Providing Guaranteed Services Without Per Flow Management *

Ion Stoica, Hui Zhang
Carnegie Mellon University
Pittsburgh, PA 15213
e-mail: {istoica,hzhang}@cs.cmu.edu

Abstract

Existing approaches for providing guaranteed services require routers to manage per flow states and perform per flow operations [12, 3]. Such a *stateful* network architecture is less scalable and robust than *stateless* network architectures like the original IP and the recently proposed Diffserv [11]. However, services provided with current *stateless* solutions, Diffserv included, have lower flexibility, utilization, and/or assurance level as compared to the services that can be provided with per flow mechanisms.

In this paper, we propose techniques that do not require per flow management (either control or data planes) at core routers, but can implement guaranteed services with levels of flexibility, utilization, and assurance similar to those that can be provided with per flow mechanisms. In this way we can simultaneously achieve high quality of service, high scalability and robustness. The key technique we use is called Dynamic Packet State (DPS), which provides a lightweight and robust mechanism for routers to coordinate actions and implement distributed algorithms. We present an implementation of the proposed algorithms that has minimum incompatibility with IPv4.

1 Introduction

Current IP networks provide one simple service: the best-effort datagram delivery. Such a simple service model allows IP routers to be stateless: except routing state, which is highly aggregated, routers do not keep any other fine grain information about traffic. Providing a minimalist service model and having the "stateless waist" in the protocol hourglass allows the Internet to scale with both the size of the network and heterogeneous applications and technologies. Together, they are two of the most important technical reasons behind the success of the Internet.

As the Internet evolves into a global communication infrastructure, there is a growing need to support more sophisticated services (e.g., traffic management, QoS) than the traditional best-effort service. Two classes of solutions emerge: those maintaining the *stateless* property of the original IP architecture, and those requiring a new *stateful* architecture. Examples of *stateless* solutions are RED for congestion control [14] and Differentiated Service (Diffserv) [11] for QoS. The corresponding examples of *stateful* solutions are Fair Queueing [10] for congestion control and Integrated Service (Intserv) [3] for QoS. In general, stateful solutions can provide more powerful and flexible services. For example, compared with RED, Fair Queueing can protect well-behaving flows from misbehaving ones and accommodate heterogeneous end-to-end congestion control algorithms [19, 24]. Similarly, as discussed in Section 2, services provided by Intserv solutions have higher flexibility, utilization, and/or assurance level than those provided by Diffserv solutions. However, as also discussed in Section 2, stateful solutions are less scalable and robust than their stateless counterparts.

The question we want to answer is: is it possible to have the best of the two worlds, i.e., providing services as powerful as those implemented by stateful networks, while utilizing algorithms as scalable and robust as those used in stateless networks? While we cannot answer the above question in its full generality, we can answer it in some specific cases of practical interest. We consider a network architecture similar to the Diffserv architecture, called Scalable Core or SCORE, in which only edge routers perform per flow management, while core routers do not. As shown in Figure 1, the goal of a SCORE network is to approximate the service provided by a *reference stateful* network. In [28] we have shown that a SCORE network can achieve fair bandwidth allocation by approximating the service provided by a reference network in which every node performs fair queueing.

In this paper, we will show that a SCORE network can provide end-to-end per flow delay and bandwidth guarantees as defined in Intserv. Current Intserv solutions assume a stateful network in which two types of per flow state are needed: *forwarding state*, which is used by the forwarding engine to ensure fixed path forwarding, and *QoS state*[1], which is used by both the admission control module in the control plane and the classifier and scheduler in the data plane. In [29], we have proposed an algorithm that implements fixed path forwarding with no per flow forwarding state. In this paper, we focus on techniques to eliminate the need for core nodes to keep per flow QoS state. In particular, we propose two algorithms: one for the data plane to schedule packets, and the other for the control plane to perform admission control. Neither requires per flow state at core routers.

The key technique used to implement a SCORE network is Dynamic Packet State (DPS). With DPS, each packet carries in its header some state that is initialized by the ingress router. Core routers process each incoming packet

*This research was sponsored by DARPA under contract numbers N66001-96-C-8528 and E30602-97-2-0287, and by NSF under grant numbers Career Award NCR-9624979 and ANI-9814929. Additional support was provided by Intel Corp. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, Intel, or the U.S. government.

[1]In the context of RSVP, we use "QoS" state to refer to both the flow spec and the filter spec.

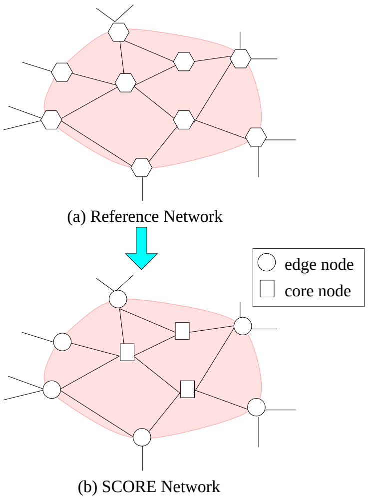


Figure 1: (a) A reference stateful network whose functionality is approximated by (b) a Scalable Core (SCORE) network. In SCORE only edge nodes perform per flow management; core nodes do not perform per flow management.

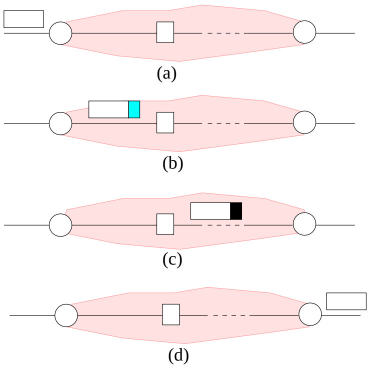


Figure 2: The illustration of the Dynamic Packet State (DPS) technique used to implement a SCORE network: (a-b) upon a packet arrival the ingress node inserts some state into the packet header; (b-c) a core node processes the packet based on this state, and eventually updates both its internal state and the packet state before forwarding it. (c-d) the egress node removes the state from the packet header.

based on the state carried in the packet's header, updating both its internal state and the state in the packet's header before forwarding it to the next hop (see Figure 2). By using DPS to coordinate actions of edge and core routers along the path traversed by a flow, distributed algorithms can be designed to approximate the behavior of a broad class of stateful networks using networks in which core routers do not maintain per flow state.

The rest of the paper is organized as follows. In Section 2, we give an overview of Intserv and Diffserv, and discuss the tradeoffs of these two architectures in providing QoS. In Sections 3 and 4 we present the details of our data and control path algorithms, respectively. Section 5 describes a design and a prototype implementation of the proposed algorithms in IPv4 networks. This demonstrates that it is indeed possible to implement algorithms with Dynamic Packet State techniques that have minimum incompatibility with existing protocols. Finally, we conclude the paper in Section 7.

2 Intserv and Diffserv

To support QoS in the Internet, the IETF has defined two architectures: the Integrated Services or Intserv [3], and the Differentiated Services or Diffserv [11]. They have important differences in both service definitions and implementation architectures. At the service definition level, Intserv provides end-to-end guaranteed [25] or controlled load service [37] on a per flow (individual or aggregate) basis, while Diffserv provides a coarser level of service differentiation among a small number of traffic classes. At the implementation level, current Intserv solutions require each router to process *per flow* signaling messages and maintain *per flow* data forwarding and QoS state on the control path, and to perform *per flow* classification, scheduling, and buffer management on the data path. Performing per flow management inside the network affects both the network *scalability* and *robustness*. The former is because the complexities of these per flow operations usually increase as a function of the number of flows; the later is because it is difficult to maintain the consistency of dynamic, and replicated per flow state in a distributed network environment. As pointed out by Clark in [5]: *"because of the distributed nature of the replication, algorithms to ensure robust replication are themselves difficult to build, and few networks with distributed state information provide any sort of protection against failure."* While there are several proposals that aim to reduce the number of flows inside the network by aggregating micro-flows that follow the same path into one macro-flow [2, 17], they only alleviate this problem, but do not fundamentally solve it — the number of macro flows can still be quite large in a network with many edge routers, as the number of paths is a quadratic function of the number of edge nodes.

Diffserv, on the other hand, distinguishes between edge and core routers. While edge routers process packets on the basis of finer traffic granularity, such as per flow or per organization, core routers do not maintain fine grain state, and process packets based on a small number of Per Hop Behaviors (PHBs) encoded by bit patterns in the packet header. By pushing the complexity to the edge and maintaining a simple core, Diffserv's *data plane* is much more scalable than Intserv. However, Diffserv still needs to address the problem of admission control on the control path. One proposal is to use a centralized bandwidth broker that maintains the topology as well as the state of all nodes in the network. In this case, the admission control can be implemented by the broker, eliminating the need for maintaining distributed reservation state. Such a centralized approach is more appropriate for an environment where most flows are long lived, and set-up and tear-down events are rare. To support fine grain and dynamic flows, there may be a need for a distributed broker architecture, in which the broker database is replicated or partitioned. Distributed broker architectures are still an active area of research. One can envision an architecture in which, when a broker receives a request, it makes an acceptance or rejection decision based on its own database, without consulting other brokers. This eliminates the need for a signaling protocol, but requires another protocol to maintain the consistency of the different broker databases. However, since it is impossible to achieve perfect consistency, this may lead to race conditions and/or resource fragmentation. In particular, since requests which arrive simultaneously at different brokers may want to reserve capacity along the same link, each broker can independently allocate only a fraction of the link capacity without running the risk of over-provisioning. This translates into a fundamental trade-off between scalability and fragmentation: while increasing the number of brokers make the solution more scalable, it also increases resource fragmentation.

