Experimental Results for Class-Based Queueing

Sally Floyd* and Michael Francis Speer

November 11, 1998

This is a preliminary and incomplete draft of a paper in progress.

1 Introduction

This paper describes the experience of implementing CBQ's top-level link sharing code as described previously in [FJ95]. This implementation incorporates work from previous implementations of Class-Based Queueing from LBNL [Jac95] and UCL [WGC+95]. This implementation extends the work of previous implementations by incorporating both Top-Level link-sharing and Weighted Round Robin within priority levels of the link-sharing structure.

As discussed in [FJ95] and [Flo97], the use of Top-Level instead of Ancestor-Only link-sharing allows a class to receive its allocated bandwidth more accurately from a CBQ implementation.

Similarly, as discussed in [FJ95] and [Flo97], weighted round robin (WRR) has two advantages over packet-by-packet round robin (PRR) scheduling within a priority level. First, WRR gives better worst-case delay behavior than PRR scheduling for higher-priority classes. Second, WRR scheduling allows excess bandwidth to be distributed among classes in a priority level according to the bandwidth allocations of those classes.

2 Description of CBQ Implementation

2.1 General CBQ description

Before discussing this CBQ implementation in detail, it is important to note that this CBQ implementation is built utilizing many components. At the high level CBQ is not just a packet scheduler; it is a link-sharing resource manager. In principle, CBQ's link-sharing could be implemented in conjunction with a number of different packet scheduling algorithms within a priority level, such as Deficit Round Robin, Weighted Fair Queueing, or Fair Queueing. This implementation utilizes an implementation of Weighted Round Robin (WRR) and/or Packet-by-Packet Round Robin (PRR) scheduling. Compared to other general scheduling algorithms, these two schedulers seem to be the least expensive in computational complexity. We discuss the details of these scheduling algorithms later in the paper.

2.2 Principles Applied in CBQ Implementation

This CBQ implementation follows many principles previously outlined in [FJ95] to allow for the maximum flexibility.

First, this CBQ implementation continues to maintain a separation of low-level mechanisms and high-level policy. The CBQ kernel code provides a rich interface to implement variety of high-level policies, including, if so desired, RSVP and Integrated Services.

Second, the CBQ implementation preserves the link-sharing model presented in [FJ95]. Link-sharing resources are associated with CBQ traffic classes, where each CBQ traffic class has a bandwidth allocation and a priority. It is left to the high-level policy daemon to make decisions about *how* to allocation bandwidth and priorities to the various classes. The CBQ implementation is able to handle Quality-of-Service (QOS) and link-sharing constraints simultaneously.

Finally, the CBQ implementation avoids the need for extensive per-conversation parameterization. Hence, this CBQ implementation is able to separate IP conversations (flows) into classes with a minimum of class and filter parameterization.

2.3 CBQ Implementation

Major components of this implementation for CBQ's top-level link sharing include a packet classifier, link-sharing framework, packet scheduler, estimator, and management interface. The packet classifier maps arriving packets into traffic classes. The link-sharing framework is needed to maintain link-sharing constraints for an interface (e.g. an output port) with a hierarchical link-sharing structure. The packet scheduler schedules traffic classes according to their bandwidth and priority considerations, with help from the estimator. The management interface allows for the creation and deletion of traffic classes; the creation and deletion of packet classifier filters to map IP flows to the appropriate traffic classes; and a simple statistical interface for inspection of CBQ's current state.

*This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

In this CBQ implementation, the link-sharing framework is implemented using Top-Level Link-Sharing, described in [FJ95]. Previous implementations implemented Ancestor-Only link-sharing.

In this CBQ implementation, the packet scheduler or selector is implemented with either a packet-by-packet round robin (PRR) or weighted round robin (WRR) scheduler. The scheduler uses priorities, first scheduling packets from the highest priority level. Round-robin scheduling is used to arbitrate between traffic classes within the same priority level. The weighted round robin scheduler differs from the packet-by-packet scheduler in that it uses weights proportional to a traffic class's bandwidth allocation. The weight determines the number of bytes that a traffic class is allowed to send during a round of the scheduler. If a packet to be transmitted by a WRR traffic class is larger than the traffic class's weight and the class is underlimit (via link-sharing constraints), then the packet is sent, allowing the traffic class to borrow ahead from its weighted allotment for future rounds of the round-robin. The implementation of the WRR scheduler largely follows that of the CBQ code in the "ns" simulator [MF95]. In the implementation of the CBQ code the scheduler components are implemented as the functions *_rmc_prr_dequeue_next* for PRR and *_rmc_wrr_dequeue_next* for WRR, both in the file rm_ class.c [FS97].

When a traffic class is overlimit and unable to borrow from parent classes, the scheduler activates the overlimit action handler for that class. There are many policies that could implemented for an overlimit class, including simply dropping arriving packets for such a class. This CBQ implementation rate-limits overlimit classes to their allocated bandwidth. The rate-limiter computes the next time that an overlimit class is allowed to send traffic. The class will not be allowed to send another packet until this future time has arrived. This rate-limiter action is implemented as *rmc_delay_action* in the file rm_class.c of the CBQ implementation [FS97].

The estimator estimates the bandwidth used by each traffic class over the appropriate time interval (or, more precisely, simply estimates whether each class is over or under its allocated bandwidth). As discussed later, the time constant for the estimator determines the interval over which the router attempts to enforce the link-sharing bandwidth constraint. Hence the parameterization of this time constraint is key to enforcing link-sharing bandwidth allocations. This implementation employs a exponential weighted moving average (EWMA) to estimate the bandwidth used by each class. In this CBQ implementation, the estimator is implemented as the function *rmc_update_util* in the file rm_ class.c of the CBQ implementation [FS97].

The network management interface for this CBQ implementation allows for RSVP [BZ97] and other resource management mechanisms to configure the output link in the manner appropriate for those mechanisms. The network management interface allows for the creation and destruction of CBQ traffic classes, the appropriate filters to map the IP flows to traffic classes via the packet classifier, and a rather crude statistical interface for monitoring CBQ's internal state. All code in the management interface can be found in the file cbq.c of this CBQ implementation [FS97].

3 CBQ Parameters

The CBQ parameters for each class are set at class creation time. Using the experimental policy daemon cbqd, classes are created and parameterized as specified in a configuration file. Each class definition supplies the *priority*, the allocated *bandwidth* for the class (expressed in terms of link bandwidth percentage), *average packetsize*, *maxburst*, *minburst* and *maxdelay*. *Average packetsize* is used in calculating *maxidle* and *offtime*, as shown below. *Maxburst* is the maximum burst size for the class (that is, the maximum number of back-to-back packets sent by a previously-idle class). *Minburst* is the burst size for an overlimit class that is being regulated to its allocated bandwidth. *Maxdelay* is the target maximum delay (in milliseconds) that *average packetsize* packets will have to wait to be scheduled. *Maxdelay* is used to determine the *maxq* (maximum queue length in number of packets) parameter for the CBQ traffic class. Using these class parameters, other class parameters such as maxidle are derived to drive the CBQ scheduling apparatus.

For each class parameter not supplied in the class definition, default values are supplied. Some of these default values are as follows: *maxburst* defaults to 20 packets; *minburst* defaults to 2 packets; *average packetsize* defaults to 1000 bytes; and *maxdelay* defaults to 100 milliseconds.

In the CBQ policy daemon associated with the distributed code [FS97], the function *cbq_create_class* in the file cbqif.c utilizes the various inputs to compute the parameters discussed below.

3.1 CBQ Parameter Definitions

In CBQ, each class has variables *idle* and *avgidle*, and a parameter *maxidle* used in computing the limit status for the class. This section discusses setting the *maxidle* parameter. At one time a *minidle* parameter was used in the ns simulator, but that parameter has been removed, and there is now no lower bound on *avgidle*.

