Scheduling support for Hard Real-Time Ethernet Networking

Jork Loeser
TU Dresden, Germany
jork@os.inf.tu-dresden.de

Jean Wolter
Dresden, Germany
jw5@os.inf.tu-dresden.de

Abstract

Previous results on traffic shaping on Switched Ethernet technology have demonstrated its practicability and effectiveness for hard real-time communication [6]. With nodes dedicated to communication the application-to-application delays on a 5-node network were reported to be less than a millisecond with both Fast Ethernet and Gigabit Ethernet technology with a link utilization of 93% and 49%.

In this paper we extend the work and analyze the network performance that can be achieved on complex systems where resources for the network management are restricted by other real-time tasks. Specifically, we analyze the scheduling needs of different implementations of the traffic shaper, a key component of hard real-time networking on Switched Ethernet. We derive the performance of the different implementations with respect to achievable network delays and the implementation costs with respect to CPU utilization.

1 Motivation

In [6] we demonstrated the practicability and effectiveness of using software-implemented traffic shaping on Switched Ethernet to achieve hard real-time communication with bounded delays and guaranteed bandwidths. To do so, we used the DROPS real-time system [3, 2], which offers static priorities. We implemented a real-time network driver which performs the required traffic shaping ensuring that the nodes do not flood the network. The threads of the network driver were given the highest priorities in the system preventing interference of other applications and achieving minimal scheduler-induced delays. However, in modular and dynamic systems where real-time applications apply for being started and stopped at arbitrary points in time, resource consumption must be bounded and threads must be scheduled accordingly. In this paper, we propose different traffic shaper implementations to ensure the nodes conformance to their network traffic contracts. We analyze the scheduling needs of these implementations and estimate their performance with respect to achievable network delays and the implementation costs with respect to CPU utilization.

2 Background

Switched Ethernet is a star-based topology providing a private collision domain to each of the ports of a switch. Collisions, a phenomenon of CSMA/CD Ethernet, do not occur. However, because Ethernet switches lack build-in policing features, nodes connected by Switched Ethernet need to be cooperative for bandwidth and jitter control. The key idea is to make use of buffers at the switches, that compensate for concurrent arrival of data from different nodes destined for the same target. Figure 1 shows a typical Ethernet switch. The switch has N=4 receive ports, control logic, buffer and N queued transmit ports. When a frame arrives at the switch, the control logic determines the transmit port and tries to transmit the frame immediately. If the port is busy because another frame is already being sent, the frame is stored in the transmit ports queue, which is a first-in first-out (FIFO) queue. If no more memory is available for storing, the received frame is dropped.

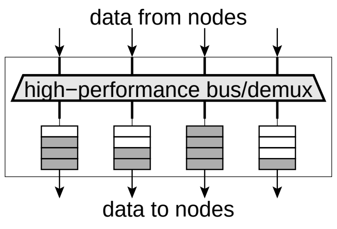

Figure 1: Buffering inside an output-queueing Switch. If queueing a frame is necessary, memory is allocated from a shared memory pool and assigned to the corresponding queue.

Bounding delays

The buffered packets in the switch result in a delay, which is specific to each output port and depends on the characteristics of the traffic sent to that output port. We use Le Boudec's network calculus [1] to calculate the delay bound for a specific output port. For this, we describe the traffic arriving at a specific switch input port k to be forwarded to this output port by a so-called T-SPEC (C, M, r_k, b_k). C is the network capacity, this is 100MBit/s for Fast Ethernet and 1000Mbit/s for Gigabit Ethernet. M is the maximum frame size, 1514 bytes for Ethernet. r_k describes the average bandwidth and b_k allows for some burstiness. The T-SPEC means that in any time interval of length t not more than $\min(C*t+M, r_k*t+b_k)$ bytes arrive at input port k for the considered output port.

For all $k = 1 \dots N$ we define g_k as $g_k = \frac{b_k - M}{C - r_k}$ and g_{max} as the maximum of all g_k. t_{mux} denotes a switch-specific parameter describing the maximum delay (without queueing effects) after which the switch starts to transmit a frame once it is received.

According to [5] the maximum delay t_{switch} of a frame for the considered output port at the switch is

$$t_{switch} \leq \sum_{k=1}^{N} \frac{b_k}{C} - g_{max} * (1 - \sum_{k=1}^{N} \frac{r_k}{C}) + t_{mux}. \qquad (1)$$

Shaping the traffic

With Ethernet, nodes must cooperate to ensure that traffic leaving a node conforms to previously defined T-SPECs. Therefore, all *sending nodes* apply traffic shaping to all transmitted data.