While Diffserv is more scalable than Intserv in terms of implementation, services provided with existing Diffserv solutions usually have lower flexibility, utilization, and assurance levels than Intserv services. Two examples of differentiated service models are the assured service [6, 7] and the premium service [21]. The assured service is a form of statistical service and achieves lower assurance than guaranteed service. The premium service provides the equivalent of a dedicated link of fixed bandwidth between two edge nodes. However, as we have shown in [30], in order for the premium service to achieve service assurance comparable to the guaranteed service, even with a relative large queueing delay bound (e.g., 200 ms), the fraction of bandwidth that can be allocated to premium service traffic has to be very low (e.g., 10%). It is debatable whether these numbers should be of significant concern. For example, low utilization by the premium traffic may be acceptable if the majority of traffic will be best effort, either because the best effort service is "good enough" for most applications or the price difference between premium traffic and best effort traffic is too high to justify the performance difference between them. Alternatively, if the guaranteed nature of service assurance is not needed, i.e., statistical service assurance is sufficient for premium service, higher network utilization can be achieved. Providing meaningful statistical service is still an open research problem. A discussion of these topics is beyond the scope of this paper. For the rest of the paper, we assume that it is a desirable goal to provide guaranteed service and at the same time achieve high resource utilization.

In summary, Intserv provides more powerful service but has serious limitations with respect to network scalability and robustness. Diffserv is more scalable, but cannot provide services that are comparable to Intserv. In addition, scalable and robust admission control for Diffserv is still an open research problem.

3 QoS Scheduling Without Per Flow State

Current Intserv solutions assume a stateful network in which each router maintains per flow QoS state. The state is used by both the admission control module in the control plane and the classifier and scheduler in the data plane.

In this paper, we propose scheduling and admission control algorithms that provide guarantee services but do not require core routers to maintain per flow state. In this section, we present techniques that eliminate the need for data plane algorithms to use per flow state at core nodes. In particular, at core nodes, packet classification is no longer needed and packet scheduling is based on the state carried in packet headers, rather than per flow state stored locally at each node. In Section 4, we will show that fully distributed admission control can also be achieved without the need for maintaining per flow state at core nodes.

The main idea behind our solution is to approximate a *reference* stateful network with a SCORE network. The key technique used to implement approximation algorithms is Dynamic Packet State (DPS). With DPS, each packet carries some state which is initialized by the ingress node, and then updated by core nodes along the packet's path. The state is used by nodes traversed by the packet to coordinate actions and implement distributed algorithms. On the

Notation	Comments
p_i^k	the k-th packet of flow i
l_i^k	length of p_i^k
$a_{i,j}^k$	arrival time of p_i^k at node j
$s_{i,j}^k$	sending time of p_i^k at node j
$e_{i,j}^k$	eligible time of p_i^k at node j
$d_{i,j}^k$	deadline of p_i^k at node j
$g_{i,j}^k$	time ahead of deadline: $g_{i,j}^k = d_{i,j}^k - s_{i,j}^k$
δ_i^k	slack delay of p_i^k
π_j	propagation delay between nodes j and $j+1$

Table 1: Notations used in Section 3.

data path, our algorithm aims to approximate a network with every node implementing the Delay-Jitter-Controlled Virtual Clock (Jitter-VC) algorithm. We make this choice for several reasons. First, unlike various Fair Queueing algorithms [10, 23], in which a packet's deadline can depend on state variables of *all* active flows, in Virtual Clock a packet's deadline depends only on the state variables of the flow it belongs to. This property of Virtual Clock makes the algorithm easier to approximate in a SCORE network. In particular, the fact that the deadline of each packet can be computed exclusively based on the state variables of the flow it belongs to, makes possible to eliminate the need of replicating and maintaining per flow state at all nodes across the path. Instead, per flow state can be stored only at the ingress node, inserted into the packet header by the ingress node, and retrieved later by core nodes, which then use it to determine the packet's deadline. Second, by regulating traffic inside network using delay-jitter-controllers (discussed below), it can be shown that with very high probability, the number of packets in the server at any given time is significantly smaller than the number of flows (see Section 3.3). This helps to simplify the scheduler.

In the remainder of this section, we will first describe the implementation of Jitter-VC using per flow state, then present our algorithm, called Core-Jitter-VC (CJVC), which uses the technique of Dynamic Packet State (DPS). In [30] we present an analysis to show that a network of routers implementing CJVC provides the same delay bound as a network of routers implementing the Jitter-VC algorithm.

3.1 Jitter Virtual Clock (Jitter-VC)

Jitter-VC is a non-work-conserving version of the Virtual Clock algorithm [40]. It uses a combination of a delay-jitter rate-controller [33, 39] and a Virtual Clock scheduler. The algorithm works as follows: each packet is assigned an eligible time and a deadline upon its arrival. The packet is held in the rate-controller until it becomes eligible, i.e., the system time exceeds the packet's eligible time (see Figure 3(a)). The scheduler then orders the transmission of eligible packets according to their deadlines.

For the k^{th} packet of flow i, its eligible time $e_{i,j}^k$ and deadline $d_{i,j}^k$ at the j^{th} node on its path are computed as follows:

$$\begin{aligned} e_{i,j}^1 &= a_{i,j}^1 \\ e_{i,j}^k &= \max(a_{i,j}^k + g_{i,j-1}^k, d_{i,j}^{k-1}), \quad i,j \geq 1, k > 1 \quad (1) \\ d_{i,j}^k &= e_{i,j}^k + \frac{l_i^k}{r_i}, \quad i,j,k \geq 1 \quad (2) \end{aligned}$$

where l_i^k is the length of the packet, r_i is the reserved rate for the flow, $a_{i,j}^k$ is the packet's arrival time at the j^{th} node traversed by the packet, and $g_{i,j}^k$, stamped into the packet header by the previous node, is the amount of time the packet was transmitted before its deadline, i.e., the difference between the packet's deadline and its actual departure time at the $j-1^{th}$ node. Intuitively, the algorithm eliminates the delay variation of different packets by forcing all packets to incur the maximum allowable delay. The purpose of having $g_{i,j-1}^k$ is to compensate at node j the variation of delay due to load fluctuation at the previous node $j-1$. Such regulations limit the traffic burstiness caused by network load fluctuations, and as a consequence, reduce both buffer space requirements and the scheduler complexity.

It has been shown that if a flow's long term arrival rate is no greater than its reserved rate, a network of Virtual Clock servers can provide the same delay guarantee to the flow as a network of WFQ servers [13, 16, 27]. In addition, it has been shown that a network of Jitter-VC servers can provide the same delay guarantees as a network of Virtual Clock servers [9, 15]. Therefore, a network of Jitter-VC servers can provide the same guaranteed service as a network of WFQ servers.

3.2 Core-Jitter-VC (CJVC)

In this section we propose a variant of Jitter-VC, called Core-Jitter-VC (CJVC), which does not require per flow state at core nodes. In addition, we show that a network of CJVC servers can provide the same guaranteed service as a network of Jitter-VC servers.

CJVC uses the DPS technique. The key idea is to have the ingress node to encode scheduling parameters in each packet's header. The core routers can then make scheduling decisions based on the parameters encoded in packet headers, thus eliminating the need for maintaining per flow state at core nodes. As suggested by Eqs. (1) and (2), the Jitter-VC algorithm needs two state variables for each flow i: r_i, which is the reserved rate for flow i and $d_{i,j}^k$, which is the deadline of the last packet from flow i that was served by node j. While it is straightforward to eliminate r_i by putting it in the packet header, it is not trivial to eliminate $d_{i,j}^k$. The difference between r_i and $d_{i,j}^k$ is that while all nodes along the path keep the same r_i value for flow i, $d_{i,j}^k$ is a dynamic value that is computed iteratively at each node. In fact, the eligible time and the deadline of p_i^k depend on the deadline of the previous packet of the same flow, i.e., $d_{i,j}^{k-1}$.

A naive implementation using the DPS technique would be to pre-compute the eligible times and the deadlines of the packet at all nodes along its path and insert all of them in the header. This would eliminate the need for core nodes to maintain $d_{i,j}^k$. The main disadvantage of this approach is that the amount of information carried by the packet increases with the number of hops along the path. The challenge then is to design algorithms that compute $d_{i,j}^k$ for all nodes while requiring a minimum amount of state in the packet header.

Notice that in Eq. (1), the reason for node j to maintain $d_{i,j}^k$ is that it will be used to compute the deadline and the eligible time of the next packet. Since it is only used in a `max` operation, we can eliminate the need for $d_{i,j}^k$ if we can ensure that the other term in `max` is never less than $d_{i,j}^k$. The key idea is then to use a *slack* variable associated with each packet, denoted δ_i^k, such that for every core node j along the path, the following holds

$$a_{i,j}^k + g_{i,j-1}^k + \delta_i^k \geq d_{i,j}^{k-1}, \quad j > 1 \tag{3}$$

By replacing the first term of `max` in Eq. (1) with $a_{i,j}^k + g_{i,j-1}^k + \delta_i^k$, the computation of the eligible time reduces to

$$e_{i,j}^k = a_{i,j}^k + g_{i,j-1}^k + \delta_i^k, \quad j > 1 \tag{4}$$

Therefore, by using one additional DPS variable δ_i^k we eliminate the need for maintaining $d_{i,j}^k$ for in core nodes.

The derivation of δ_i^k proceeds in two steps. First, we express the eligible time of packet p_i^k at an arbitrary core node j, $e_{i,j}^k$, as a function of the eligible time of p_i^k at the ingress node $e_{i,1}^k$ (see Eq. (7)). Second, we use this result and Ineq. (4) to derive a lower bound for δ_i^k.

