Experiences in Programming a Traffic Shaper

Dah Ming Chiu, Miriam Kadansky,
Joe Provino, and Joseph Wesley

Experiences in Programming a Traffic Shaper

Dah Ming Chiu, Miriam Kadansky,
Joe Provino, and Joseph Wesley

SMLI TR-99-77 September 1999

Abstract:

We report some experiences gained in programming a rate-based traffic shaper, seemingly a straightforward task. Specifically, we discuss how to compensate for an often-overlooked problem with inaccurate *sleep* functions, and how to accommodate intermittent data arrival streams. The objective is to produce an accurate, yet simple, robust, and efficient traffic shaper.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
dahming.chiu@east.sun.com
miriam.kadansky@east.sun.com
joe.provino@east.sun.com
joseph.wesley@east.sun.com

© 1999 Sun Microsystems, Inc. All rights reserved. The SMLTechnical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, Java, and the Sun logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.

Experiences in Programming a Traffic Shaper

Dah-Ming Chiu, Miriam Kadansky, Joe Provino, and Joseph Wesley
Sun Microsystems Laboratories
Burlington, Massachusetts

1. Introduction

The notion of traffic shaping has been discussed in several different contexts. One motivation is based on rate-based flow control in transport protocols, proposed by [Charny], among others. Rate-based flow control has been studied in the context of ATM's ABR (Available Bit Rate) service [ATM] and multicast transport protocols, e.g., [Handley] and [Chiu]. Some form of rate-based flow control is also incorporated directly into various video and audio applications, e.g., in [Rejaie]. These applications can adapt to network bandwidth availability by varying the content quality. As part of a rate-based transport, a traffic shaper is used at the sending host. Another application of traffic shaping is in the context of the differentiated services where traffic shapers are placed in strategic places inside the network. It is shown that if flows going through these traffic shapers (also referred to as traffic conditioners) are afforded with expedited forwarding treatment[1] by the rest of the routers, then the network is able to provide virtual leased bandwidth service [Nichols].

In all these models, the basic function of a traffic shaper is the same – convert an otherwise bursty flow of packets into a smoothened flow of traffic satisfying a desired (average) data rate. The concept of a traffic shaper is therefore very simply defined.

This paper is based on some experiences implementing a traffic shaper in the context of a rate-based flow control algorithm in a multicast transport protocol. The abstract set-up is illustrated in Figure 1.

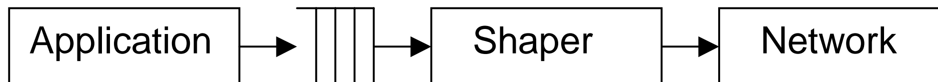

Figure 1: Traffic shaper

The application data is packetized and put into a data transmission queue. The traffic shaper takes the packets off the queue and passes them to the network layer with time gaps between successive packets, so that the average throughput is approximately *R,* the desired rate configured with the traffic shaper.

The objectives for the traffic shaper are therefore to ensure that:

- The average data rate is as close to R as possible.
- The burstiness is as low as possible.
- The algorithm is as simple and efficient as possible.

Since packetization is done before traffic shaping, it is assumed that the decision of what packet size to use is based on other considerations. The *burst rate* is defined as the rate of sending any subsequence of data

[1] Expedited Forwarding is a term used in differentiated services to describe a service level by routers and switches in the network. The intention of this service level is to forward packets without queuing delay.

packets. In its general form, the traffic shaper takes both a desired rate R, as well as a maximum burst rate R_{max} as its configuration parameters. For many applications, R_{max} is not a concern. So initially we ignore this parameter (assume it is infinity) and revisit this topic in a later section.

2. Differential Time Algorithms

The basic mechanism used by the traffic shaper to achieve a steady transmission rate is to sleep an appropriate time in-between sending packets. A simple model is based on the following send-sleep cycle:

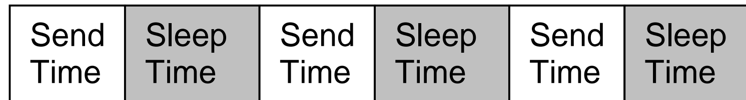

Figure 2: Send-sleep cycle

Here, the sendTime represents the processing time in sending a packet: queuing (by application), dequeuing (by traffic shaper) and other state processing tasks. For constant sendTime T, and packet size S, the sleepTime is computed as

$$sleepTime = \max\left(\frac{S}{R} - T, \quad 0\right)$$

The term "differential time" refers to the fact that we tried to measure the sendTime for each cycle. A naive Differential Time algorithm (DT0) is as follows[2]:

```
DT0:
    done = false;
    sendTime = 0;
    start = getTime();
    while (!done) {
        packet = queue.getPacket();
        socket.send(packet);
        size = packet.getSize();
        transmitTime = size / rate;
        sleepTime = transmitTime - sendTime;
        if (sleepTime > 0)
            sleep(sleepTime);
        end = getTime();
        sendTime = end - start - sleepTime;
        if (sendTime < 0)
            sendTime = 0;
        start = end;
    }
```

Figure 3: The naïve algorithm

[2] Written in pseudo code, with an emphasis on clarity rather than code efficiency. Our real scheduler and tests are all written in the Java™ programming language .

The duration is measured once per cycle, and the sendTime is determined by subtracting the sleepTime from the cycle duration. The rest of the algorithm is quite straightforward.

The expected performance of this algorithm can be described using the following two plots.

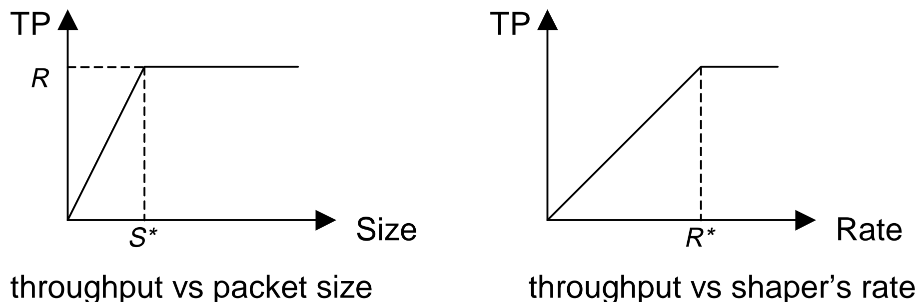

Figure 4: Expected performance of a traffic shaper

The first plots the throughput (or achieved rate) against different packet sizes, S, for a given (traffic shaper's) rate *R*. The second plots the throughput against *R*, for a given packet size *S*.

The value of sendTime is roughly a constant;[3] let this value be *T*. For small packet sizes, the achieved throughput is limited by the packet size divided by sendTime (*T*), represented by the sloped line of the first graph. In this region, the local host is sending packets back-to-back, and there is no sleeping. Only when the packet size reaches some value *S**, the traffic shaper is able to achieve the desired rate *R*, as represented by the flat part of the curve. The value of *S** is given by

$$S^* = RT$$

In the second plot, the throughput increases with the rate *R*, as represented by the *45°* line. This time, the throughput saturates at a certain rate *R** after which increasing the rate yields no higher throughput. This limit is caused by the same reason as above: only one packet of size *S* is sent for each cycle (the minimum time of which is sendTime). Therefore, the value of *R** satisfies

$$R^* = \min\left(\frac{S}{T}, R\right)$$

*R** can be considered as the maximum throughput of the sender for a given packet size *S*.

3. The Oversleeping Problem

DT0 was implemented as part of a transport protocol. It was not until we continuously experienced poor performance that we started to suspect a problem with DT0. The problem has to do with coarse granularity of the sleep function.[4] The actual sleep function on our experimental platform had a granularity of 10 milliseconds (for example, sleep(2 milliseconds) results in a sleep of 10 milliseconds) as illustrated by the following plot:

[3] With perhaps some small dependence on packet sizes.
[4] This is less of a problem when a real-time kernel is used.

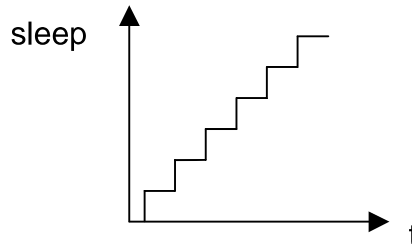

Figure 5: The behavior of the sleep() function

It is not unreasonable for many platforms[5] to implement the sleep function with a coarse granularity. There is an overhead in context switching from one thread (or process) to another; the coarse granularity protects the operating system from excessive context switching overheads.