We implemented the network access model in our own network stack, but extended the stack to manage multiple connections at each node, with one traffic shaper per connection. This way the network stack can give different bandwidth guarantees to the various applications at one node. Figure 2 illustrates this: The real-time network driver (RT-Net driver) directly interacts with the network interface card (NIC). It uses per-connection traffic shapers running in separated threads. To its clients the driver offers connection-oriented packet-based interfaces that account the transmitted traffic. The abstraction seen by the clients is that of a leased line: a client i may send its packets with a maximum bandwidth b_i. Packets exceeding the bandwidth might be queued and being sent later instead of being sent immediately.

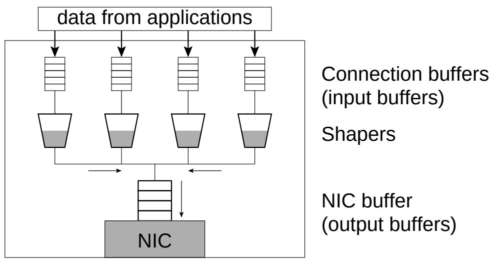
data from applications
Connection buffers (input buffers)
Shapers
NIC buffer (output buffers)
NIC

Figure 2: Multiplexing of multiple shaped connections to one NIC.

Scheduling model

The DROPS real-time system is based on an L4 type microkernel which provides three basic services: (i) address spaces as protection domains, (ii) threads as abstractions of activities and (iii) synchronous IPC as communication primitive. Everything else is implemented at user level. The microkernel uses a fixed priority scheduling scheme with round robin scheduling within the same priority class. It provides a capacity reserves like reservation mechanism by allowing to attach priority/quantum/period tupels (p_i, w_i, T_i) to threads[7]. At the begin of each period T_i the kernel sets the priority of the thread to p_i and allows the thread to run on this priority for w_i time units. Before the end of their time quantum, threads can voluntarily yield the CPU and wait for the next period. Admission is done by a user-level admission controller, which assigns the (p_i, w_i, T_i) tuples and can guarantee deadlines $D_i < T_i$ using response time analysis.

The microkernel supports three different execution models: strictly periodic threads, periodic threads and aperiodic threads (using the terminology of [4]). While strictly periodic threads are always released at the begin of their period, periodic threads have a minimal inter-release time after which they can be released by the arrival of a wakeup event (IPC)[1].

3 Traffic Shaper implementations

Figure 2 gives a general idea of traffic shaping at a node: The sending applications put packets into connection buffers and signal the availability of the packets to the connection-specific shaper. Depending on the scheduling model the shaper either periodically looks into its connection buffer or wakes up after the client submitted a packet. If packets are available the shaper takes some of them and moves them to the output buffer of the network card.

Definition: For the rest of the paper, we call a packet that is sent by the client of connection *i conforming* if its timely distance to the previous sent packet corresponds to the bandwidth b_i of the connection.

Note that if the client suffers a scheduling jitter J_i it may produce non-conforming packets accidentally. These packets will experience an additional delay of up to J_i to fit to the leased line model. Having this in mind, we do not consider this client-induced delay any longer, but concentrate on conforming packets.

Performance of shapers There are multiple ways to implement traffic shapers that generate a token-bucket shaped stream of rate r_i. However, the streams may differ in their *burstiness* b_i, which can be calculated by

$$b_i = max_{t>s}\{\alpha(t) - \alpha(s) - r_i * (t-s)\} \qquad (2)$$

with $\alpha(x)$ being the amount of data produced by time x (the term in the braces is the difference between the amount of data actually sent in this interval and the bucket replenishment in this interval). From Section 2 we know that this *burstiness parameter* essentially influences the delays of the other streams at the switch, and hence it is an important performance measure. Another performance measure is the *maximum delay* of conforming packets at the shaper.

CPU utilization Another important parameter of the shaper implementation is its CPU usage and scheduling requirements. As the shaper is run in its own operating system context, frequent changes between the applications and the shaper severely influence the CPU usage. As shown in [6], the CPU utilization is dominated by the number of shaper invocations: A node sending with a bandwidth of 32MBit/s used its CPU to 9% with a shaper-invocation every 1ms, but only to 2.9% when the shaper was called every 10ms to send larger chunks.

[1]If the wakeup event comes in to early the kernel delays the release until the minimal inter-release time is over.

The scheduling priority of a shaper thread derived from its maximum deadline is of importance, too. When the shaper thread is assigned a high priority to achieve low scheduling delays, other threads in the system may suffer high scheduling delays.