We now proceed with the first step. Recall that $g_{i,j-1}^k$ represents the time by which p_i^k is transmitted before its deadline at node $j-1$, i.e., $d_{i,j-1}^k - s_{i,j-1}^k$. Let π_{j-1} denote the propagation delay between nodes $j-1$ and j. Then the arrival time of p_i^k at node j, $a_{i,j}^k$, is given by

$$a_{i,j}^k = s_{i,j-1}^k + \pi_{j-1} = d_{i,j-1}^k - g_{i,j-1}^k + \pi_{j-1}. \tag{5}$$

By replacing $a_{i,j}^k$, given by the above expression, in Eq. (4), and then using Eq. (2), we obtain

$$e_{i,j}^k = d_{i,j-1}^k + \delta_i^k + \pi_{j-1} = e_{i,j-1}^k + \frac{l_i^k}{r_i} + \delta_i^k + \pi_{j-1} \tag{6}$$

By iterating over the above equation we express $e_{i,j}^k$ as a function of $e_{i,1}^k$:

$$e_{i,j}^k = e_{i,1}^k + (j-1)\left(\frac{l_i^k}{r_i} + \delta_i^k\right) + \sum_{m=1}^{j-1} \pi_m, \quad j > 1 \tag{7}$$

We are now ready to compute δ_i^k. Recall that the goal is to compute the minimum δ_i^k which ensures that Ineq. (3) holds for every node along the path. After combining Ineq. (3), Eq. (4) and Eq. (2) this reduces to ensure that

$$e_{i,j}^k \geq d_{i,j}^{k-1} \Rightarrow e_{i,j}^k \geq e_{i,j}^{k-1} + \frac{l_i^{k-1}}{r_i}, \quad j > 1 \tag{8}$$

By plugging $e_{i,j}^k$ and $e_{i,j}^{k-1}$ as expressed by Eq. (7) into Ineq. (8), we get

$$\delta_i^k \geq \delta_i^{k-1} + \frac{l_i^{k-1} - l_i^k}{r_i} + \frac{e_{i,1}^{k-1} + l_i^{k-1}/r_i - e_{i,1}^k}{(j-1)}, \quad j > 1 \tag{9}$$

From Eqs. (1) and (2) we have $e_{i,1}^k \geq d_{i,1}^{k-1} = e_{i,1}^{k-1} + l_i^{k-1}/r_i$. Thus, the right-hand side term in Ineq. (9) is maximized when $j = h$. As a result we compute δ_i^k as

$$\begin{aligned} \delta_i^1 &= 0, \quad (10) \\ \delta_i^k &= \max\left(0, \delta_i^{k-1} + \frac{l_i^{k-1} - l_i^k}{r_i} - \frac{e_{i,1}^k - e_{i,1}^{k-1} - l_i^{k-1}/r_i}{h-1}\right), \\ & \quad k > 1, h > 1. \end{aligned}$$

In this way, CJVC ensures that the eligible time of every packet p_i^k at node j is no smaller than the deadline of the previous packet of the same flow at node j, i.e., $e_{i,j}^k \geq d_{i,j}^{k-1}$. In addition, the Virtual Clock scheduler ensures that the deadline of every packet is met.[2]

[2]For simplicity we ignore here the transmission time of a packet of maximum size, τ_{max}, which represents the maximum time by which a packet can miss its deadline in the packet system [40]. Taking into account this term would not affect our results. For a complete discussion see [30].

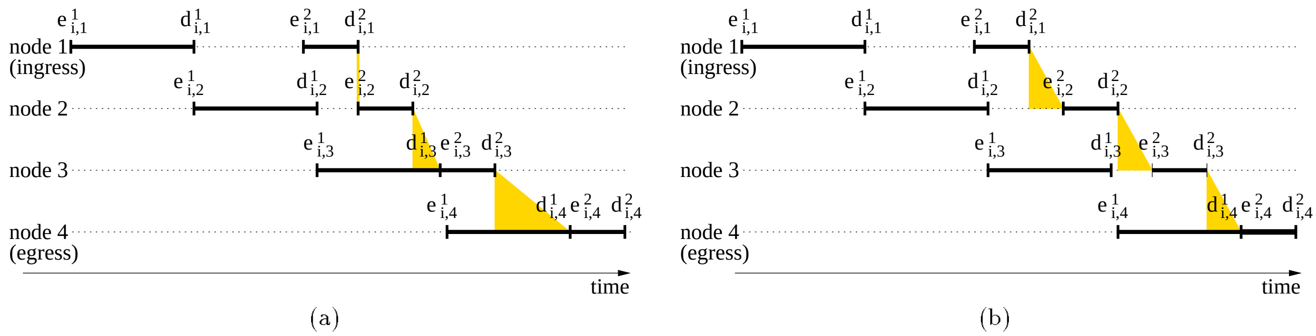


Figure 3: The time diagram of the first two packets of flow i along a four nodes path under (a) Jitter-VC, and (b) CJVC, respectively.

In [30], we have shown that a network of CJVC servers provide the same worst case delay bounds as a network of Jitter-VC servers. More precisely, we have proven the following result.

Theorem 1 *The deadline of a packet at the last hop in a network of CJVC servers is equal to the deadline of the same packet in a corresponding network of Jitter-VC servers.*

The example in Figure 3 provides some intuition behind the above result. The basic observation is that, with Jitter-VC, not counting the propagation delay, the difference between the eligible time of packet p_i^k at node j and its deadline at the previous node $j-1$, i.e., $e_{i,j}^k - d_{i,j-1}^k$, never *decreases* as the packet propagates along the path. Consider the second packet in Figure 3. With Jitter-VC, the differences $e_{i,j}^2 - d_{i,j-1}^2$ (represented by the bases of the gray triangles) increase in j. By introducing the slack variable δ_i^k, CJVC *equalizes* these delays. While this change may increase the delay of the packet at intermediate hops, it does not affect the end-to-end delay bound.

Figure 4 shows the computation of the scheduling parameters $e_{i,j}^k$ and $d_{i,j}^k$ by a CJVC server. The number of hops h is computed at the admission time as discussed in Section 4.1.

```
ingress node
on packet p arrival
   i = get_flow(p);
   if (first_packet_of_flow(p, i))
      e_i = current_time;
      δ_i = 0;
   else
      δ_i = max(0, δ_i + (l_i − length(p))/r_i −
            max(current_time − d_i, 0)/(h − 1)); /* Eq. (10) */
      e_i = max(current_time, d_i);
   l_i = length(p);
   d_i = e_i + l_i/r_i;
on packet p transmission
   label(p) ← (r_i, d_i − current_time, δ_i);
core/egress node
   on packet p arrival
      (r, g, δ) ← label(p);
      e = current_time + g + δ; /* Eq. (4) */
      d = e + length(p)/r
   on packet p transmission
      if (core node)
         label(p) ← (r, d − current_time, δ);
      else /* this is an egress node */
         clear_label(p);
```

Figure 4: Algorithms performed by ingress, core, and egress nodes at the packet arrival and departure. Note that core and egress nodes do not maintain per flow state.

3.3 Data Path Complexity

While our algorithms do not maintain per flow state at core nodes, there is still the need for core nodes to perform regulation and packet scheduling based on eligible times and deadlines. The natural question to ask is: why is this a more scalable scheme than previous solutions requiring per flow management?

There are several scalability bottlenecks for solutions requiring per flow management. On the data path, the expensive operations are per flow classification and scheduling. On the control path, the complexity is the maintenance of consistent and dynamic state in a distributed environment. Among the three, it is easiest to reduce the complexity of the scheduling algorithm as there is a natural tradeoff between the complexity and the flexibility of the scheduler [35]. In fact, a number of techniques have already been proposed to reduce the scheduling complexity, including those requiring constant time complexity [26, 36, 38].

We also note that due to the way we regulate traffic, it can be shown that with very high probability, the number of packets in the server at any given time is significantly smaller than the number of flows. This will further reduce the scheduling complexity and in addition reduce the buffer space requirement. More precisely, we have proven in [30] the following result.

Theorem 2 *Consider a server traversed by n flows. Assume that the arrival times of the packets from different flows are independent, and that all packets have the same size. Then, for any given probability ε, the queue size of the server during an arbitrary busy interval is bounded above by s, where*

$$s = \sqrt{\beta n(\log(n/\beta) - (\log \varepsilon)/2 - 1)}, \quad (11)$$

with a probability larger than $1-\varepsilon$. For identical reservations $\beta = 1$; for heterogeneous reservations $\beta = 3$.

As an example, let $n = 10^6$, and $\varepsilon = 10^{-10}$, which is the same order of magnitude as the probability of a packet being corrupted at the physical layer. Then, by Eq. (11) we obtain $s = 4932$ if all flows have identical reservations, and $s = 8348$ if flows have heterogeneous reservations. Thus the probability of having more packets in the queue than

# flows	bound (s)	max. queue size
100	31	28
1,000	109	100
10,000	374	284
100,000	1276	880
1,000,000	4310	2900

(a)

# flows	bound (s)	max. queue size
100	50	30
1,000	179	95
10,000	622	309
100,000	2134	904
1,000,000	7241	2944

(b)

Table 2: The upper bound of the queue size, s, computed by Eq. (11) for $\varepsilon = 10^{-5}$ versus the maximum queue size obtained over 10^5 independent trials: (a) when all flows have identical reservations; (b) when flows' reservations differ by a factor of 20.

specified by Eq. (11) can be neglected at the level of the entire system even in the context of guaranteed services.

In Table 2 we compare the bounds given by Eq. (11) to simulation results. In each case we report the maximum queue size obtained over 10^5 independent trials, and the corresponding bound computed by Eq. (11) for $\varepsilon = 10^{-5}$. The results show that our bounds are reasonably close (within a factor of two) when all reservations are identical, but are more conservative when the reservations are different. Finally, we make two comments. First, by performing per packet regulation at every core node, the bounds given by Eq. (11) hold for any core node and are *independent* of the path length. Second, if the flows' arrival patterns are not independent, we can easily enforce this by randomly delaying the first packet from each backlogged period of the flow at ingress nodes. This will increase the end-to-end packet delay by at most the queueing delay of one extra hop.

4 Admission Control With No Per Flow State

A key component of any architecture that provides guaranteed services is the admission control. The main job of the admission control is to ensure that the network resources are not over-committed. In particular it has to ensure that the sum of the reservation rates of all flows that traverse any link in the network is no larger than the link capacity, i.e., $\sum_i r_i < C$. A new reservation request is granted if it passes the admission test at each hop along its path. As discussed in Section 2, implementing such a functionality is not trivial: traditional distributed architectures based on signaling protocols are not scalable and are less robust due to the requirement of maintaining dynamic and replicated state; centralized architectures have scalability and availability concerns.

In this section, we propose a fully distributed architecture for implementing admission control. Like most distributed admission control architectures, in our solution, each node keeps track of the aggregate reservation rate for each of its out-going links and makes local admission control decisions. However, unlike existing reservation protocols, this distributed admission control process is achieved without core nodes maintaining per flow state.