To isolate the study of the traffic shaper, we wrote dummy getPacket() and send(packet) functions that result in a sendTime of 2 milliseconds (roughly equivalent to some measurements we had done earlier). The following two plots are based on these controlled measurements of the DT0 algorithm.

The throughput produced by the actual algorithm (DT0) deviated significantly from the (expected) ideal case. Once we know this is caused by the sleep granularity, the plots can be readily explained.

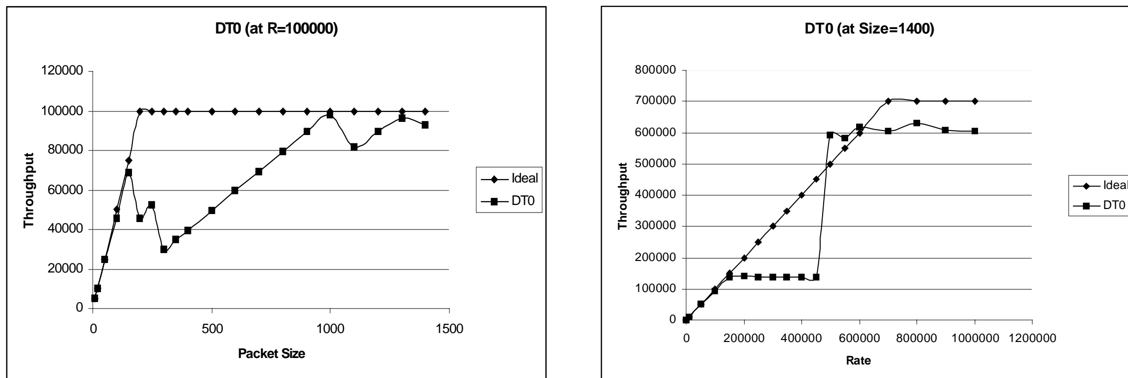

Figure 6: Measured performance of the DT0 algorithm

In the first plot, *R* is set to 100,000 (bytes/second) while packet size is varied. For values of *S* less than *R*T* (=200 bytes), there is no need to sleep, and the curve follows the ideal case quite closely. As soon as *S* is larger than 200, sleeps are introduced. Since the granularity of sleep is 10 milliseconds, any value of *S* that results in short sleeps (between 0 and 10 milliseconds) result in significant oversleeping. This explains the apparent large discrepancy when *S* is between 200 and 500.

In the second plot, the packet size is fixed at 1400 (bytes) while rate *R* is varied. For small values of R, the sleep times are much larger than the granularity of 10 milliseconds. For *R* less than 150,000 bytes/sec, the error caused by sleep granularity is apparently insignificant to cause much deviation from the ideal case. When *R* is above 150,000 bytes/sec, the sleep times are usually less then 10 milliseconds, and become smaller as *R* increases. Since they all result in 10 millisecond sleeps, it explains the apparent constant

[5] Especially when it is not a real-time kernel, or a dedicated server of some sort.

resultant throughput for a whole range of values of *R*. After *R* crosses over some threshold, there are less and less sleeps; hence, the throughput returns to the ideal curve.

To further support our explanation, the following bar charts show the total requested sleep time and total actual sleep time for sending 100 packets. The first chart of Figure 7 includes the cases when R is fixed at 100,000 bytes/sec and packet size is varied. For example, for packet size of 500, the total requested sleep time is 152 milliseconds, and the total actual sleep time is 819 milliseconds (i.e., 667 milliseconds of oversleeping). The second chart of Figure 7 includes the corresponding cases when packet size is fixed at 1400 bytes and R is varied.