3.1 Shaper versions

In the following, we analyze three traffic shaper implementations:

Strictly periodic shaper This most basic form of a traffic shaper runs strictly periodically and allows one packet to pass in each period.

Periodic shaper with data dependency Similar to the strictly periodic shaper this shaper sends one packet per invocation. But, instead of a strict period it has a minimal inter-release time. Whenever a packet is ready in the connection buffer, and the minimal inter-release time has elapsed since the last invocation, the shaper is started and sends a packet.

Token-bucket shaper This shaper runs strictly periodically while managing a bucket containing tokens for sending packets. The bucket is replenished on each invocation by some amount and the shaper sends up to as much data as there are tokens in the bucket.

For each shaper, we analyze (i) the maximum delay it adds to a conforming packet, and (ii) the burstiness parameter b_i of the generated stream.

3.2 Strictly periodic shaper

The straight-forward shaper implementation is a strictly periodic thread that sends up to one packet per invocation:

```
strictly-periodic-shaper(T_i, D_i, M) {
  set_periodic(T_i)
  while (1) {
    p = next_packet(); /* 0 of none avail */
    if (p!=0) send_packet(p);
    next_period();
  }
}
```

With T_i we denote the period of the thread and with D_i its deadline ($D_i \leq T_i$). To generate a stream with rate r_i and packet size M, T_i must be set to

$$T_i = M/r_i \tag{3}$$

Figure 3 illustrates the maximum delay that is induced by the traffic shaper to a conforming packet: The traffic shaper is activated early in the first period, and the packet is sent just after this. In the next period, the traffic shaper is activated as late as possible to just meet the deadline. Thus, the maximum delay is:

$$d_i = T_i + D_i \tag{4}$$

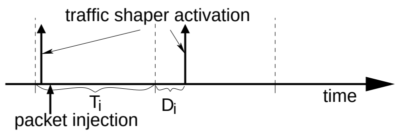

Figure 3: Maximum delay of a packet at the traffic shaper.

The maximum burst of the generated stream corresponds to two consecutive packets sent in their minimum distance. This is $T_i - D_i$, as illustrated in Figure 4.

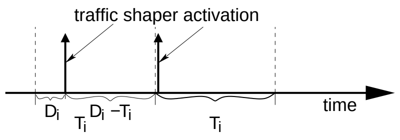

Figure 4: Maximum burst of the stream generated by the strictly periodic shaper.

According to Equation 2, the burstiness parameter of the generated stream can be calculated as

$$\begin{aligned} b_i &= 2*M-(T_i-D_i)*r_i = 2*M-T_i*ri+D_i*r_i \\ b_i &= M+D_i*r_i \end{aligned} \tag{5}$$

Thus, the generated stream conforms to a $(\mathbf{r_i}, \mathbf{M}+\mathbf{D_i}*\mathbf{r_i})$ token-bucket shaper.

3.3 Periodic shaper with data dependency

If an application is not executed in-phase with the shaper, that means it can not guarantee that a data packet is generated immediately before the shaper is activated, the worst-case delay of the strictly periodic shaper is more than one period. Modifying the shaper to wait until data is available avoids an out-of-phase client to miss the send operation of the current send period (Figure 5). With T_i we now denote the minimal inter-release time of the thread.

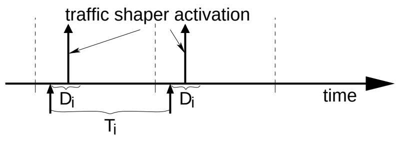

Figure 5: Maximum delay induced by the periodic shaper with data dependency.

As the traffic shaper thread is guaranteed to be finished within D_i after getting a conforming packet from its client, the maximum delay d_i that is induced by the traffic shaper is

$$d_i = D_i \tag{6}$$

The burstiness parameter of the generated stream is calculated the same way as for the strictly periodic shaper.

3.4 Strictly periodic shaper with long periods

To lower the CPU load due to context switches, the period of the traffic shaper thread can be increased at the cost of larger bursts and thus larger delays. This modification also favors systems were the thread period lengths are fixed or harmonic.

The naive approach is to modify the strictly periodic shaper to send not just one but up to a certain number of packets per invocation. However, this results in a very coarse granularity of possible bandwidth reservations: For Fast Ethernet and a period of T_i=1ms, the difference between sending n packets of 1514 bytes per 1ms period and sending $n+1$ packets per period accumulates to 12.1MBit/s or 1/8 of the overall bandwidth.