4.1 Ingress-to-Egress Admission Control

We consider an architecture in which a lightweight signaling protocol is used within the SCORE domain. Edge routers are the interface between this signaling protocol and an inter-domain signaling protocol such as RSVP. For the purpose of this discussion, we consider only unicast reservations. In addition, we assume a mechanism like the one proposed in [29] or Multi-Protocol Label Switching (MPLS) [4] that can be used to pin a flow to a route.

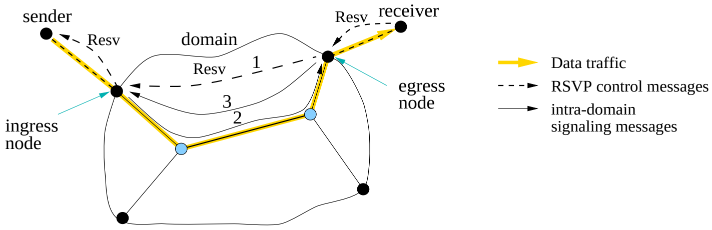


Figure 5: Ingress-egress admission control when RSVP is used outside the SCORE domain.

From the point of view of RSVP, a path through the SCORE domain is just a virtual link. There are two basic control messages in RSVP: *Path* and *Resv*. These messages are processed only by edge nodes; no operations are performed inside the domain. For the ingress node, upon receiving a *Path* message, it simply forwards it through the domain. For the egress node, upon receiving the first *Resv* message for a flow (i.e., there was no RSVP state for the flow at the egress node before receiving the message), it will forward the message (message "1" in Figure 5) to the corresponding ingress node, which in turn will send a special signaling message (message "2" in Figure 5) along the path toward the egress node. Upon receiving the signaling message, each node along the path performs a local admission control test as described in Section 4.2. In addition, the message carries a counter h that is incremented at each hop. The final value h is used for computing the slack delay δ (see Eq. (10)). If we use the route pinning mechanism described in [29], message "2" is also used to compute the label of the path between the ingress and egress. This label is used then by the ingress node to make sure that all data packets of the flow are forwarded along the same path. When the signaling message "2" reaches the egress node, it is reflected back to the sender, which makes the final decision (message "3" in Figure 5). RSVP refresh messages for a flow that already has per flow RSVP state installed at edge routers will not trigger additional signaling messages inside the domain.

Since RSVP uses raw IP or UDP to send control messages, there is no need for retransmission for our signaling messages, as message loss will not break the RSVP semantics. If the sender does not receive a reply after a certain timeout, it simply drops the *Resv* message. In addition, as we will show in Section 4.3, there is no need for a special termination message inside the domain when a flow is torn down.

4.2 Per-Hop Admission Control

Each node needs to ensure that $\sum_i r_i < C$ holds at all times. At first sight, one simple solution that implements this test and also avoids per flow state is for each node to maintain the

Notation	Comments
r_i	flow i's reserved rate
b_i^k	total number of bits flow i is entitled to transmit during $[s_{i,1}^{k-1}, s_{i,1}^k]$, i.e., $b_i^k = r_i(s_{i,1}^k - s_{i,1}^{k-1})$
$R(t)$	aggregate reservation at time t
$R_{bound}(t)$	upper bound of $R(t)$, used by admission test
$R_{DPS}(t)$	estimate of $R(t)$, computed by using DPS
$R_{new}(t)$	sum of all new reservations accepted from the beginning of current estimation interval until t
$R_{cal}(t)$	upper bound of $R(t)$, used to calibrate R_{bound}, computed based on R_{DPS} and R_{new}

Table 3: Notations used in Section 4.3.

aggregate reserved rate R, where R is updated to $R = R + r$ when a new flow with the reservation rate r is admitted, and to $R = R - r'$ when a flow with the reservation rate r' terminates. The admission control reduces then to checking whether $R + r \leq C$ holds. However, it can be easily shown that such a simple solution is not robust with respect to various failure conditions such as packet loss, partial reservation failures, and network node crashes. To handle packet loss, when a node receives a set-up or tear-down message, the node has to be able to tell whether it is a duplicate of a message already processed. To handle partial reservation failures, a node needs to "remember" what decision it made for the flow in a previous pass. That is why all existing solutions maintain per flow reservation state, be it hard state as in ATM UNI or soft state as in RSVP. However, maintaining *consistent* and *dynamic* state in a *distributed* environment is itself challenging. Fundamentally, this is due to the fact that the update operations assume a *transaction* semantic, which is difficult to implement in a distributed environment [1, 34].

In the remaining of the section, we show that by using DPS, it is possible to significantly reduce the complexity of admission control in a distributed environment. Before we present the details of the algorithm, we point out that our goal is to estimate a close *upper bound* on the aggregate *reserved* rate. By using this bound in the admission test we avoid over-provisioning, which is a necessary condition to provide deterministic service guarantees. This is in contrast to many measurement-based admission control algorithms [18, 32], which, in the context of supporting controlled load or statistical services, base their admission test on the measurement of the *actual* amount of traffic transmitted. To achieve this goal, our algorithm uses two techniques. First, a conservative upper bound of R, denoted R_{bound}, is maintained at each core node and is used for making admission control decisions. R_{bound} is updated with a simple rule: $R_{bound} = R_{bound} + r$ whenever a new request of a rate r is accepted. It should be noted that in order to maintain the invariant that R_{bound} is an upper bound of R, this algorithm does not need to detect duplicate request messages, generated either due to retransmission in case of packet loss or retry in case of partial reservation failures. Of course, the obvious problem with this algorithm is that R_{bound} will diverge from R. In the limit, when R_{bound} reaches the link capacity C, no new requests can be accepted even though there might be available capacity.

To address this problem, a separate algorithm is introduced to periodically estimate the aggregate reserved rate. Based on this estimate, a second upper bound for R, denoted R_{cal}, is computed and used to re-calibrate R_{bound}. An important aspect of the estimation algorithm is that the discrepancy between the upper bound R_{cal} and the actual reserved rate R can be bounded. The re-calibration then becomes choosing the minimum of the two upper bounds R_{bound} and R_{cal}. The estimation algorithm is based on DPS and does not require core routers to maintain per flow state.

Our algorithms have several important properties. First, they are robust in the presence of network losses and partial reservation failures. Second, while they can over-estimate R, they will never under-estimate R. This ensures the semantics of the guaranteed service – while over-estimation can lead to under-utilization of network resources, under-estimation can result in over-provisioning and violation of performance guarantees. Finally, the proposed estimation algorithms are self-correcting in the sense that over-estimation in a previous period will be corrected in the next period. This greatly reduces the possibility of serious resource under-utilization.

4.3 Aggregate Reservation Estimation Algorithm

In this section, we present the estimation algorithm of the aggregate reserved rate which is performed at each core node. In particular, we will describe how R_{cal} is computed and how it is used to re-calibrate R_{bound}. In designing the algorithm for computing R_{cal}, we want to balance between two goals: (a) R_{cal} should be an upper bound on R; (b) over-estimation errors should be corrected and kept to the minimum.

To compute R_{cal}, we start with an inaccurate estimate of R, denoted R_{DPS}, and then make adjustments to account for estimation inaccuracies. In the following, we first present the algorithm that computes R_{DPS}, then describe the possible inaccuracies and the corresponding adjustment algorithms.

The estimate R_{DPS} is calculated using the DPS technique: ingress nodes insert additional state in packet headers, which is in turn used by core nodes to estimate the aggregate reservation R. In particular, the following state b_i^k is inserted in the header of packet p_i^k:

$$b_i^k = r_i(s_{i,1}^k - s_{i,1}^{k-1}), \tag{12}$$

where $s_{i,1}^{k-1}$ and $s_{i,1}^k$ are the times the packets p_i^{k-1} and p_i^k are transmitted by the ingress node. Therefore, b_i^k represents the total amount of bits that flow i is entitled to send during the interval $[s_{i,1}^{k-1}, s_{i,1}^k]$. The computation of R_{DPS} is based on the following simple observation: the sum of b values of all packets of flow i during an interval is a good approximation for the total number of bits that flow i is *entitled* to send during that interval according to its reserved rate. Similarly, the sum of b values of *all* packets is a good approximation for the total number of bits that all flows are entitled to send during the corresponding interval. Dividing this sum by the length of the interval gives the aggregate reservation rate. More precisely, let us divide time into intervals of length T_W: $(u_k, u_{k+1}]$, $k > 0$. Let $b_i(u_k, u_{k+1})$ be the sum of b values of packets in flow i received during $(u_k, u_{k+1}]$, and let $B(u_k, u_{k+1})$ be the sum of b values of *all* packets during $(u_k, u_{k+1}]$. The estimate is then computed at the end of each interval $(u_k, u_{k+1}]$ as follows

$$R_{DPS}(u_{k+1}) = \frac{B(u_k, u_{k+1})}{u_{k+1} - u_k} = \frac{B(u_k, u_{k+1})}{T_W}. \tag{13}$$

While simple, the above algorithm may introduce two types of inaccuracies. First, it ignores the effects of the

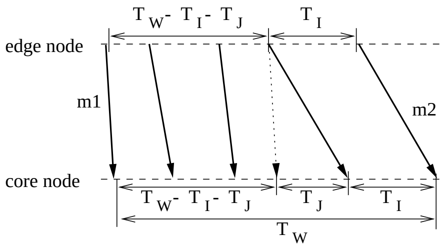


Figure 6: The scenario in which the lower bound of b_i, i.e., $r_i(T_W - T_I - T_J)$, is achieved. The arrows represent packet transmissions. T_W is the averaging window size; T_I is an upper bound on the packet inter-departure time; T_J is an upper bound on the delay jitter. Both $m1$ and $m2$ miss the estimation interval T_W.

delay jitter and the packet inter-departure times. Second, it does not consider the effects of accepting or terminating a reservation in the middle of an estimation interval. In particular, having newly accepted flows in the interval may result in the under-estimation of $R(t)$ by $R_{DPS}(t)$. To illustrate this, consider the following simple example: there are no guaranteed flows on a link until a new request with rate r is accepted at the end of an estimation interval $(u_k, u_{k+1}]$. If no data packet from the new flow reaches the node before u_{k+1}, $B(u_k, u_{k+1})$ would be 0, and so would be $R_{DPS}(u_{k+1})$. However, the correct value should be r.

