introduced new and more efficient interfaces for multiplexing. These are discussed in section 6

[3] The author of Netbrake notes that Netbrake is intended to simply be a hack, and that it is not mature software.[5]