Definition: idle. The variable *idle* is the difference between the desired time and the measured actual time between the most recent packet transmissions for the last two packets sent from this class. When the connection is sending perfectly at its allotted rate p, then *idle* is zero. When the connection is sending more that its allocated bandwidth, then *idle* is negative.

Definition: avgidle. The variable *avgidle* is the average of *idle*, and is computed using a exponential weighted moving average (EWMA). When *avgidle* is zero or lower, then the class is overlimit (the class has been exceeding its allocated bandwidth in a recent short time interval).

Definition: maxidle. The parameter *maxidle* gives an upper bound for *avgidle*. Thus *maxidle* limits the `credit' given to a class that has recently been under its allocation.

Definition: offtime. The parameter *offtime* gives the time interval that a overlimit class must wait before sending another packet. This parameter is determined in part by the steady-state burst size *minburst* for a class when the class is running over its limit. In the ns simulator [MF95], this steady-state burst size is controlled by the *extradelay* parameter. A steady-state burst size of one packet can be achieved in the ns simulator by setting setting *extradelay* to 0. In the CBQ implementation a small steady-state burst size is achieved by setting *minburst* to 1.

3.2 Setting Maxidle

Maxidle controls the burstiness allowed to a class. As Appendix A shows, to permit a maximum burst of $maxburst$ back-to-back packets, *maxidle* is set as follows:

$$maxidle \leftarrow t(1/p-1)\frac{1-g^{maxburst}}{g^{maxburst}},$$

for t the interpacket time for `average' sized packets sent back-to-back, p the fraction of the link bandwidth allocated to the class, and weight g, for

$$\begin{aligned} g &= 1 - 2^{-RM\_FILTER\_GAIN} \\ &= 1 - 1/RM\_POWER. \end{aligned}$$

In addition, the following constraint should be observed:

$$maxidle \geq t(1-g).$$

Appendix A.3 shows that the calculation of *avgidle* in the code in fact corresponds to the equations in [FJ95]. Appendix B justifies the equations used for setting the variable *undertime* in the procedure *rmc_update_class_util*.

3.3 Setting Offtime

For leaf classes, *offtime* controls the steady-state burst size for a regulated class. In cbqd, for a regulated class with a burst size of 1, *offtime* in its unscaled value is set as follows:

$$offtime \leftarrow t(1/p-1).$$

This is the target waiting time to maintain the allocated bandwidth with a steady-state burst size of only one packet.

For a steady-state burst size of $minburst + 1$ packets for $minburst \geq 1$, cbqd further modifies *offtime* as follows:

$$offtime \leftarrow offtime * \left(1 + \frac{1}{(1-g)}\frac{(1-g^{minburst})}{g^{minburst}}\right).$$

3.4 Setting Efficient Mode

Additionally, this CBQ implementation implements a work conserving mode called *efficient mode*. When activated via the configuration file, *efficient mode* enables the CBQ implemenation to select a packet from a overlimit class if all classes of the link sharing hierarchy are overlimit. In practice, this will be the first overlimit class discovered in the link-sharing hierarchy. In employing *efficient mode*, it is important to note that the link will always be transmitting a packet. Hence, the link will achieve 100link. Equally, some traffic classes will experience more raw throughput than allocated in the link-sharing structure.

4 Description of experimental testbed

In the development and the testing of this CBQ implementation, one testbed was used to test and refine the CBQ implementation and collect the results from various experiments. The testbed as seen in Figure 1.

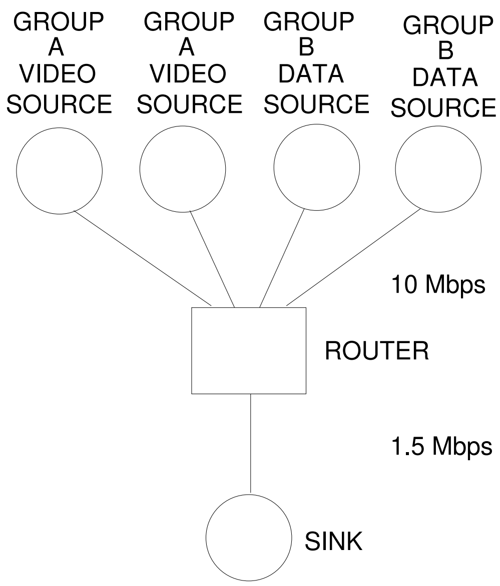

Figure 1: Network Setup for Link Sharing Experiments

This testbed employs 6 Sun SPARCstation 20 and Sun SPARCstation 5 workstations. The router in Figure 1 is a Sun SPARCstation 20 with two 125 MHz HyperSPARC CPUs with 256 KBytes of external cache. The router running Solaris 2.5.1 has been updated with TCP/IP kernel modules that will accommodate RSVP operation, routing functionality, IGMPv2, and DVMRP multicast routing. Within this testbed, *cbqd* was employed to configure the link between the router and the sink to test the CBQ implementation in various link sharing experiments. Each of the sources in the testbed where connected to the router via switched ethernet on individual networks.

For testing RSVP operation with CBQ, the testbed in

Figure 1 was upgraded. The link between the router and the sink was upgraded to 10 MBit/second switched ethernet. All the links between the router and the sources remained the same.

In performing CBQ and/or RSVP experiments, a number of parameters were captured and examined to gain insight on the performance of the CBQ machinery. These parameters included *packet delay*, *throughput*, *packet drops*, and *avgidle* within a class. To capture these parameters, a number of tools where employed including *tcpdump* and *adb*.

5 Description of the simulator

This note compares simulation acceptance tests for CBQ (class-based queueing) as implemented in the ns simulator [MF95] with the performance of CBQ in an actual implementation.

The simulator implements three separate algorithms for link-sharing described in [FJ95]: Formal, Top-level, and Ancestor-Only link-sharing. The simulator implements both WRR and PRR scheduling within classes of the same priority level. The WRR scheduling algorithm is described in Appendix A of [FJ95].

[Flo97] discusses the validations tests for the CBQ simulation in the ns simulator. Several of the validation tests were reproduced in the experimental testbed to validate the experimental code.

6 Experiments

In this sections we compare the experimental results of CBQ with results from simulations. The test scenarios are from the test suite used to validate the CBQ implementation in the NS simulator, and illustrate bandwidth allocations and hierarchical link-sharing. Additional test scenarios demonstrate the differences between weighted round-robin (WRR) and packet round-robin (PRR) scheduling within a priority level. The final test scenarios show the effects of the parameter that controls the maximum burstiness from a previously-idle class.

6.1 Hierarchical link-sharing

The simulation and experiment in Figure 3 verify that hierarchical link-sharing works correctly. The simulation scenario is given in Figure 1, and the link-sharing structure for the congested link is shown in Figure 2. The link bandwidth is shared by two agencies, each with two subclasses. When one of the subclasses has no data to send, that bandwidth should be available to be used by the other subclass of that agency.

The simulations on the left in Figure 3 generally reproduce the simulations shown in Figure 11 of [FJ95]. These simulations are run in the ns-1 simulator with the command

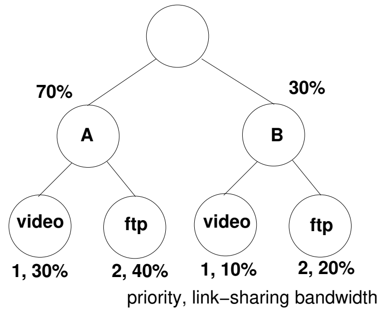

Figure 2: Link-sharing structure for two-agency link-sharing.