In the following, we present the algorithm to compute an upper bound of $R(u_{k+1})$, denoted $R_{cal}(u_{k+1})$. In doing this we account for both types of inaccuracies. Let $\mathcal{L}(t)$ denote the set of reservations at time t. Our goal is then to bound the aggregate reservation at time u_{k+1}, i.e., $R(u_{k+1}) = \sum_{i \in \mathcal{L}(u_{k+1})} r_i$. Consider the division of $\mathcal{L}(u_{k+1})$ into two subsets: the subset of new reservations that were accepted during the interval $(u_k, u_{k+1}]$, denoted $\mathcal{N}(u_{k+1})$, and the subset containing the rest of reservations which were accepted no later than u_{k+1}. Next, we express $R(u_{k+1})$ as

$$R(u_{k+1}) = \sum_{i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})} r_i + \sum_{i \in \mathcal{N}(u_{k+1})} r_i. \qquad (14)$$

The idea is then to derive an upper bound for each of the two right-hand side terms, and compute R_{cal} as the sum of these two bounds. To bound $\sum_{i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})} r_i$, we note that

$$B(u_k, u_{k+1}) \geq \sum_{i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})} b_i(u_k, u_{k+1}). \qquad (15)$$

The reason that (15) is an inequality instead of an equality is that when there are flows terminating during the interval $(u_k, u_{k+1}]$, their packets may still have contributed to $B(u_k, u_{k+1})$ even though they do not belong to $\mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})$. Next, we compute a lower bound for $b_i(u_k, u_{k+1})$. By definition, since $i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})$, it follows that flow i holds a reservation during the entire interval $(u_k, u_{k+1}]$. Let T_I be the maximum inter-departure time between two consecutive packets of a flow at the edge node, and let T_J be the maximum delay jitter of a flow, where both T_I and T_J are much smaller than T_W. Now, consider the scenario shown in Figure 6 in which a core node receives the packets $m1$ and $m2$ just outside the estimation window. Assuming the worst case in which $m1$ incurs the lowest possible delay, $m2$ incurs the maximum possible delay, and that the last packet before $m2$ departs T_I seconds earlier, it is easy to see that that the sum of the b values carried by the packets received during the estimation interval by the core node cannot be smaller than $r_i(T_W - T_I - T_J)$. Thus, we have

$$\begin{aligned} b_i(u_k, u_{k+1}) &> r_i(T_W - T_I - T_J), & (16)\\ & \quad \forall i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1}). & (17) \end{aligned}$$

By combining Ineqs. (15) and (16), and Eq. (13) we obtain

$$\begin{aligned} \sum_{i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})} r_i &< \sum_{i \in \mathcal{L}(u_{k+1}) \setminus \mathcal{N}(u_{k+1})} \frac{b_i(u_k, u_{k+1})}{T_W(1-f)} \\ &\leq \frac{R_{DPS}(u_{k+1})}{1-f}, \qquad (18) \end{aligned}$$

where $f = (T_I + T_J)/T_W$.

Next, we bound the second right-hand side term in Ineq. (14): $\sum_{i \in \mathcal{N}(u_{k+1})} r_i$ For this, we introduce a new global variable R_{new}. R_{new} is initialized at the beginning of each interval $(u_k, u_{k+1}]$ to zero, and is updated to $R_{new} + r$ every time a new reservation r is accepted. Let $R_{new}(t)$ denote the value of this variable at time t. For simplicity, here we assume that a flow which is granted a reservation during the interval $(u_k, u_{k+1}]$ becomes active no later than u_{k+1}.[3] Then it is easy to see that

$$\sum_{i \in \mathcal{N}(u_{k+1})} r_i \leq R_{new}(u_{k+1}). \qquad (19)$$

The inequality holds when no duplicate reservation requests are processed, and none of the new accepted reservations terminate during the interval. Then we define $R_{cal}(u_{k+1})$ as

$$R_{cal}(u_{k+1}) = \frac{R_{DPS}(u_{k+1})}{1-f} + R_{new}(u_{k+1}). \qquad (20)$$

From Eq. (14), and Ineqs. (18) and (19) follow easily that $R_{cal}(u_{k+1})$ is an upper bound for $R(u_{k+1})$, i.e., $R_{cal}(u_{k+1}) > R(u_{k+1})$. Finally, we use $R_{cal}(u_{k+1})$ to re-calibrate the upper bound of the aggregate reservation, R_{bound}, at u_{k+1} as

$$R_{bound}(u_{k+1}) = \min(R_{bound}(u_{k+1}), R_{cal}(u_{k+1})). \qquad (21)$$

Figure 7 shows the pseudocode of control algorithms at core nodes. Next we make several observations.

First, the estimation algorithm uses only the information in the current interval. This makes the algorithm robust with respect to loss and duplication of signaling packets since their effects are "forgotten" after one time interval. As an example, if a node processes both the original and a duplicate of the same reservation request during the interval $(u_k, u_{k+1}]$, R_{bound} will be updated twice for the same flow. However, this erroneous update will not be reflected in the computation of $R_{DPS}(u_{k+2})$, since its computation is based only on the b values received during $(u_{k+1}, u_{k+2}]$.

As a consequence, an important property of our admission control algorithm is that it can asymptotically reach a link utilization of $C(1-f)/(1+f)$. In particular, the following result is proven in [30]:

Theorem 3 *Consider a link of capacity C at time t. Assume that no reservation terminates and there are no reservation failures or request losses after time t. Then if there is sufficient demand after t the link utilization approaches asymptotically $C(1-f)/(1+f)$.*

[3] Otherwise, to account for the worst case in which a reservation that was accepted by the node during $(u_{k-1}, u_k]$ becomes at time $u_k + RTT$, we need to subtract $RTT \times R_{new}(u_k)$ from $B(u_k, u_{k+1})$.

Per-hop Admission Control

on reservation request r
if $(R_{bound} + r \leq C)$ /* *perform admission test* */
  $R_{new} = R_{new} + r$;
  $R_{bound} = R_{bound} + r$;
  accept request;
else
  deny request;
on reservation termination r /* *optional* */
  $R_{bound} = R_{bound} - r$;

Aggregate Reservation Bound Comp.

on packet arrival p
  $b \leftarrow get\_b(p)$; /* *get b value inserted by ingress (Eq. (12))* */
  $L = L + b$;
on time-out T_W
  $R_{DPS} = L/T_W$; /* *estimate aggregate reservation* */
  $R_{bound} = \min(R_{bound}, R_{DPS}/(1-f) + R_{new})$;
  $R_{new} = 0$;

Figure 7: The control path algorithms executed by core nodes; R_{new} is initialized to 0.

Second, note that since $R_{cal}(u_k)$ is an upper bound of $R(u_k)$, a simple solution would be to use $R_{cal}(u_k) + R_{new}$, instead of R_{bound}, to perform the admission test during $(u_k, u_{k+1}]$. The problem with this approach is that R_{cal} can overestimate the aggregate reservation R. An example is given in Section 5.3 to illustrate this issue (Figure 13(b)).

Third, we note that a possible optimization of the admission control algorithm is to add reservation termination messages (see Figure 7). This will reduce the discrepancy between the upper bound R_{bound} and the aggregate reservation R. However, in order to guarantee that R_{bound} remains an upper bound for R, we need to ensure that a termination message is sent at most *once*, i.e., there are no retransmissions if the message is lost. In practice, this property can be enforced by edge nodes, which maintain per flow state.

Finally, to ensure that the maximum inter-departure time is no larger than T_I, the ingress node may need to send a dummy packet in the case when no data packet arrives for a flow during an interval T_I. This can be achieved by having the ingress node to maintain a timer with each flow. An optimization would be to aggregate all "micro-flows" between each pair of ingress and egress nodes into one flow, and compute b values based on the aggregated reservation rate, and insert a dummy packet only if there is no data packet of the aggregate flow during an interval.

5 Implementation and Experiments

The key technique of our algorithms is DPS, which encodes states in the packet header, and thus eliminates the need for maintaining per flow state at each node. Since there is limited space in protocol headers and most header bits have been allocated, the main challenge of implementing these algorithms is to (a) find space in the packet header for storing DPS variables and at the same time remain fully compatible with current standards and protocols; and (b) efficiently encode state variables so that they fit in the available space without introducing too much inaccuracy.

In the remaining of the section, we will first present how we address the above two problems in the context of IPv4 networks, describe a prototype implementation of our algorithms in FreeBSD v2.2.6, and, finally we give results from experiments in local testbed. The main goal of these experiments is to provide a proof of concept of our design.

5.1 Carrying State in Data Packets

Two possibilities to encode state in the packet header are: (1) introduce a new IP option and insert the option at the ingress router, or (2) introduce a new header between layer 2 and layer 3, similar to the way labels are transported in Multi-Protocol Label Switching (MPLS) [4]. While both of these solutions are quite general and can potentially provide large space for encoding state variables, for the propose of our implementation we consider a third option: store the state in the IP header. By doing this, we avoid the penalty imposed by most IPv4 routers in processing the IP options, or the need of devising different solutions for different technologies as it would have been required by introducing a new header between layer 2 and layer 3.

The biggest problem with using the IP header is to find enough space to insert the extra information. The main challenge is to remain compatible with current standards and protocols. In particular, we want the network domain to be transparent to end-to-end protocols, i.e., the egress node should restore the fields changed by ingress and core nodes to their original values. To achieve this goal, we first use four bits from the type of service (TOS) byte (now renamed the Differentiated Service (DS) field) bits which are specifically allocated for local and experimental use [20]. In addition, we observe that there is an *ip_off* field of 13 bits in the IPv4 header to support packet fragmentation/reassembly which is rarely used. For example, by analyzing the traces of over 1.7 million packets on an OC-3 link [22], we found that less than 0.22% of all packets were fragments. Therefore, in most cases it is possible to use *ip_off* field to encode the DPS values. This idea can be implemented as follows. When a packet arrives at an ingress node, the node checks whether a packet is a fragment or needs to be fragmented. If neither of these are true, the *ip_off* field in the packet header will be used to encode DPS values. When the packet reaches the egress node, the *ip_off* is cleared. Otherwise, if the packet is a fragment, it is forwarded as a best-effort packet. In this way the use of *ip_off* is transparent outside the domain. We believe that forwarding a fragment as a best-effort packet, is acceptable in practice, as end-points can easily avoid fragmentation by using an MTU discovery mechanism. Also note that in the above we implicitly assume that packets can be fragmented only by egress nodes.