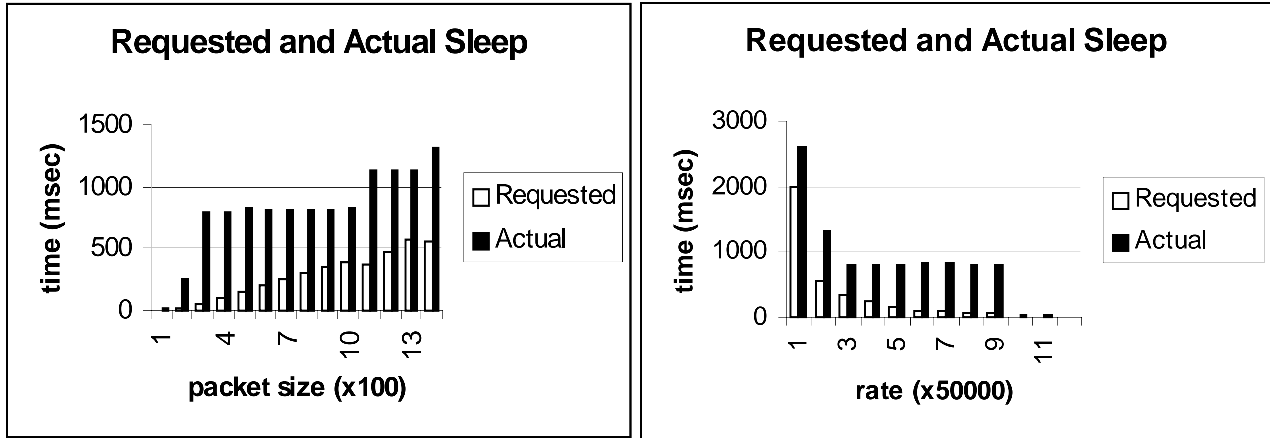

Figure 7: Sleep and overslept times

An interesting revelation from Figure 7 (a) is that oversleeping does not seem to affect throughput for packet sizes sufficiently large (>900 bytes). For packet size of 1000 bytes, very close to ideal throughput is achieved while there is 400 milliseconds of oversleeping (as much as the total sleep time). This is because the way sendTime is calculated (in DT0) actually absorbs the oversleeping time, so that the oversleeping time in one cycle goes to reduce the sleepTime calculated in the subsequent cycle.

In the packet size of 1000 bytes case, the elapsed time to send 100 packets at a rate of 100,000 bytes/second should take 1 second. The emulated sendTime is 2 millisecond/packet, resulting in a total CPU time of 200 milliseconds to send 100 packets. The total actual sleep time as shown in Figure 7(a) is roughly 800 milliseconds, which is exactly what is needed to produce an average data rate of 100,000 bytes/second. This explains why the throughput for this case is near the ideal value.

The way the calculated sendTime compensates oversleeping in DT0 is, however, far from perfect. For relatively small packet size cases (200 to 900 bytes), the required sleeping time to maintain the desired rate (of 100,000 bytes/second) is relatively small. For example, if the packet size is 500 bytes, at the desired rate of 100,000 bytes/second, a packet should be transmitted every 5 milliseconds. The CPU sendTime is 2 milliseconds; thus the required sleep time would be 3 millisecond. The 7 milliseconds of oversleeping cannot be compensated even if sleeping is skipped in the subsequent cycle. The rest of the paper discusses various algorithms to properly compensate for oversleeping.

4. Correcting for Oversleeping

Oversleeping can be simply compensated by sending multiple packets after each sleep. The send-sleep cycle thus becomes:

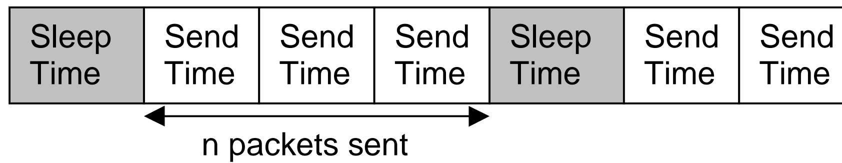

Figure 8: Send-sleep cycles with correction for oversleeping

In each cycle, n packets are sent so that, roughly

$$n = \frac{RG}{S - RT}$$

Where G is the granularity (in our case 10 milliseconds) of sleep. When the value of S and R are such that

$$1 \leq \frac{S}{R} - T < G$$

By incorporating this idea, DT0 becomes DT1 as follows.

```
DT1:
    done = false;
    start = getTime();
    lostTime = 0;
    while (!done) {
        packet = queue.getPacket();
        socket.send(packet);
        size = packet.getSize();
        transmitTime = size / rate;
        end = getTime();
        sendTime = end - start;
        sleepTime = transmitTime - sendTime;
        if ( (lostTime>0) && (sleepTime>0) ) {
            if ( lostTime > sleepTime ) {
                lostTime = lostTime - sleepTime;
                sleepTime = 0;
            } else {
                sleepTime = sleepTime - lostTime;
                lostTime = 0;
            }
        }
        if (sleepTime > 0) {
            wakeUpAt = getTime() + sleepTime;
            sleep(sleepTime);
            end = getTime();
            ostTime = end - wakeUpAt;
            if (lostTime < 0 )
                lostTime = 0;
        }
        start = end;
    }
```

Figure 9: An algorithm that corrects oversleeping, DT1

The simple assumption in DT1 is that time can be measured (by getTime) more accurately than the accuracy of sleep. On the platform we experimented, the granularity of getTime is 1 millisecond and sleep is 10 milliseconds.