“ns test-suite-cbq.tcl cbqFor”, and in the ns-2 simulator with the command “ns test-suite-cbq-v1.tcl cbqFor” in the directory tcl/test. These simulations are discussed further in Section V.A. of [FJ95]. The reader is referred to the file “test-suite-cbq-v1.tcl” for the details of the simulation set-up.

The simulations use Formal link-sharing, with WRR scheduling within a priority class. The x-axis of the graph shows time in seconds, and the y-axis shows each class's bandwidth averaged over one-second intervals.

Initially, all classes have full demand, and receive their respective allocations of 10%, 20%, 30%, and 40% of the link bandwidth. Each class in turn has a period of no demand, and the simulation shows that in each case, the other subclass in that agency gets to use the bandwidth from that class. At the end of the simulation the two high-priority classes both have no demand, and the bandwidth is shared among the two lower-priority classes of the two agencies.

The right graph in Figure 3 shows an experiment with the same scenario. The CBQ implementation for the experiment uses a variant of Top-Level link-sharing, also with WRR scheduling within a priority class. The experiment shows in steady state, when all classes have full demand, that each class receives roughly its allocated bandwidth (10%, 20%, 30%, and 40%). The experiment also shows that while hierarchical link-sharing is working roughly as it should, “borrowed” bandwidth is allocated somewhat less precisely. When one of the classes stops sending, its “companion” class in the same agency generally does not get to borrow enough to receive all of the bandwidth allocated to the parent class.

6.2 Classes with small bandwidth allocations

The simulations and experiments in this section test CBQ's ability to treat priority and bandwidth as orthogonal. More precisely, the test scenario shows CBQ's ability to allocate and deliver a small bandwidth to an overlimit high-priority class.

Figure 4 shows a simulation and experiment of a link that is shared by two classes, a high-priority class A allocated 0.1% of the link bandwidth, and a lower-priority class B allocated the remaining 99.9% of the bandwidth. This sce-

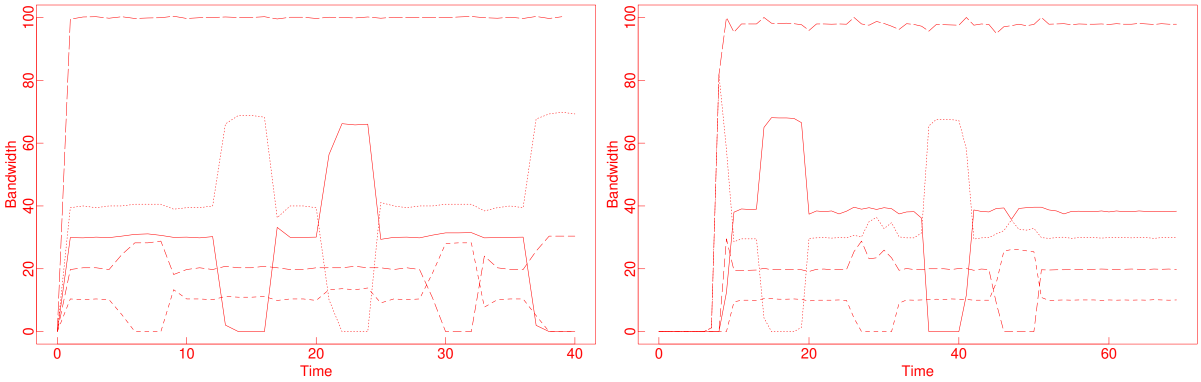

Figure 3: Simulations (left) and experiments (right, 11/01/98) of hierarchical link-sharing.

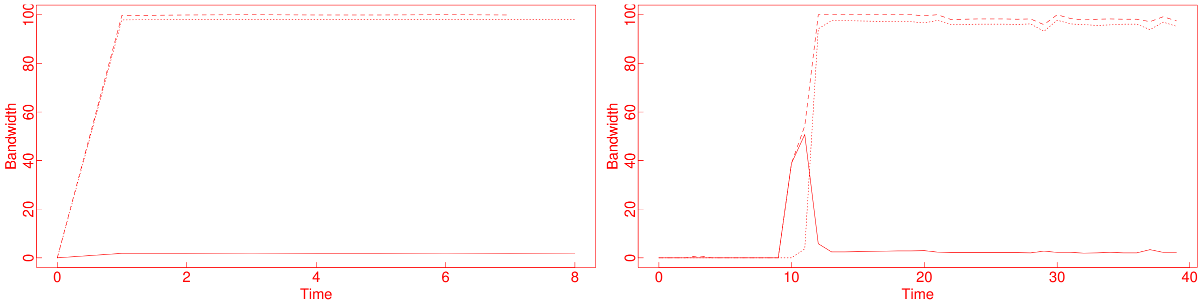

Figure 4: Simulations (left) and experiments (right, 11/01/98) of classes with small bandwidth allocations.

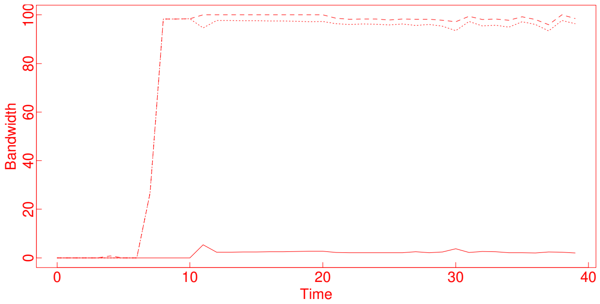

Figure 5: PRR, 11/1/98

nario tests the robustness of the link-sharing mechanisms and the sensitivity to the CBQ parameters when there is a high-priority class with a small allocated bandwidth.

This scenario serves in part as a stress-test of the link-sharing algorithms in an implementation. In the simulator, this scenario demonstrates the limitations of Ancestor-Only link-sharing (as compared to Formal or Top-Level link-sharing). Of the three link-sharing mechanisms in the NS simulator, Ancestor-Only link-sharing is closest to the link-sharing mechanisms in the 1995 CBQ code.

The data source for Class A is a CBR flow that sends 190-byte packets every 0.001 seconds. The data source for Class B is a CBR flow that sends 500-byte packets every 0.002 seconds.

The left graph of Figure 4 shows simulations using Formal link-sharing. The higher-priority class is properly restricted to a small fraction of the link bandwidth. This test is run in ns-1 with the command "ns test-suite-cbq.tcl cbqTwoF", and in ns-2 with the command "ns test-suite-cbq-v1.tcl cbqTwoF" in the directory tcl/test.

The bottom graph of Figure 4 shows an experiment of the same scenario using a variant of Top-Level link-sharing. As Figure 4 shows, in the experiment the bandwidth of the high-priority class is controlled somewhat less precisely that in the simulation. We have not yet determined what changes are needed in the CBQ code to bring the results more in line with those in the simulator.

6.3 Experiments on WRR and PRR

The simulations and experiments in this section demonstrate one of the differences between the PRR and WRR scheduling algorithms within a priority level. These tests show that with WRR scheduling, extra bandwidth is allocated according to the relative bandwidth allocations of the other high-priority classes. With PRR scheduling, the allocation of extra bandwidth is according to the relative packet sizes of the other high-priority classes, and not according to their relative bandwidth allocations.

Another motivation for using WRR instead of PRR scheduleing within a priority level can be that WRR scheduling reduces the worst-case delay that can be experienced by a set of packets arriving to an empty queue. This aspect of WRR scheduling is not demonstrated in this paper.

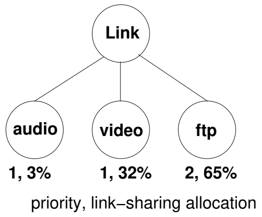

Figure 6: Link-sharing structure for two-agency link-sharing.