3.5 Token-bucket shaper

A token-bucket shaper avoids the coarse bandwidth granularity problem identified in the previous section:

```
token-bucket-shaper(r_i, M, B_i, T_i) {
  set_periodic(T_i) ; level = B_i ;
  while (1) {
    p = next_packet(); /* 0 of none avail */
    while(level < M || p==0) do {
      next_period() ;
      level = min(level + r_i*T_i, B_i) ;
      if(!p) p = next_packet();
    }
    if(p) { send_packet(p) ; level -= M ; }
} }
```

The input parameters to the algorithm are the rate r_i, the packet size M, the bucket size B_i, the deadline D_i and the period T_i. Note that with the token-bucket shaper, T_i is typically larger than it is for the shapers from Sections 3.2 and 3.3.

To obtain the worst-case delay of a conforming packet we argue the same way as we did in Section 3.2, and thus get

$$d_i = T_i + D_i \tag{7}$$

Figure 6 illustrates the burstiness bound of the generated stream: In the first period the maximum amount of data is sent as late as possible, this is B_i bytes at offset D_i within the period. The following transmission occurs as early as possible in the period, this is at offset 0, and sends $T_i * r_i$ bytes. Thus, the burstiness parameter is calculated as

$$\begin{aligned} b_i &= B_i + T_i * r_i - r_i * (T_i - D_i) \\ b_i &= B_i + r_i * D_i \end{aligned} \tag{8}$$

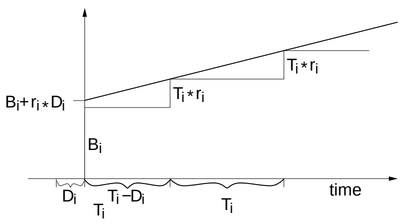

Figure 6: Obtaining the burstiness parameter of the stream generated by the token-bucket shaper.

Minimum bucket size for the Token-bucket shaper

The bucket must have at least a size that all tokens that may arrive between the blocking of the shaper (due to missing tokens) and its next activation fit into it. When the shaper blocks, at most M tokens are in the bucket. At the next period the bucket is replenished by $r_i * T_i$ and this must fit into it. Thus, the minimum bucket size is

$$B_i = r_i * T_i + M \tag{9}$$

4 Comparison

To give an idea on the impact of the different shapers to the app-to-app delay, we calculated the maximum time needed for a conforming packet to be processed by the traffic shaper, to be sent to the switch and to be processed by the switch and forwarded to the destination node. We assumed 5 nodes sending traffic with 16MBit/s to a 6th node over Fast Ethernet. t_{mux} was set to 45µs as in [6]. C is 12325 bytes/s considering the framing overhead with 1514 byte frames. For the Token-bucket shaper we choose periods of T_i=1ms and T_i=10ms for comparison. The periods of the other shapers are calculated by Equation 3. We selected deadlines of D_i=200µs and D_i=T_i for comparison.

Shaper	T_i	D_i	$d_i + 123\mu s + t_{switch}$
Strictly periodic	0.76ms	200µs	1.89ms
		T_i	2.88ms
Periodic with data dependency	0.76ms	200µs	1.13ms
		T_i	2.12ms
Token-bucket	1ms	200µs	2.91ms
		T_i	4.33ms
	10ms	200µs	18.88ms
		T_i	36.28ms

The table illustrates that low scheduling delays and high invocation frequencies result in moderate networking delays. However, if the scheduling delays increase, e.g. due to other high-priority tasks in the system, or if the invocation frequency of the shapers is decreased to lower the CPU consumption, the application-to-application-delay increases significantly.

References

[1] J.-Y. Le Boudec and P. Thiran. *Network Calculus*. Springer Verlag Lecture Notes in Computer Science volume 2050, July 2001.

[2] C.-J. Hamann, J. Löser, L. Reuther, S. Schönberg, J. Wolter, and H. Härtig. Quality Assuring Scheduling - Deploying Stochastic Behavior to Improve Resource Utilization. In *22nd IEEE Real-Time Systems Symposium (RTSS)*, London, UK, December 2001.

[3] H. Härtig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, June 1999.

[4] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[5] J. Loeser. Buffer Bounds of a FIFO Multiplexer . Technical Report TUD-FI03-15, Technische Universität Dresden, November 2003.

[6] J. Loeser and H. Haertig. Low-latency hard real-time communication over switched ethernet. In *16th Euromicro Conference on Real-Time Systems*, Catania, Sicily, July 2004.

[7] Udo Steinberg. Quality-Assuring Scheduling in the Fiasco Microkernel. Master's thesis, TU Dresden, March 2004.