Algorithm DT1 measures the duration of the sleepTime and sendTime for each send-sleep cycle. A slight modification to DT1 requires duration measurement only once per send-sleep cycle, as in DT2.

While DT2 is a little simpler than DT1, it also gives up something. There is no attempt to avoid sleeping at all, but simply trying to make up for oversleeping. Therefore, DT2 is inevitably more bursty than DT1.

```
DT2:
    done = false;
    start = getTime();
    lostTime = 0;
    size = 0;
    while (!done) {
        packet = queue.getPacket();
        socket.send(packet);
        size += packet.getSize();
        transmitTime = size / rate;
        sleepTime = transmitTime;
        if ( (lostTime>0)) {
            if ( lostTime > sleepTime ) {
                lostTime = lostTime - sleepTime;
                sleepTime = 0;
            } else {
                sleepTime = sleepTime - lostTime;
                lostTime = 0;
            }
        }
        if (sleepTime > 0) {
            sleep(sleepTime);
            end = getTime();
            lostTime = end - start - sleepTime;
            if (lostTime < 0 )
                lostTime = 0;
            start = end;
        }
    }
```

Figure 10: Another algorithm correcting oversleeping, DT2

5. Absolute Time Algorithms

The algorithms considered so far are all differential time algorithms. Instead of measuring durations for each send-sleep cycle, it is also possible to measure elapsed time from the beginning of the transmission. The resultant algorithm is based on absolute time.

The use of absolute time seems to considerably simplify the traffic shaper, as demonstrated by algorithm AT0 below

```
AT0:
    done = false;
    dataSent = 0;
    start = getTime();
        while (!done) {
            packet = queue.getPacket();
            socket.send(packet);
            size = packet.getSize();
            dataSent = dataSent + size;
            next = (dataSent/rate) + start;
            sleepTime = next - getTime();
            if (sleepTime > 0)
                sleep(sleepTime);
        }
    }
```

Figure 11: An algorithm based on using absolute time

The following plots show the measured performance of DT1, DT2, and AT0, under the same settings as the earlier measurements. They validated the earlier analysis.

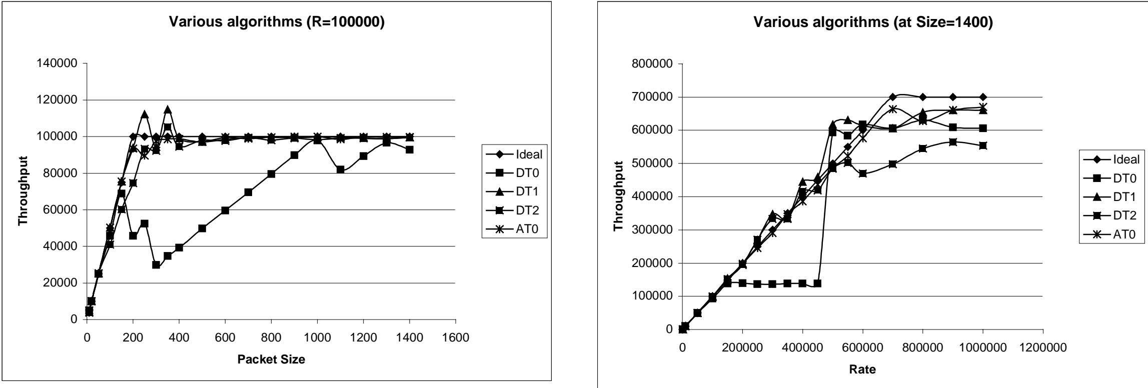

Figure 12: Comparison of performance of DT1, DT2 and AT0

6. Session Rate and Data Gaps

The contrast of AT0 and our earlier differential time algorithms brings our attention to a subtle point that was not stated in our original objectives statement in section 1. The first objective is to send data at a rate as close to R (given rate) as possible, on average. So the given rate, R, is implied to be a data rate for the whole session to achieve.