Figure 7 reproduces Figure 10 from [FJ95]. The link-sharing structure for the congested link is shown in Figure 6. The top left graph shows a simulation using PRR, and the bottom left graph shows a simulation using WRR. The two simulations differ in the distribution of "extra" bandwidth to the two high-priority classes when the lower priority ftp class has no data to send. When the ftp class has no data to send, WRR distributes the extra bandwidth to the audio and video classes in proportion to their allocated bandwidth. In contrast, PRR distributes the extra bandwidth to the audio and video classes in proportion to their packet sizes, which are 190 bytes and 500 bytes respectively.

These simulations can be run in ns-1 with the commands "ns test-suite-cbq.tcl cbqPRR" and "ns test-suite-cbq.tcl cbqWRR", and in ns-2 with the commands "ns test-suite-cbq.tcl PRR" and "ns test-suite-cbq.tcl WRR" in directory tcl/test.

The top left graph shows the experiment using PRR, and the bottom left graph shows the experiment using WRR. Both experiments show the expected distribution of bandwidth between the higher-priority classes when the low-priority class stops sending at time 22-30. The main discrepancy between the experiments and the simulations is that when the higher-priority video class stops sending at time 34-42, a small amount of the "extra" bandwidth goes to the lower-priority ftp class. The initial spikes in the experiment are due to the staggered starting times of the three classes, and each experiment also shows a "drop-out" where no data was collected.

6.4 Experiments showing effects of changing parameters

6.4.1 The maximum burstiness for underlimit classes

Figure 8 shows the use of the maxidle parameter to control the maximum burstiness allowed from a previously-idle class. Consider a high-priority class that has been using less than its allocated bandwidth. If packets arrive for that class in a burst, then the worst-case delay for packets in that class is reduced if the packets are allowed to leave in a burst. At the same time, limitations have to be put on the short-term bursts allowed from high-priority classes, in order to limit the delays experienced by lower-priority classes. This maximum burstiness allowed to underlimit classes is controlled by the maxidle parameter.

These figures show simulations with different values for maxidle (which determines the maximum number of back-to-back packets). The "maxidle" parameter serves a similar function as does the bucket size in a token bucket.

For each graph in Figure 8, the bottom row shows packets for Class A, and the top row shows packets for Class B. Each of the two classes is allocated 30% of the link bandwidth. The x-axis shows time and the y-axis shows the packet number (mod 90).

For the simulations, there is a mark for each packet when it arrives at the congested router, and another mark when the packet leaves the congested router. Class B has a CBR source that sends 1000-byte packets at 0.01 second intervals. At time 1, eighty 1000-byte packets arrive at the router for Class A back-to-back. The graphs show that a burst of packets is allowed to be sent at line rate (where the size of the burst is either 25 packets or 5 packets, depending on the value of maxidle). After the initial burst, the remaining packets in the queue for Class A are sent at the allocated bandwidth of the class. These simulations are run in ns-1 with the commands "ns test-suite-cbq.tcl Max1" and "ns test-suite-cbq.tcl Max2", and in ns-2 with the commands "ns test-suite-cbq.tcl MAX1" and "ns test-suite-cbq.tcl MAX2" in directory tcl/test.

For the experiments, there is a mark for each packet as it arrives at the receiver. At a fixed time, a flow assigned to Class A begins to transmit packets at a rate higher than the allocated bandwidth of the class. The experiments show that maxidle successfully controls the size of the initial burst.

7 Acknowledgements

References

[BZ97] R. Braden and L. Zhang. Resource reservation protocol version 1 functional specification. (Internet draft, work in progress), June 1997. URL ftp://ds.intenic.net/internet-drafts/draft-ietf-rsvp-spec-16.txt.

[FJ95] S. Floyd and V. Jacobson. Link-sharing and resource management models for packet networks. *IEEE/ACM Transactions on Networking*, 3(4), 1995. URL http://www-nrg.ee.lbl.gov/nrg-papers.html/.

[Flo97] S. Floyd. Ns simulator tests for class-based queueing. *Unpublished draft*, Apr. 1997. URL ftp://ftp.ee.lbl.gov/papers/cbqsims.ps.Z.

[FS97] S. Floyd and M. Speer. Lbnl's cbq code v2.0, May 1997. URL ftp://ftp.ee.lbl.gov/cbq2.0.tar.Z.

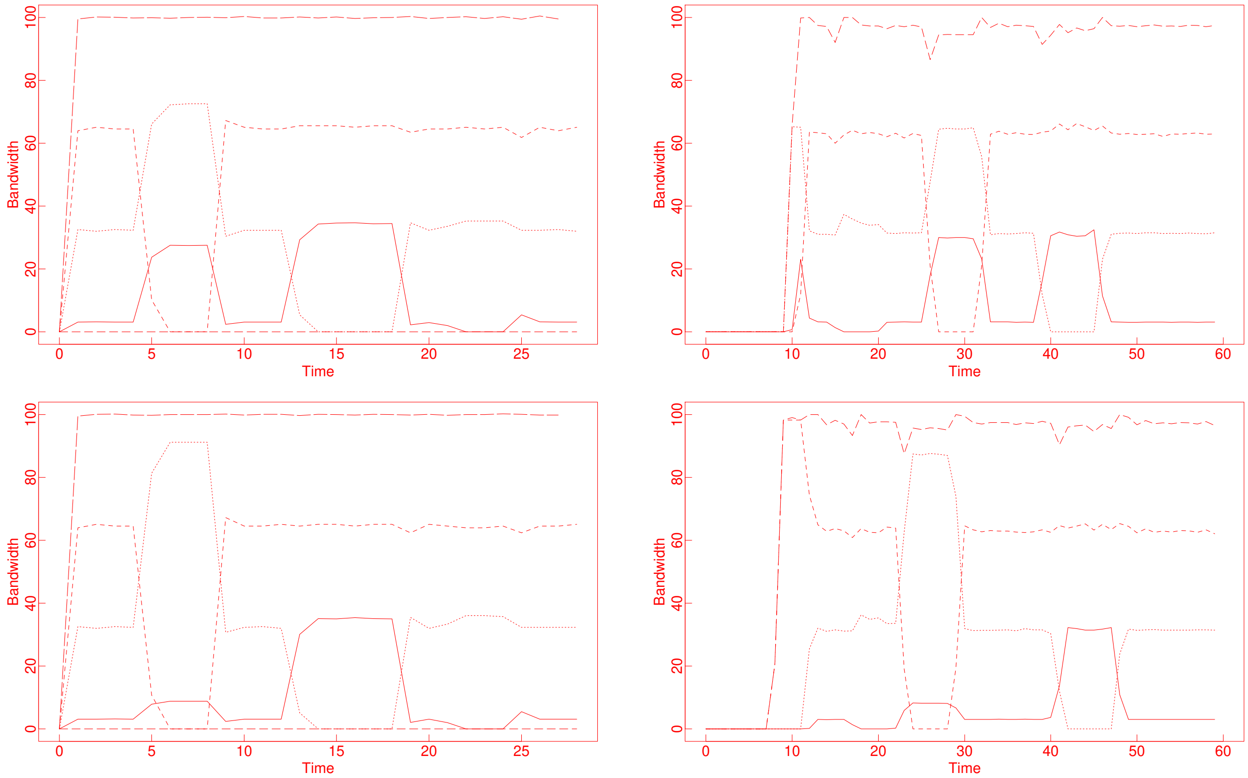

Figure 7: Simulations (left column) and experiments (right column) with PRR (top row) and WRR (bottom row).