In summary, we have up to 17 bits available in the current IPv4 header to encode four state variables. The next section discusses how we use this space to encode the DPS states.

5.2 State Encoding

There are four pieces of state that need to be encoded: three are for scheduling purposes, (1) the reserved rate r or equivalently l/r, (2) δ, as computed by Eq. (10), and (3) g; and one for admission control purpose, (4) b. All are positive values.

One possible solution is to restrict each state variable to only a small number of possible values. For example if a state variable is limited to eight values, only three bits are needed to represent it. While this can be a reasonable solution in practice, in our implementation we use a more sophisticated representation based on a floating point like format. The details of this representation are presented

```
void intToFP(int val, int *mantissa, int *exponent) {
  int nbits = get_num_bits(val);
  if (nbits <= m) {
    *mantissa = val;
    *exponent = (1 << n) - 1;
  } else {
    *exponent = nbits - m - 1;
    *mantissa = (val >> *exponent) - (1 << m);
  }
}
```

```
int FPToInt(int mantissa, int exponent) {
  int tmp;
  if (exponent == ((1 << n) - 1))
    return mantissa;
  tmp = mantissa | (1 << m);
  return (tmp << exponent)
}
```

Figure 8: The C code for converting between integer and floating point formats. m represents the number of bits used by the mantissa; n represents the number of bits in the exponent. Only positive values are represented. The exponent is computed such that the first bit of the mantissa is always 1, when the number is $\geq 2^m$. By omitting this bit, we gain an extra bit in precision. If the number is $< 2^m$ we set by convention the exponent to $2^n - 1$ to indicate this.

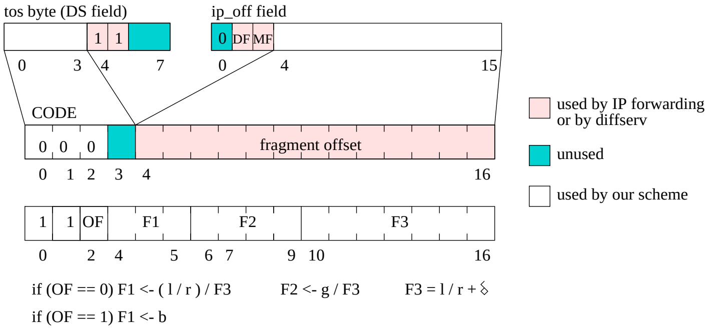


Figure 9: For carrying state we use the four bits from the TOS byte (or DS field) reserved for local use and experimental purposes, and up to 13 bits from the *ip_off*. The first three bits specify whether *ip_off* is used to encode DPS variables. F1, F2, and F3 are used to encode the DPS variables corresponding to a data packet (codes 11x identify the state in data packet headers).

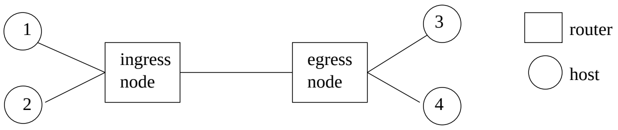


Figure 10: The test configuration used in experiments.

in [30]. Here, we show only the C code of the conversion between this representation and an integer representation (see Figure 8). To further optimize the use of the available space we employ two additional techniques. First, we use the floating point format only to represent the *largest* value, and then represent the other value(s) as a fraction of the largest value. In this way we are able to represents a much larger range of possible values. Second, in the case in which there are states which are not required to be simultaneously encoded in the same packet, we use the same field to encode them.

Figure 9 shows how the 17 bits available in the current IPv4 header are used to encode DPS states in a data packet. The 17 bits are divided in four fields: a *code* field which specifies whether the *ip_off* is used to encode state variables, and three *data* fields, denoted $F1$, $F2$ and $F3$, used to encode our variables.

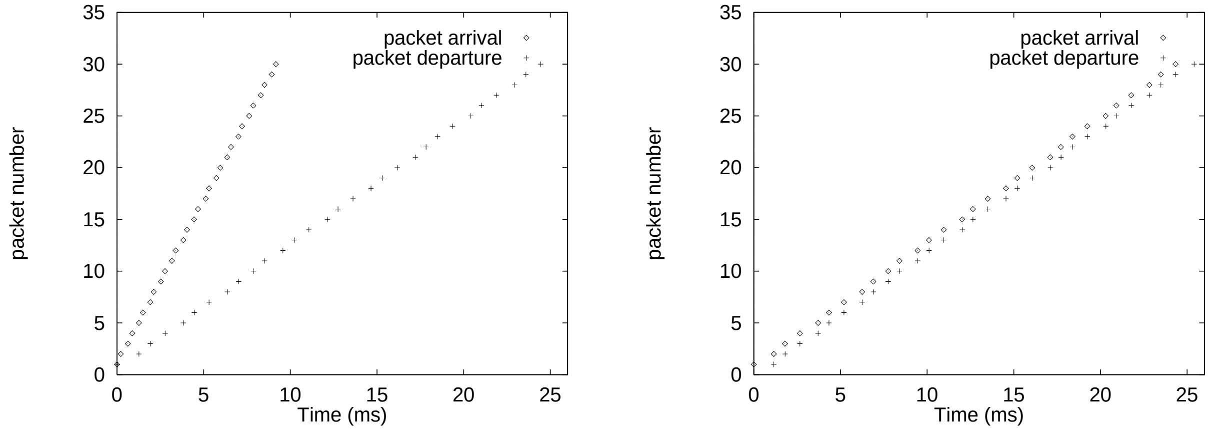
The code field consists of three bits: 000 means that the packet is a fragment and therefore no state is encoded; any other value means that up to 13 bits of *ip_off* are used to encode the state. In particular, the code values specify the layout and the states encoded in the packet header. For example, 11x specifies that the encoded states correspond to a data packet, while 100 specifies that the encoded states correspond to a dummy packet. Due to space limitations, in Figure 9 we show the state encoding for a data packet only. In this case, the last bit of the code field, also called *Offset Field* (OF), determines the content of $F1$. If this bit is 1, then $F1$ encodes the b value. Otherwise it encodes $(l/r)/F3$, where $F3 = l/r + \delta$. Finally, $F2$ encodes $g/F3$. We make several observations. First, since $F3$ encodes the largest value among all fields, we represent it in floating point format [30]. By using this format, with seven bits we can represent any positive number in the range $[1..15 \times 2^{15}]$, with a relative error within $(-6.25\%, 6.25\%)$ [30]. Second, since the deadline determines the delay guarantees, we use a representation that trades the eligible time accuracy[4] for the deadline accuracy. In particular, the deadline is computed as $d = current\_time + F2 * F3 + F3 \simeq current\_time + g + l/r + \delta$. If OF is 0, the eligible time is computed as $e = d - F1 * F3 \simeq current\_time + g + \delta$. $F1$ uses only three bits and its value is computed such that $F1 * F3$ always over-estimates l/r. If OF is 1, the eligible time is computed simply as $e = current\_time$. Third, we express b in units equals with the maximum packet size. In this way we eliminate the need for each packet to carry the b value. In fact, if a flow sends at its reserved rate, only one packet every other eight packets needs to carry the b value. This observation, combined with the fact that the under-estimation of the packet eligible time does not affect the guaranteed delay of the flow, allows us to alternatively encode either b or $(l/r)/F3$ in $F1$, without impacting the correctness of our algorithms.

5.3 Experimental Results

We have implemented these algorithms in FreeBSD v2.2.6 and deployed them in a testbed consisting of 266 MHz and 300 MHz Pentium II PCs connected by point-to-point 100 Mbps Ethernets. The testbed allows configuring a path with up to two intermediate routers.

In the following, we present results from four simple experiments. The experiments are designed to illustrate the microscopic behaviors of the algorithms, rather than their scalability. All experiments were run on the topology shown in Figure 10. The first router is configured as an ingress

[4] As long as the eligible time value is under-estimated, its inaccuracy will affect only the scheduling complexity, as the packet may become eligible earlier.



(a) (b)

Figure 11: Packet arrival and departure times for a 10 Mbps flow at (a) the ingress node, and (b) the egress node.

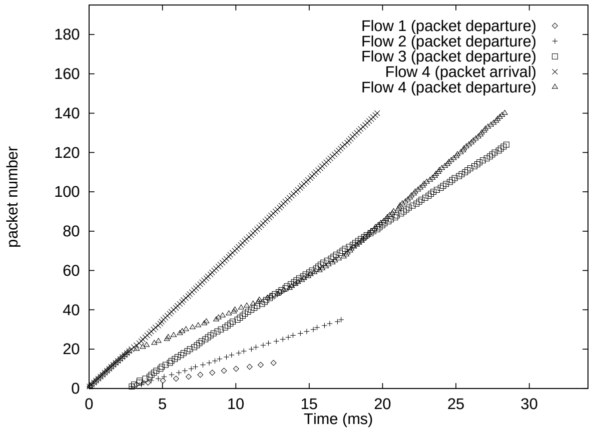


Figure 12: The packets' arrival and departure times for four flows. The first three flows are guaranteed, with reservations of 10 Mbps, 20 Mbps, and 40 Mbps. The last flow is best effort with an arrival rate of about 60 Mbps.

node, while the second router is configured as an egress node. An egress node also implements the functionalities of a core node. In addition, it restores the initial values of the *ip_off* field. All traffic is UDP and all packets are 1000 bytes, not including the header.

In the first experiment we consider a flow between hosts 1 and 3 that has a reservation of 10 Mbps but sends at a much higher rate of about 30Mbps. Figures 11(a) and (b) plot the arrival and departure times for the first 30 packets of the flow at the ingress and egress node, respectively. One thing to notice in Figure 11(a) is that the arrival rate at the ingress node is almost three times the departure rate, which is the same as the reserved rate of 10 Mbps. This illustrate the non-work-conserving nature of the CJVC algorithm, which enforces the traffic profile and allows only 10 Mbps traffic into the network. Another thing to notice is that all packets incur about 0.8 ms delay in the egress node. This is because they are sent by the ingress node as soon as they become eligible, and therefore $g \simeq l/r = 8 \times 1052\text{bits}/10\text{Mbps} = 0.84$ ms. As a result, they will be held in the rate-controller for this amount of time at the next hop[5], which is the egress node in our case.