It turns out there are two legitimate models for application data arrivals. The implicitly assumed model is that all application data arrive at the beginning of the session – this is the file transfer model. In the other model, application data arrive piecemeal with gaps in-between different fragments. A real-world example of this is the distribution of news as they occur. We call this application model the live-data model.

With the file transfer model, it makes sense to use the given rate R as a session rate. With the live-data model, however, it does not make sense any more. Consider the following example: 10K bytes of data arrive at the beginning of every 10-second interval. The given data rate R is equal to 2K bytes per second. Using the session rate objective, the first 10K bytes will be sent smoothly over 5 seconds. By the time we get to the second 10K bytes, we will be sending those packets back-to-back since the session rate has been 1K bytes/second thus far. This is the case for all subsequent data fragments.

This example suggests that if there are relatively long gaps in the application data arrivals, the given session rate *R* should be applied to each fragment of data separately.

One side feature of algorithms DT0 and DT1 is that they already support this live-data model adequately. In both DT0 and DT1, a measured sendTime is subtracted from the transmitTime to derive the sleepTime. When there is a significant gap in application data arrivals, the measured sendTime will be bigger than the value of transmitTime, and hence will cause the sleepTime to be set to zero. No matter how large this one sendTime (i.e., gap) is, it does not affect future scheduling decisions. Effectively, after each gap the traffic shaper is re-initiated, as desired by the live-data model.

7. Accommodating Live-data Applications

Both AT0 and DT2 can be modified to accommodate for live-data applications. We describe one simple modification to AT0, as in AT1, below. This technique can be applied to DT2 in a similar manner.

```
AT1:
    done = false;
    dataSent = 0;
    start = getTime();
    while (!done) {
        if ( queue.notEmpty() ) {
            packet = queue.getPacket();
        } else {
            blockAt = getTime();
            packet = queue.getPacket();
            now = getTime();
            blockTime = now - blockAt;
            if ( blockTime > threshold ) {
                start = now;
            }
        }
        socket.send(packet);
        size = packet.getSize();
        dataSent = dataSent + size;
        next = (dataSent/rate) + start;
        sleepTime = next - getTime();
        if (sleepTime > 0)
            sleep(sleepTime);
    }
```

Figure 13: Modified version of AT0 to support live-data applications

This modified version of AT0 measures the gap as the time getPacket takes (whenever the data queue becomes empty). When the gap exceeds a threshold, we assume it is the start of a new data fragment in the live-data model; the start time is reset, hence the algorithm is re-initialized. In DT0 and DT1, the value of threshold is implicitly set to be the latest transmitTime.

The following figure illustrates the measured performance of AT0 and AT1 for the example live-data model discussed earlier.

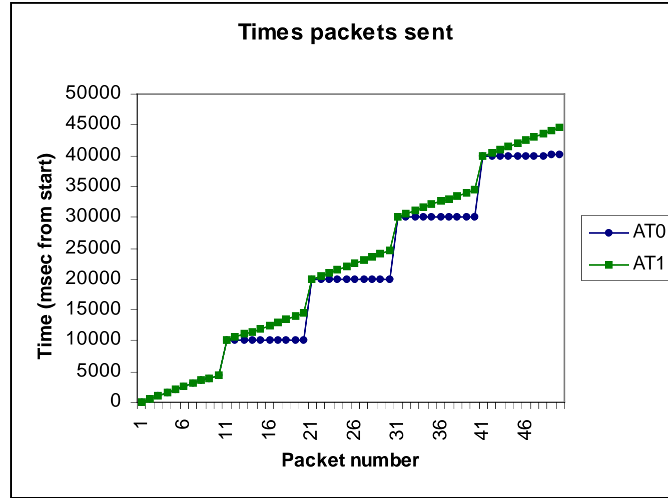

Figure 14: Comparison of AT0 and AT1

The packet size is 1000. Therefore 10 packets (10K) of data arrive at the beginning of each 10 second interval. Using either algorithm, the first 10 packets are sent at smoothly, with approximately a 500-millisecond gap between successive packets (since the rate is 2K bytes per second). Starting with the packets arriving after at 10 seconds, AT0 sends them at a very high instantaneous rate (with hardly any gap in between successive packets). AT1 behaves correctly by sending the packets with approximately 500-millisecond gaps.