[Jac95] V. Jacobson. Lbnl's cbq code v1.1, August 1995. URL ftp://ftp.ee.lbl.gov/cbq.tar.Z.

[MF95] S. McCanne and S. Floyd. Ns (network simulator), 1995. URL http://www-mash.cs.berkeley.edu/ns/.

[WGC+95] I. Wakeman, A. Ghosh, J. Crowcroft, V. Jacobson, and S. Flo yd. Implementing real time packet forwarding policies using streams. *Usenix 1995 Technical Conference, January 1995, New Orleans, Louisiana, pp. 71-82.*, January 1995. URL ftp://cs.ucl.ac.uk/darpa/usenix-cbq.ps.Z.

A Maxidle

We assume that *maxidle* is set when a class is created. *Maxidle* determines the maximum size burst allowed for a class that has sent no packets in the recent time interval.

Definitions: t, g, p. Let t be the time to transmit a packet, for the most recent packet sent from a class. Let p be the fraction of the link bandwidth allocated to that class. If the actual interpacket time for packets sent back-to-back from the class is t, then the `target' interpacket time (the time between transmitting two packets of that size at the allocated rate) is t/p, and *idle* is $t\,(1 - 1/p)$. The formula for computing *avgidle* is

$$\mathit{avgidle} \leftarrow g\;\mathit{avgidle} + (1-g)\,\mathit{idle}.$$

The weight g would typically be $15/16$ (that is, $g = 1 - 1/RM\_POWER$, for RM_POWER set to 2^4) or $31/32$ (for RM_POWER set to 5). The weight g determines the "time constant" of the averager.

Assume that *avgidle* initially has the value $maxidle$. Then after n back-to-back packets, *avgidle* is

$$g^n maxidle + \sum_{i=0}^{n-1} g^i\,(1-g)\;\mathit{idle}$$
$$= g^n\,maxidle + \mathit{idle}\,(1-g^n).$$

This derivation uses the fact that

$$\sum_{i=0}^{m} g^i = \frac{1-g^{m+1}}{1-g}.$$

If $avgidle$ reaches 0 after n consecutive packets, and $avgidle$ had the value $maxidle$ at the beginning of the burst, then the maximum size burst allowed for that class is n packets. In order to allow a maximum size burst of n packets of S bytes each, $maxidle$ should be set to

$$maxidle = t_S(1/p - 1)\frac{(1-g^n)}{g^n}, \qquad (1)$$

where t_S is the packet transmission time for a packet of S bytes.

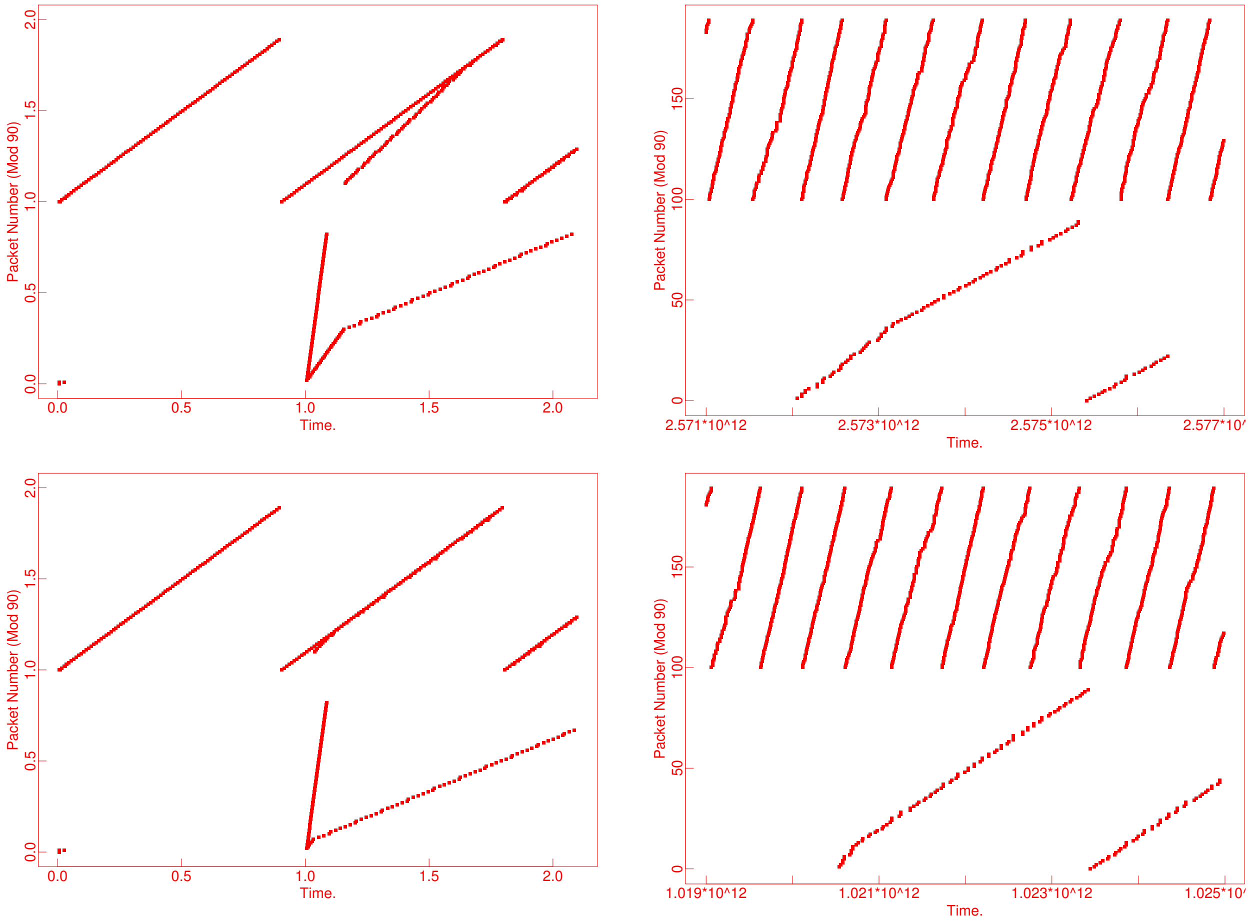

Figure 8: Simulations (left) and experiments (right, 11/19/97) with maxidle set for a max burst of 25 (top) or 5 (bottom) back-to-back packets.

A.1 An additional constraint on maxidle

In addition to equation (1) above, *maxidle* needs to be sufficiently large to allow for the normal variation in *idle*, and therefore *avgidle*, over one round of the round-robin scheduling. This is not a problem for classes with moderate bandwidth allocations, but an additional constraint is required for classes with bandwidth allocations greater than half the link bandwidth.

Let t be the packet transmission time for a "typical" packet. When a class sends two packets back-to-back, the resulting value of *idle* is $t - t/p$, as shown earlier. However, any class allocated less than 100% of the link bandwidth will occasionally have to wait for at least one other packet to be transmitted, and in this case the resulting value of *idle* will be at least $2t - t/p$ (making the simplifying assumption for the moment that all packets are the same size). Thus *maxidle* has to be sufficiently large not to "lose" the information that a class waited for another packet to be transmitted.

Consider a class A that has just become overlimit (e.g., *avgidle* has just become negative), and has to wait for a packet from another class to be transmitted. After that transmission, class A's value for *idle* is $2t - t/p$, and this is averaged into the previous value of $avgidle = \epsilon$ as follows:

$$avgidle \leftarrow g(-\epsilon) + (1-g)idle.$$

This gives

$$avgidle \leftarrow t\left(2 - \frac{1}{p}\right)(1-g) - g\epsilon$$
$$< t(1-g).$$

Thus, to ensure that a high-bandwidth class does not "lose" information about having waiting for some other packet to be transmitted, it is sufficient that the following condition on *maxidle* be observed:

$$maxidle \geq t(1-g).$$

A.2 Maxidle with arbitrary packet sizes

Of course, packets can come in a wide range of sizes. Assume that the actual packets are $a\,S$ bytes, for some $a > 0$, with transmission times of $a\,t_S$. Then what is the maximum number of back-to-back packets of this size that could be sent, if *avgidle* is initially at the value for *maxidle* given by the equation above?