In the second experiment we consider three guaranteed flows between hosts 1 and 3 with reservations of 10 Mbps, 20 Mbps, and 40 Mbps, respectively. In addition, we consider a fourth UDP flow between hosts 2 and 4 which is treated as best effort. The arrival rates of the first three flows are slightly larger than their reservations, while the arrival rate of the fourth flow is approximately 60 Mbps. At time 0, only the best-effort flow is active. At time 2.8 ms, the first three flows become simultaneously active. Flows 1 and 2 terminate after sending 12 and 35 packets, respectively. Figure 12 shows the packet arrival and departure times for the best-effort flow 4, and the packet departure times for the real-time flows 1, 2, and 3. As can be seen, the best-effort packets experience very low delay in the initial period of 2.8 ms. After the QoS flows become active, best-effort packets experience longer delays while QoS flows receive service at their reserved rate. After flow 1 and 2 terminate, the best-effort traffic grabs the remaining bandwidth.

The last two experiments illustrate the algorithms for admission control described in Section 4.3. The first experiment demonstrates the accuracy of estimating the aggregate reservation based on the b values carried in the packet headers. The second experiment illustrates the computation of the aggregate reservation bound, R_{bound}, when a new reservation is accepted or a reservation terminates. In these experiments we use an averaging interval, T_W, of 5 seconds, and a maximum inter-departure time, T_I, of 500 ms. For simplicity, we neglect the delay jitter, i.e., we assume $T_J = 0$. This gives us $f = (T_I + T_J)/T_W = 0.1$.

In the first experiment we consider two flows, one with a reservation of 0.5 Mbps, and the other with a reservation of 1.5 Mbps. Figure 13(a) plots the arrival rate of each flow, as well as the arrival rate of the aggregate traffic. In addition, Figure 13(a) plots the bound of the aggregate reservation used by admission test, R_{bound}, the estimate of the aggregate reservation R_{DPS}, and the bound R_{cal} used to recalibrate R_{bound}. According to the pseudocode in Figure 7, both R_{DPS} and R_{cal} are updated at the end of each estimation interval. More precisely, every 5 seconds R_{DPS} is computed based on the b values carried in the packet headers, while R_{cal} is computed as $R_{DPS}/(1-f) + R_{new}$. Note

[5]Note that since all packets have the same size, $\delta = 0$.

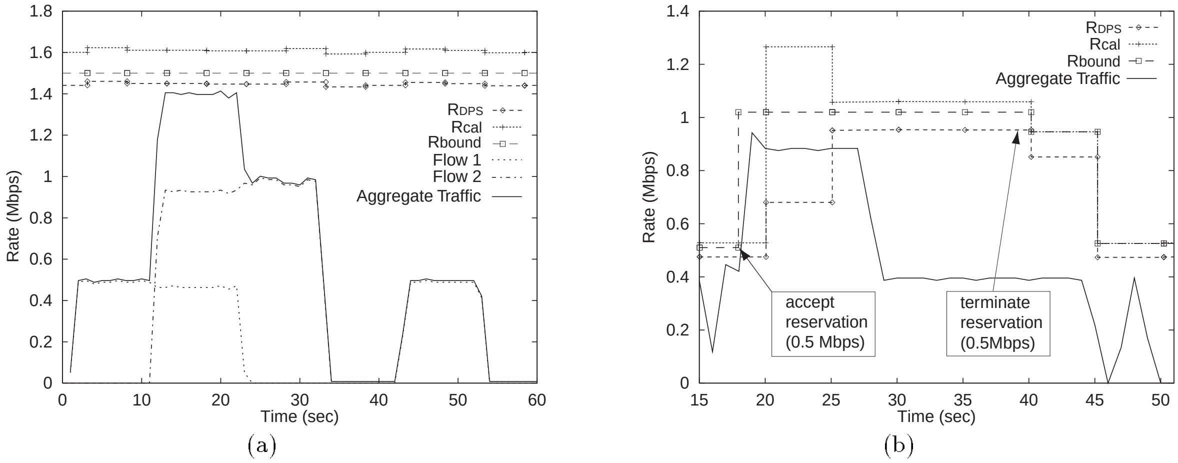


Figure 13: The estimate aggregate reservation R_{cal}, and the bounds R_{bound} and R_{cal} in the case of (a) two ON-OFF flows with reservations of 0.5 Mbps, and 1.5 Mbps, respectively, and in the case when (b) one reservation of 0.5 Mbps is accepted at time $t = 18$ seconds, and then is terminated at $t = 39$ seconds.

that since in this case no new reservation is accepted, we have $R_{new} = 0$, which yields $R_{cal} = R_{DPS}/(1 - f)$. The important thing to note in Figure 13(a) is that the rate variation of the actual traffic (represented by the continuous line) has little effect on the accuracy of computing the aggregate reservation estimate R_{DPS}, and consequently of R_{cal}. In contrast, traditional measurement based admission control algorithms, which base their estimation on the *actual* traffic, would significantly under-estimate the aggregate reservation, especially during the time periods when no data packets are received. In addition, note that since in this experiment R_{cal} is always larger than R_{bound}, and no new reservations are accepted, the value of R_{bound} is never updated.

In the second experiment we consider a scenario in which a new reservation of 0.5 Mbps is accepted at time $t = 18$ seconds and terminates approximately at time $t = 39$ seconds. For the entire time duration, plotted in Figure 13(b), we have a background traffic with an aggregate reservation of 0.5 Mbps. Similarly to the previous case, we plot the rate of the aggregate traffic, and, in addition, R_{bound}, R_{cal}, and R_{DPS}. There are several points worth noting. First, when the reservation is accepted at time $t = 18$ seconds, R_{bound} increases by the value of the accepted reservation, i.e., 0.5 Mbps (see Figure 7). In this way, R_{bound} is guaranteed to remain an upper bound of the aggregate reservation R. In contrast, since both R_{DPS} and R_{cal} are updated only at the end of the estimation interval, they under-estimate the aggregate reservation, as well as the aggregate traffic, before time $t = 20$ seconds. Second, after R_{cal} is updated at time $t = 20$ seconds, as $R_{DPS}/(1-f) + R_{new}$, the new value significantly over-estimates the aggregate reservation. This is the main reason for which we do not use R_{cal} $(+R_{new})$, but R_{bound}, to do the admission control test. Third, note that unlike the case when the reservation was accepted, R_{bound} does not change when the reservation terminates at time $t = 39$ seconds. This is simply because in our implementation no tear-down message is generated when a reservation terminates. However, as R_{cal} is updated at the end of the next estimation interval (i.e., at time $t = 45$ seconds), R_{bound} drops to the correct value of 0.5 Mbps. This shows the importance of using R_{cal} to recalibrate R_{bound}. In addition, this illustrates the robustness of our algorithm, i.e., the over-estimation in a previous period is corrected in the next period. Finally, note that in both experiments R_{DPS} always under-estimates the aggregate reservation. This is due to the truncation errors in computing both the b values and the R_{DPS} estimate.

5.4 Processing Overhead

To evaluate the overhead of our algorithm we have performed three experiments on a 300 MHz Pentium II involving 1, 10, and 100 flows, respectively. The reservation and actual sending rates of all flows are identical. The aggregate sending rate is about 20% larger than the aggregate reservation rate. Table 4 shows the means and the standard deviations for the enqueue and dequeue times at both ingress and egress nodes. Each of these numbers is based on a measurement of 1000 packets. For comparison we also show the enqueue and dequeue times for the unmodified code. There are several points worth noting. First, our implementation adds less than 5 μs overhead per enqueue operation, and about 2 μs per dequeue operation. In addition, both the enqueue and dequeue times at the ingress node are greater than at the egress node. This is because ingress node performs per flow operations. Furthermore, as the number of flows increases the enqueue times increase only slightly, i.e., by less than 20%. This suggests that our algorithm is indeed scalable in the number of flows. Finally, the dequeue times actually *decrease* as the number of flows increases. This is because the rate-controller is implemented as a calendar queue with each entry corresponding to a 128 μs time interval. Packets with eligible times falling between the same interval are stored in the same entry. Therefore, when the number of flows is large, more packets are stored in the same calendar queue entry. Since all these packets are transferred during one operation when they become eligible, the actual overhead per packet decreases.

6 Related Work

Our scheme shares its intellectual roots with two pieces of related work: Diffserv and the Core-Stateless Fair Queueing.

The idea of implementing QoS services by using a core-stateless architecture was first proposed by Jacobson [21]

	Baseline		1 flow				10 flows				100 flows			
			ingress		egress		ingress		egress		ingress		egress	
	avg	std	avg	std	avg	std	avg	std	avg	std	avg	std	avg	std
enqueue	1.03	0.91	5.02	1.63	4.38	1.55	5.36	1.75	4.60	1.60	5.91	1.81	5.40	2.33
dequeue	1.52	1.91	3.14	3.27	2.69	2.81	2.79	3.68	2.30	2.91	2.77	2.82	1.73	2.12

Table 4: The average and standard deviation of the enqueue and dequeue times, measured in μs.

and Clark [7], and is now being pursued by the IETF Diffserv working group [11]. There are several differences between our scheme and the existing Diffserv proposals. First, our algorithms operate at a much finer granularity both in terms of time and traffic aggregates: the state embedded in a packet can be highly dynamic, as it encodes the *current* state of the flow, rather than the static and global properties such as dropping or scheduling priority. In addition, the goal of our scheme is to implement distributed algorithms that try to approximate the services provided by a network in which all routers implement per flow management. Therefore, we can provide service differentiation and performance guarantees on a *per flow* basis. In contrast, existing Diffserv solutions provide service differentiation only among a small number of traffic classes. Finally, we propose fully distributed and dynamic algorithms for implementing both data and control functionalities, where existing Diffserv solutions rely on more centralized and static algorithms for implementing admission control.