8. Burst Rate

As we mentioned in the introduction, one of the objectives for the traffic shaper is to minimize the burstiness of data transmission. One way to approach this is to specify a maximum burst rate R_{max} for the traffic shaping algorithm to enforce. This mechanism can be added to each of the above algorithms in a similar manner.

This is one method of implementation. Two accounting variables are introduced: *burstStart* and *burstSent.* These variables are initialized at the beginning, and re-initialized after each sleep. Each time sleepTime is calculated, a pauseTime is also calculated based on *burstStart* and *burstSent.* The value of pauseTime is the time to sleep due to the burst of packets. If pauseTime is greater than sleepTime computed earlier, then sleepTime is reset to pauseTime. This update to the algorithms is shown below:

```
…
burstStart = start;
burstSent = 0;
while (!done) {
    …
    sleepTime = …;
    burstSent = burstSent + size;
    burstTime = burstSent / burstRate;
    now = getTime();
    pauseTime = burstTime - (now – burstStart);
    if (pauseTime > sleepTime) {
        sleepTime = pauseTime;
        burstStart = now;
        burstSent = 0;
    }
    …
}
```

Figure 15: Modification to control burst rate

9. Conclusions

In this paper, we reported some experiences in programming a rate-based traffic shaper. The most interesting result was the discovery of a bug in not correctly handling the use of the sleep function. It is interesting because a small loss of granularity translates into a relatively large loss in performance, and it is easy to overlook. We also discussed the need to accommodate different data arrival models and burst rate constraints, and presented modified algorithms to meet these requirements.

10. Acknowledgements

The authors thank Guy Steele for many insightful comments, in particular for suggesting that we look at the absolute time algorithms.

11. Reference

[Charny] Anna Charny, “An Algorithm for Rate Allocation in a Packet-Switching Network with Feedback”, Masters thesis, MIT 1994.

[ATM] ATM Forum Traffic Management 4.1 Specification, ATM Forum/af-tm-0121.000, March 1999.

[Handley] Mark Handley and Sally Floyd "Strawman Specification for TCP Friendly (Reliable) Multicast Congestion Control (TFMCC)", Memo to Reliable Multicast Research Group, December 1998.

[Chiu] Dah Ming Chiu et al., "TRAM: A Tree-based Reliable Multicast Protocol", Technical Report, Sun Labs, 1998

[Rejaie] Reza Rejaie, et al., “Quality Adaptation for Congestion Controlled Video Playback over the Internet”, ACM SIGComm99 Proceedings, Sept 1999.

[Nichols] Nichols, K., et al., “A Two-bit Differentiated Services Architecture for the Internet”, RFC 2638, July 1999.

About the Authors

Dah Ming Chiu is a researcher at Sun Microsystems Laboratories in Burlington, Massachusetts. His recent research is on reliable multicast, and multicast flow and congestion control. Prior to Sun, he worked at Digital Equipment Corporation, and AT&T Bell Labs. His research interests include performance modeling and analysis of network protocols, distributed systems, and WWW-based applications. He received a Ph.D. degree from Harvard University and a B.Sc. degree from Imperial College, University of London.

Miriam Kadansky is a Senior Staff Engineer at Sun Microsystems Laboratories in Burlington, Massachusetts. She has a B.A. in Mathematics from Harvard College, and worked for several networking vendors before joining Sun. Her current interests include secure reliable multicasting and Java.

Joe Provino is a Senior Staff Engineer at Sun Microsystems Laboratories in Burlington, Massachusetts. He is currently a member of the Reliable Multicast team. He has worked for Sun since 1985 and has worked on PC-NFS, Solaris, and PC-SKIP. Provino has a Bachelor's degree in Mathematics from the Rochester Institute of Technology and a Master's degree in Computer Science from the University of California at Berkeley. His interests include operating systems, network protocols, and security.

Joseph S. Wesley is a Member of Technical Staff at Sun Microsystems Laboratories in Burlington, Massachusetts. His current interests include reliable multicast and multicast security. Prior to joining Sun, he worked in the router development group at BBN Systems and Technologies at Cambridge, Massachusetts. He has a M.S. in Computer Engineering from the University of Massachusetts, Lowell, and a B.E. in Electronics and Communications from the University of Mysore, India.