Idle will be $a\,t_S\,(1 - 1/p)$, and after $b\,n$ back-to-back packets,

$$avgidle = g^{b\,n}\,maxidle + idle\,(1 - g^{b\,n})$$
$$= g^{b\,n-n} t_S(1/p - 1)(1 - g^n)$$
$$+ a\,t_S\,(1 - 1/p)\,(1 - g^{b\,n}).$$

We would like to know the value of b when *avgidle* first becomes zero.

Solving we get

$$g^{n(b-1)} t_S(1/p-1)(1-g^n) - a\,t_S\,(1/p-1)\,(1-g^{b\,n}) = 0,$$
$$g^{n(b-1)}(1-g^n) - a\,(1-g^{b\,n}) = 0,$$
$$g^{n(b-1)}(1-g^n) + a\,g^{n(b-1)}\,g^n = a,$$
$$g^{n(b-1)}(1+(a-1)\,g^n) = a,$$
$$g^{n(b-1)} = \frac{a}{1+(a-1)\,g^n},$$

and

$$b = \frac{\log \frac{a}{1+(a-1)\,g^n}}{n \log g} + 1.$$

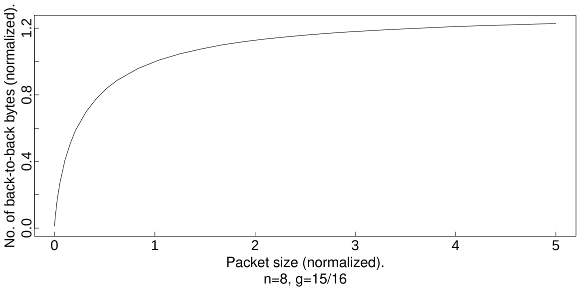

Figure 9: Number of back-to-back bytes allowed by *maxidle* given a range of packet sizes. Fraction of allocated throughput, for $g = 15/16$.

The initial t_S transmission time was based on S-byte packets. Now, instead of sending $n\,S$ back-to-back bytes, with packets of $a\,S$ bytes we get to send $b\,n\,a\,S$ back-to-back bytes. Figure 9 shows $a\,b$ plotted as a function of a, for $n = 8$ and $g = 15/16$. As Figure 9 shows, *maxidle* is fairly effective in controlling the maximum number of back-to-back *bytes* even for a range of packet sizes.

A.3 The calculation of avgidle

This section shows that the calculation of *avgidle* in the code in fact corresponds to the equation in [FJ95]. From [FJ95], *avgidle* is calculated from *idle* as follows:

$$avg \leftarrow (1-w)\,avg + w * diff,$$

for some weight w chosen as a negative power of two.

In the code in rm_class.c, there are two cases. When the option `USE_HRTIME' is employed, meaning that a 64-bit representation of wall-clock time is used, `avgidle` is represented in its true unscaled value, and the following equation is used:

```
avgidle +=                              (1)
((idle - avgidle) >> RM_FILTER_GAIN);
```

This is equivalent to the following:

```
avgidle +=
  (idle - avgidle) (1/2^RM_FILTER_GAIN);
```

or equivalently,

```
avgidle =
  (1 - 1/2^RM_FILTER_GAIN) avgidle
  + (1/2^RM_FILTER_GAIN) idle;
```

This gives the correct equation.

When `USE_HRTIME' is not employed, the scaled value `avgIdle` is used as follows:

```
avgIdle = avgidle * 2^RM_FILTER_GAIN,
```

for `avgidle` the true unscaled value. In this case, the following equation is used instead of equation (1):

```
avgIdle +=                           (2)
  idle - ( avgIdle >> RM_FILTER_GAIN),
```

This is therefore equivalent to

```
avgidle * 2^RM_FILTER_GAIN +=
  idle - avgidle.
```

Simplifying,

```
avgidle +=
  ( idle - avgidle )/2^RM_FILTER_GAIN.
```

Thus, equation (1) when `USE_HRTIME' is employed is equivalent to equation (2) and a scaled value for `avgIdle` when `USE_HRTIME' is not employed.

B Regulating overlimit classes: the details

Definition: undertime, now, overlimit. The CBQ scheduler checks the class variable *undertime* to see if a class can send packets without borrowing. A class is not allowed to send a packet when $undertime > now$ and the class is unable to borrow. If $avgidle$ is positive after a packet has been sent from a class, then $undertime$ should be set to zero (or to something else less than the current time now).

After a packet is sent from a class, $avgidle$ is updated. This section explains the equations used when $avgidle$ is negative and a class that is unable to borrow therefore has to be regulated to its allocated bandwidth. If the class is to be regulated, then it must wait at least the target waiting time *ptime* before sending another packet, for

$$ptime \leftarrow t(1/p - 1).$$

If $avgidle$ is negative, then the class must also wait at least

$$(1 - RM\_POWER) * avgidle$$

additional seconds, to ensure that avgidle will no longer be negative when the next packet is sent.[1]

To show that this is correct, the class will wait

$$t(1/p - 1) + (1 - RM\_POWER) * avgidle$$

seconds before sending the next packet. Assume that this packet is the same size as the last one, and also has a transmission time of t seconds. Then $idle$ will be calculated as

$$(t(1/p-1) + (1 - RM\_POWER) * avgidle) - t(1/p-1)$$
$$= (1 - RM\_POWER) * avgidle,$$

and the next value for $avgidle$ will be as follows:

$$avgidle \leftarrow g\, avgidle + (1-g)\, idle$$
$$= (1 - 1/RM\_POWER)\, avgidle$$
$$+ (1/RM\_POWER)\,(1 - RM\_POWER) * avgidle$$
$$= 0.$$

There is an optional parameter called *extradelay_* in the ns simulator that can be used in determining how long to additionally delay an overlimit class. The parameter *extradelay_* gives the additional time interval that a overlimit class must wait before sending another packet. This parameter determines the steady-state burst size for a class when the class is running over its limit. When *extradelay_* is set to 0, then the steady-state burst size for an overlimit class is one packet. We do not discuss this further in this paper.

For the experimental code, the parameter *offtime* is used to determine how long an overlimit class is to be delayed. For a steady-state burst size of one packet, *offtime* is set to *ptime*, for *ptime* as defined in the beginning of this section. For a steady-state burst size of b packets, for $b > 1$, then *offtime* is set as follows:

$$offtime \leftarrow ptime * \left(1 + \frac{1}{(1-g)} \frac{(1-g^{b-1})}{g^{b-1}}\right),$$

for

$$g = 1 - 1/RM\_POWER$$
$$= 1 - 1/2^{RM\_FILTER\_GAIN}.$$

[1] In the simulator ns, this is called $POWEROFTWO$ instead of RM_POWER. Recall that RM_POWER is defined by the following equation, for g the weight used in the exponential weighted moving average: $g = 1 - 1/RM\_POWER$.

B.1 Controlling the minimum burstiness for a regulated class

The guidelines above for calculating undertime assume that after a regulated class sends a packet, it will have to wait the minimum possible time before sending the next packet. However, for efficiency of implementation, it might in some environments be desirable to have the regulated class wait longer after sending a packet, and to therefore send small bursts of packets, giving a steady-state burst size for the regulated class of more than one packet.