We first proposed the idea of using Dynamic Packet State to encode dynamic per flow state in the context of approximating the Fair Queueing algorithm in a SCORE architecture [28]. While algorithms proposed in this paper share the same architecture as CSFQ, there are important differences both in high level goals and low level mechanisms. First, while CSFQ was designed to support best-effort traffic, algorithms proposed here are designed to support guaranteed services. As a consequence, while CSFQ can use a probabilistic forwarding algorithm to statistically approximate the Fair Queueing service, CJVC needs to use more elaborate mechanisms to provide performance guarantees *identical* to those provided by Virtual Clock or Weighted Fair Queueing algorithms. In particular, CJVC uses three types of Dynamic Packet State for scheduling purpose and regulates traffic at each hop. One more type of Dynamic Packet State was used to implement the admission control, which was not needed in CSFQ. Finally, we have proposed a detailed design for encoding the DPS variables in IPv4.

In this paper, we propose a technique to estimate the aggregate reservation rate and use that estimate to perform admission control. While this may look similar to measurement-based admission control algorithms [18, 32], the objectives and thus the techniques are quite different. The measurement-based admission control algorithms are designed to support controlled-load type of services, the estimation is based on the *actual* amount of traffic transmitted in the past, and is usually an *optimistic* estimate in the sense that the estimated aggregate rate is smaller than the aggregate reserved rate. While this has the benefit of increasing the network utilization by the controlled-load service traffic, it has the risk of incurring transient overloads that may cause the degradation of QoS. In contrast, our algorithm aims to support guaranteed service, and the goal is to estimate a close upper bound on the aggregate *reserved* rate even when the the actual arrival rate may vary.

In [8], Cruz proposed a novel scheduling algorithm called SCED+ in the context of ATM networks. In SCED+, virtual circuits sharing a same path segment are aggregated into a virtual path. At each switch, only per virtual path state instead of per virtual circuit needs to be maintained for scheduling purpose. In addition, an algorithm is proposed to compute, the eligible times and the deadlines of a packet at subsequent nodes, when the packet enters a virtual path. We note that by doing this and using DPS to carry this information in the packets' headers, it is possible to remove per path scheduling state from core nodes. However, unlike our solution, SCED+ do not provide per flow delay differentiation within an aggregate. In addition, the SCED+ work focuses on the data path mechanism, while we addresses both data path and control path issues.

7 Conclusion

In this paper, we developed two distributed algorithms that implement QoS scheduling and admission control in a SCORE network where core routers do not maintain per flow state. Combined, these two algorithms significantly enhance the scalability of both the data and control plane mechanisms for implementing guaranteed services, and at the same time, provide guaranteed services with flexibility, utilization, and assurance levels similar to those that can be provided with per flow mechanisms. The key technique used in both algorithms is called Dynamic Packet State (DPS), which provides a lightweight and robust means for routers to coordinate actions and implement distributed algorithms. By presenting a design and prototype implementation of the proposed algorithms in IPv4 networks, we demonstrate that it is indeed possible to apply DPS techniques and have minimum incompatibility with existing protocols.

As a final note, we believe DPS is a powerful concept. By using DPS to coordinate actions of edge and core routers along the path traversed by a flow, distributed algorithms can be designed to approximate the behavior of a broad class of "stateful" networks with networks in which core routers do not maintain per flow state. We observe that it is possible to extend the current Diffserv framework to accommodate algorithms using Dynamic Packet State such as the ones proposed in this paper and Core-Stateless Fair Queueing [28]. The key extension needed is to associate with each Per Hop Behavior (PHB) additional space in the packet header for storing PHB specific Dynamic Packet State [31]. Such a paradigm will significantly increase the flexibility and capabilities of the services that can be built with a Diffserv-like architecture.

References

[1] Ö. Babaoğlu and S. Toueg. Non-Blocking Atomic Commitment. *Distributed Systems, S. Mullender (ed.)*, pages 147–168, 1993.

[2] F. Baker, C. Iturralde, F. Le Faucheur, and B. Davie. Aggregation of RSVP for IP4 and IP6 Reservations. Internet Draft, draft-baker-rsvp-aggregation-00.txt.

[3] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview, June 1994. Internet RFC 1633.

[4] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan. A framework for multiprotocol label switching, November 1997. Internet Draft, draft-ietf-mpls-framework-02.txt.

[5] D. Clark. The design philosophy of the DARPA internet protocols. In *Proceedings of ACM SIGCOMM'88*, pages 106–114, Stanford, CA, August 1988.

[6] D. Clark. Internet cost allocation and pricing. *Internet Economics, L. W. McKnight and J. P. Bailey (eds.)*, pages 215–252, 1997.

[7] D. Clark and J. Wroclawski. An approach to service allocation in the Internet, July 1997. Internet Draft.

[8] R. L. Cruz. SCED+: Efficient Management of Quality of Service Guarantees. In *Proceedings of INFOCOM'98*, pages 625–642, San Francisco, CA, 1998.

[9] R.L. Cruz. Quality of service guarantees in virtual circuit switched network. *IEEE Journal on Selected Areas in Communications*, 13(6):1048–1056, August 1995.

[10] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Journal of Internetworking Research and Experience*, pages 3–26, October 1990. Also in Proceedings of ACM SIGCOMM'89, pp 3-12.

[11] Y. Bernet et. al. A framework for differentiated services, November 1998. Internet Draft, draft-ietf-diffserv-framework-01.txt.

[12] D. Ferrari and D. Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, April 1990.

[13] N. Figueira and J. Pasquale. An upper bound on delay for the virtualclock service discipline. *IEEE/ACM Transactions on Networking*, 3(4), April 1995.

[14] S. Floyd and V. Jacobson. Random early detection for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, July 1993.

[15] L. Georgiadis, R. Guerin, V. Peris, and K. Sivarajan. Efficient network QoS provisioning based on per node traffic shaping. *IEEE/ACM Transactions on Networking*, 4(4):482–501, August 1996.

[16] P. Goyal, S. Lam, and H. Vin. Determining end-to-end delay bounds in heterogeneous networks. In *Proceedings of the 5th International Workshop on Network and Operating System Support For Digital Audio and Video*, pages 287–298, Durham, New Hampshire, April 1995.

[17] R. Guerin, S. Blake, and S. Herzog. Aggregating RSVP-based QoS Requests. Internet Draft, draft-guerin-aggreg-rsvp-00.txt.

[18] S. Jamin, P. Danzig, S. Shenker, and L. Zhang. A measurement-based admission control algorithm for integrated services packet networks. In *Proceedings of SIGCOMM'95*, pages 2–13, Boston, MA, September 1995.

[19] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM'91*, pages 3–15, Zurich, Switzerland, September 1991.

[20] K. Nichols, S. Blake, F. Baker, and D. L. Black. Definition of the Differentiated Services Field (DS Field) in the ipv4 and ipv6 Headers, October 1998. Internet Draft, draf-ietf-diffserv-header-04.txt.

[21] K. Nichols, V. Jacobson, and L. Zhang. An approach to service allocation in the Internet, November 1997. Internet Draft.

[22] NLANR. Network Traffic Packet Header Traces. URL: http://moat.nlanr.net/Traces/.

[23] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control - the single node case. In *Proceedings of the INFOCOM'92*, 1992.

[24] S. Shenker. Making greed work in networks: A game theoretical analysis of switch service disciplines. In *Proceedings of ACM SIGCOMM'94*, pages 47–57, London, UK, August 1994.

[25] S. Shenker, C. Partridge, and R. Guerin. Specification of guaranteed quality of service, September 1997. Internet RFC 2212.

[26] D.C. Stephens, J.C.R. Bennett, and H. Zhang. Implementing scheduling algorithms in high speed networks. *To Appear in IEEE JSAC*, 1999.

[27] D. Stilliadis and A. Verma. Latency-rate servers: A general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(2):164–174, April 1998.

[28] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *Proceedings of ACM SIGCOMM*, Vancouver, CA, August 1998.

[29] I. Stoica and H. Zhang. Lira: A model for service differentiation in the internet. In *NOSSDAV'98*, London, UK, July 1998.

[30] I. Stoica and H. Zhang. Providing guranteed services without per flow management, May 1999. Technical Report CMU-CS-99-133.

[31] I. Stoica, H. Zhang, S. Shenker, R. Yavatkar, D. Stephens, Y. Bernet, Z. Wang, F. Baker, J. Wroclawski, and R. Wilder C. Song. Per hop behaviors based on dynamic packet states, February 1999. Internet Draft.

[32] D. Tse and M. Grosslauser. Measurement-based Call Admission Control: Analysis and Simulation. In *Proceedings of INFOCOM'97*, pages 981–989, Kobe, Japan, 1997.

[33] D. Verma, H. Zhang, and D. Ferrari. Guaranteeing delay jitter bounds in packet switching networks. In *Proceedings of Tricomm'91*, pages 35–46, Chapel Hill, North Carolina, April 1991.

[34] W. E. Weihl. Transaction-Processing Techniques. *Distributed Systems, S. Mullender (ed.)*, pages 329–352, 1993.

[35] D. Wrege, E. Knightly, H. Zhang, and J. Liebeherr. Deterministic delay bounds for vbr video packet-switching networks: Fundmental limits and practical trade-offs. *IEEE/ACM Transactions on Networking*, 4(3):352–362, June 1996.

[36] D.E. Wrege and J. Liebeherr. A Near-Optimal Packet Scheduler for QoS Networks. In *Proceedings of INFOCOM'97*, pages 576–583, Kobe, Japan, 1997.

[37] J. Wroclawski. Specification of controlled-load network element service, 1997. Internet RFC 2211.

[38] H. Zhang and D. Ferrari. Rate-controlled static priority queueing. In *Proceedings of IEEE INFOCOM'93*, pages 227–236, San Francisco, California, April 1993.

[39] H. Zhang and D. Ferrari. Rate-controlled service disciplines. *Journal of High Speed Networks*, 3(4):389–412, 1994.

[40] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proceedings of ACM SIGCOMM'90*, pages 19–29, Philadelphia, PA, September 1990.