Let *offtime* be the time that the class has to wait after sending a packet. Assume that in steady state, for a class with plenty of demand that is being restricted to its link-sharing bandwidth), the CBQ implementation regulates the output for that class to a steady-state burst of n packets. (This refers to a steady-state where the class is allowed to send a burst of n packets, and then is forced to wait some time before sending another burst of n packets, and so on.) Let $avgidle_n$ be the value for $avgidle$ that allows a burst of size n before $avgidle$ reaches 0. Then

$$avgidle_n = t(1/p - 1)\frac{(1-g^n)}{g^n}.$$

Assume that a class is made to wait when *avgidle* becomes at most zero. Then we want to set *offtime* so that, if the class is allowed to send a packet after *offtime* seconds, the new value for *avgidle* will be $avgidle_{n-1}$, so that exactly $n-1$ more consecutive packets can be sent until *avgidle* reaches zero again. This is true if

$$(1-g)\, idle = avgidle_{n-1},$$

for *idle* as follows:

$$idle = offtime + t - t/p.$$

This gives

$$offtime = \frac{1}{1-g} avgidle_{n-1} + t\,(1/p - 1)$$
$$= \frac{1}{1-g}\, t\,(1/p-1)\frac{(1-g^{n-1})}{g^{n-1}} + t\,(1/p-1).$$

As examples, for $g = 15/16$, $t = 0.01$, and $n = 8$, this is

$$offtime = 0.1014(1/p - 1).$$

For $g = 31/32$,

$$offtime = 0.0896(1/p - 1).$$

(This concurs with the findings later in this section that for a class with a steady-state burst size of n, the throughput is higher with higher values of g. As g increases, then *offtime* approaches closer to $nt(1/p-1)$, the value that would be needed for the class to achieve 100% of its throughput allocation.)

What is a class's actual throughput if this procedure is used, and a regulated class is required to send its packets in small bursts? The class transmits n packets in $n\,t$ seconds, and then waits for *offtime* seconds. Thus the actual throughput, as a fraction of the maximum bandwidth of the link, is

$$\frac{n\,t}{n\,t + \mathit{offtime}}$$

$$= \frac{n}{n + \frac{1}{1-g}(1/p-1)\frac{(1-g^{n-1})}{g^{n-1}} + 1/p - 1}.$$

A connection that sends bursts of n packets in this manner will get slightly less that the specified fraction p of the bandwidth, for $n > 1$. Figure 10 shows the fraction f of its allocated throughput achieved by a delayed class, for $g = 15/16$. The x-axis shows the steady-state burst size n and the y-axis shows the allocated throughput for the class. The z-axis shows the fraction of allocated throughput achieved by the delayed class. For this figure, we assume that $t = 0.01$ seconds, but the results are essentially the same for a wide range of values for t (e.g., for t as small as 0.01 ms). The results for $g = 31/32$ are similar to those in Figure 10. This data argues for a small steady-state burst size, particularly for classes with small allocations. In our simulations, we generally use a steady-state burst size of $n = 1$ packet.

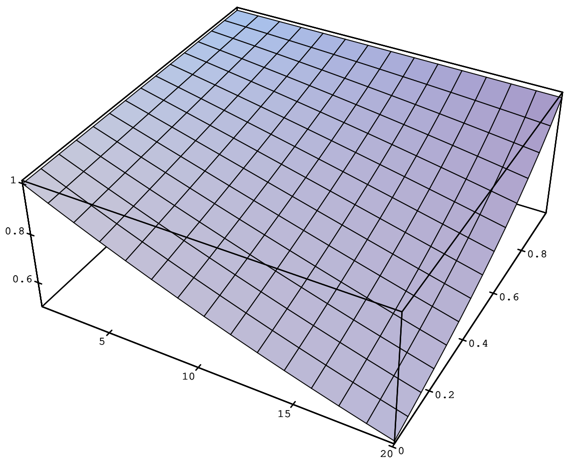

Figure 10: Fraction of allocated throughput, for $g = 15/16$.

(What is the intuition behind this behavior? With a steady-state consisting of a burst of n packets followed by a delay, the computed *avgidle* oscillates above and below the true steady-state average of the variable *idle*. With this mechanism, the class is delayed when the true average for idle is greater than zero, and therefore the true throughput is less than the allocated throughput.

For any $f \leq 1$, in order to guarantee that a class achieves at least the fraction f of its allocated throughput, it is sufficient to pick a steady-state burst size of at most n, for n such that

$$\frac{n}{n + \frac{1}{1-g}(1/p-1)\frac{(1-g^{n-1})}{g^{n-1}} + (1/p-1)}\,\frac{1}{p} = f.$$

Figure 11 shows the upper bound on burst size for a class to achieve at least 90% of its allocated throughput, for $g = 31/32$. Thus, for a steady-state burst size of 8 packets, a class should achieve at least 90% of its allocated throughput, regardless of the allocated bandwidth for that class.

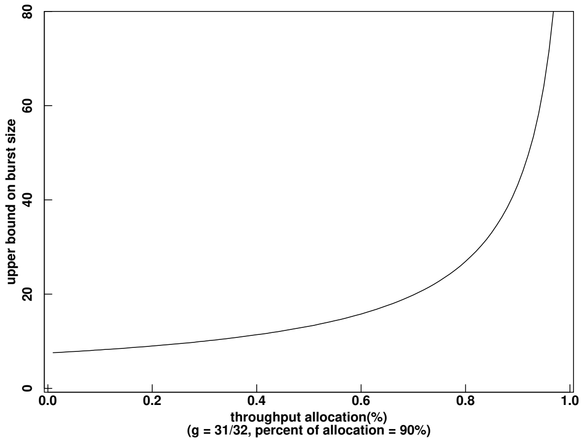

Figure 11: Upper bound on burst size for 90% of throughput, for $g = 31/32$.

B.2 Offtime with arbitrary packet sizes

Assume that *offtime* is pre-computed based on an assumption of a typical packet size of s bytes, with a transmission time of t_1 seconds, but that actual packets have a size of as bytes, as in Section A.2. Using the results in Section A.2, we can infer that *offtime* should be fairly effective in maintaining the steady-state burst size in bytes, even with a range of typical packet sizes. Simulations confirm that the throughput achieved by a class in bytes-per-second is fairly insensitive to the packet size in bytes.

C Guaranteed Service

This is a note on implementing guaranteed service using CBQ. We assume that a flow that has been accepted (by the admissions control procedure) for guaranteed service will be assigned its own priority-one class at the router. This note does not discuss the admissions control procedure.

C.1 The guaranteed service template

From the guaranteed service template, the router is given the TSpec (traffic specification) and RSpec (service specification) for the flow. The TSpec consists of a token bucket with a bucket depth b and a bucket rate r; the source should be policed at the edge of the network by this token bucket. The RSpec consists of a rate R, for $R \geq r$. The flow's service at the router is characterized by the bandwidth R and a buffer size B, where R represents the bandwidth that the flow is entitled to receive, and B represents the allocated buffer space in the router. It is assumed that the router will derive, from the TSpec, the required buffer space B to that no packets are dropped at the router.

The router exports two error terms C and D for a particular accepted flow. The router ensures that for a flow policed by the specified token bucket, the per-packet queueing delay at the router will be less than

$$b/R + C/R + D.$$

The service template gives the example of WRQ, where C is the MTU of the outbound link and D is 0. The queueing delay of b/R can be attributed to a bucket b of packets arriving instantaneously and effectively being transmitted at rate R. The C/R and D terms should include the queueing delay due to the transmission delay for the packet currently in service when a real-time packet arrives at the gateway, as well as the queueing delay due to other real-time packets scheduled to be sent before the packet in question.

C.2 CBQ

In CBQ, the priority-1 classes are served round-robin. The service discipline for the priority-1 classes can be either packet-by-packet round-robin (PRR) or weighted round-robin (WRR). For each class's turn in the round-robin scheduling, the amount of data that that packet gets to send depends on two factors, the scheduling algorithm (PRR or WRR), and whether or not that class is classified as overlimit. A class that is classified as overlimit might be additionally limited in the amount of data that it can send, being essentially rate-limited to its link-sharing bandwidth.

In CBQ, the worst-case delay for a packet from a guaranteed-service connection i occurs when packets from connection i arrive at the gateway just after connection i's turn in the round-robin scheme has passed.

Consider the delay for connection i at a particular gateway. Assume that there are at most k priority-1 classes, with the ith class having a bandwidth allocation of ρ_i. Let the capacity of the output link be ρ bytes/sec. (Let $p_i = \rho_i/\rho$ be connection i's allocated fraction of the bandwidth, for $\rho_i \geq R_i$, the entitled bandwidth for connection i. We assume that at most half of the bandwidth is allocated to priority-1 classes: $\sum_{i=1}^{k} p_i \leq 1/2$. (This is not because we believe that this is a necessary condition for providing guaranteed service in our scheme; this is simply because this greatly simplifies the analysis.)

C.2.1 Packet-by-packet round robin

The main reason to examine PRR is to explain why WRR scheduling gives better performance for guaranteed service traffic than WRR scheduling. With PRR for servicing priority-1 classes, as in the current distribution of CBQ code, priority-1 classes are serviced in round-robin order, and each class that is not overlimit gets to send exactly one packet per round.

We assume that k is an upper bound on the possible number of priority-1 classes. Assume that all of the packets for the priority-1 classes contain at most $L = MTU$ bytes. Then a class-C_i packet that arrives at the gateway just after its "turn" has to wait at most $(k-1)L/\rho$ seconds for the other priority-1 classes to send a packet. (If a non-priority-one packet was in service when the class-C_i packet arrived, the class C_i also has to wait for the router to finish transmitting that packet. However, with CBQ no lower-priority packets will be sent when there is a non-overlimit higher-priority class with data to send.) If each packet for class C_i has at least s_i bytes, then the bandwidth received by class C_i in each round of the round-robin is at least

$$\frac{s_i}{(k-1)L + s_i}\rho.$$

If

$$\frac{s_i}{(k-1)L + s_i}\rho > R_i, \tag{2}$$

where R_i is class C_i's entitled bandwidth, then class C_i receives at least its entitled bandwidth in every round-robin round. In this case, class-C_i's delay due to the wait for its turn in the round-robin is at most $(k-1)L/\rho$ seconds.

Equation 2 is likely to hold for a link where the aggregate bandwidth allocated to priority-one classes is small relative to the link bandwidth. Equation 2 is also likely to hold for a priority-one class that does not send small data packets and does not have a disproportionate allocation of the bandwidth among the priority-one classes.

If equation 2 does not hold, the packets arriving for class C_i at rate r could accumulate in a queue for that class until some of the other priority-one classes are classified as overlimit, and rate-limited to their allocated bandwidth. In this case, in order to calculate the worst-case delay for class C_i packets, we would have to calculate the worst-case time until other priority-one classes that are receiving more than their share of the bandwidth can be classified as overlimit, and then calculate the worst-case time until the accumulated backlog for class i has been dispersed. This depends on such factors as the time constant used by the estimator in estimating the bandwidth used by each class. Rather than going through these rather tedious calculations, we point out below that for a version of WRR scheduling where each priority-one class is guaranteed to receive its allocated bandwidth in each round of the round-robin, it is much more straightforward to bound the worst-case delay of an arriving guaranteed-service packet.

C.3 Weighted round robin

In our implementation of WRR, the priority-1 classes are served in round robin order. Each priority-1 class that is not overlimit is allowed to send $p_i M$ bytes. The parameter M is required to be at least $2kL$, for L the maximum packet size in bytes, and k the maximum number of priority-one classes that will be active at one time. Each class is allowed to finish sending the packet that contains the $p_i M$-th byte. If a class exceeds its allotment by x bytes, then its allotment for the next round is decremented by that amount. We say that a class that has exceeded its allotment in this fashion has a *deficit*.

Note that with this implementation of WRR, for a priority-1 class with a sufficiently large allocation (that is, where $p_i M$ is larger than the biggest packet for that class), that class will never be forced to "wait out" a round of the round robin due to a large accumulated deficit. However, given a static value for M, if none of the priority-1 classes with "larger" allocations have data to send, and all of the priority-1 classes with "smaller" allocations have to "wait out" a round of the round robin due to a large accumulated deficit, then the scheduling algorithm could (conceptually) take several rounds of the round robin before reducing the deficits and finding some priority-one class that is able to send a packet.

We explore the worst-case delay for a class-C_i packet that arrives at the gateway with no class-C_i backlog and no deficit.

Observation 1: At most

$$\sum_{i=1}^{k} (p_i M) + kL \leq M/2 + kL$$

bytes can be transmitted in one round of the priority-one round-robin. (Each class can send at most $L-1$ bytes over its allotted number of bytes in one round. This also uses the assumption that at most half of the link bandwidth is allocated to priority-1 classes.) □

Observation 2: Each priority-one class is guaranteed to receive its allocated share of the bandwidth in each round-robin round, unless some of this allocation was used `in advance' in the most recently-sent packet from that connection.

Proof: Priority-1 class C_i gets to send at least $p_i M$ bytes in a round (unless some of those bytes were sent `in advance' in the most recent packet). Thus class C_i gets a fraction at least

$$\frac{p_i M}{M/2 + kL} \geq p_i$$

of the bandwidth in each round-robin round. (This calculation uses the assumption that $M \geq 2kL$.) □

Observation 3: For a priority-one class with no backlog at the gateway and with no deficit, an arriving packet from class C_i can wait at most

$$\frac{M/2 + kL}{\rho} \geq \frac{2kL}{\rho}$$

seconds before receiving service. (This includes the transmission delay for the packet in service when the class-C_i packet arrives at the gateway.) □

Thus, a CBQ node that implements WRR for providing guaranteed service could advertise the constants

$$C = 0$$

and

$$D = \frac{M/2 + kL}{\rho}.$$

C.4 Comparisons of Weighted Round Robin and Weighted Fair Queueing

Note that unlike weighted fair queueing, with our implementation of weighted round robin the worst case packet delay is the same for any priority-1 class, and is not a function of the allocated bandwidth for that class. In comparison, the worst-case packet delay in WFQ, L/R, is a function of the bandwidth R assigned to that class.

The worst-case delay for high-bandwidth guaranteed service classes could be reduced further, if desired, by giving those classes more than one turn in each round of the round-robin. The frame structure is the order in which classes are listed for one round-robin round. `Choosing a good frame structure' in Section 1.6.3 of Parekh's thesis gives an easy algorithm for creating a good frame structure. This kind of change would move WRR closer to a fluid-flow scheduling model.

An essentially equivalent way to explore the space between the WRR and WFQ scheduling algorithms would be to modify the version of WRR described in this paper to reduce M, the maximum number of bytes that can be sent in one round of the round robin without "borrowing" from rounds of the round-robin, and to place restrictions on the number of bytes that a class can borrow from future rounds. This kind of change would move WRR closer to a fluid-flow scheduling model: rounds would be smaller, a class with a large allocation would be guaranteed to receive its allocated bandwidth in every round of the round robin, and a class with a small allocation would be guaranteed to receive its allocated bandwidth in every k rounds of the round robin.