Packet-Pair Flow Control

Srinivasan Keshav

AT&T Bell Laboratories
600 Mountain Avenue, Murray Hill, NJ 07974, USA
keshav@research.att.com

Abstract

This paper presents the packet-pair rate-based feedback flow control scheme. This scheme is designed for networks where individual connections do not reserve bandwidth and for the available bitrate (best-effort) component of integrated networks. We assume a round-robin-like queue service discipline in the output queues of the network's switches, and propose a linear stochastic model for a single conversation in a network of such switches. These model motivates the Packet-Pair rate probing technique, which forms the basis for provably stable discrete and continuous time rate-based flow control schemes. We present a novel state estimation scheme based on fuzzy logic. We then address several practical concerns: dealing with system startup, retransmission and timeout strategy, and dynamic setpoint probing. We present a finite state machine as well as source code for a model implementation. The dynamics of a single source, the interactions of multiple sources, and the behavior of packet-pair sources in a variety of benchmark scenarios are evaluated by means of detailed simulations. We close with some remarks about possible extensions to packet-pair, limitations of this work, and an outline of related work.

1. Introduction

The emerging paradigm for data communication is based on the concept of an integrated network, where traffic from constant bitrate, variable bitrate and bursty sources are intelligently multiplexed to provide quality of service guarantees to individual connections. Integrated networks are expected to provide at least two classes of service. In the first class, users regulate their traffic to fit within some behavior envelope, and in exchange the network gives them some guarantees of performance. For example, a traffic source may agree to obey a leaky bucket descriptor, in exchange for a bound on its expected loss rate. In the second class, sources do not specify a bound on their traffic, and in return, are not given performance guarantees. These sources must adapt to changing network conditions in order to achieve their data transfer goals. Such a service is useful for sources that are incapable of describing their expected behavior, since this behavior may be bursty or unpredictable. The second class of service, which we call available bitrate (ABR) service, is compatible with existing computer networks, such as the Internet and various LAN technologies, and hence it is likely to be an important component of future networks as well. In this paper, we present a scheme for congestion control of the ABR component of integrated networks which is equally applicable to existing datagram-oriented networks.

Congestion control for ABR traffic involves packet (or cell) schedulers at queuing points and the flow control protocol at traffic sources [12, 34]. We first consider the role of packet schedulers in congestion control. Recent work has shown that there is a strong motivation to use round-robin-like packet schedulers for bursty data traffic, since this provides several advantages over the traditional first-come-first-served discipline [12, 24, 34, 44, 46]. Round-robin service automatically enforces a min-max fair allocation of resources [20]. It also automatically polices sources, since a source sending faster than its fair share is the one that will be subjected to packet or cell loss. This ensures that well-behaved users are protected from ill-behaved users, which is desirable in public data networks [18]. Due to these advantages, it is likely that round-robin schedulers will be widely implemented. packet-pair flow control is one way to exploit the properties of such schedulers to do intelligent flow control.

While 'packet-pair' strictly refers only to the technique for probing the bottleneck bandwidth availability, we extend the meaning of 'packet-pair flow control' to include several additional components. These are:

1) The technique for probing network state
2) The estimators that smooth this state information
3) The control law that uses the smoothed network state
4) The technique for flow control at startup or when current network state information is not available
5) The strategy used to compute timeout intervals
6) The retransmission strategy
7) The buffer management strategy

Taken together, these schemes and strategies are intended to provide a complete solution to the problem of flow control for best-effort sources in high speed networks of round-robin-like servers.

The paper is presented in roughly three stages: theory, implementation and simulations. In Section 2 we present and justify an analytic model for networks of round-robin like servers. This model serves as a basis for the control law, the choice of estimators and the packet-pair probing technique discussed in Section 3. Section 4 presents a strategy to use the state information in a control framework, followed by the a discrete and continuous time control scheme, their stability analysis, and the design of a fuzzy-logic based state estimators. Sections 5-7 present schemes for startup, timeout and retransmission strategy and buffer management strategy. Section 8 discusses implementation considerations, and Section 9 is a detailed simulation study of the scheme. Section 10 discusses some possible extensions to packet-pair, and Section 11 presents some conclusions as well as a review of related work.

2. Network Model

In this section, we present and justify an analytical model for a network of round-robin-like servers. We first describe several variants of round-robin service, then present a generic model. We will assume here that all packets from a source to a destination traverse the same unicast route. This is true for virtual circuit networks (such as ATM networks) and datagram networks where typical transport (or higher) layer connect times are shorter than the routing table update time.

2.1. Round Robin and Rate Allocating Servers

Packet servers (or cell servers in ATM networks) are associated with every queueing point in a packet-switched network, and are typically present at the output queues of packet switches. A round-robin server is one where each active connection is associated with a logical or physical per-connection data queue and the server serves non-empty data queues in turn. The time duration spanned by a visit of the server to all non-empty data queues is called a *round*. If a single packet is served per round of service, the scheduler is strict round-robin. If more than one packet is served per round, the scheduler is called weighted round-robin.

If the packet size is small and fixed, then simple round robin service provides an equal bandwidth share to each connection. When packet sizes are variable and potentially large, a packet must receive service in inverse proportion to its length in order to achieve fair bandwidth allocation. This can be done by tagging packets with the completion time had the service been head-of-line processor sharing, and then serving packets in increasing order of their tags, as in Fair Queueing [12]. It can be shown that Fair Queueing emulates head-of-line processor sharing asymptotically with conversation length [21, 22]. Connections may be allocated unequal bandwidth shares by varying the weights, as is done in the Weighted Fair Queueing discipline.

If the round spans a constant time duration, with the output trunk being kept idle if necessary (i.e. a non-work-conserving discipline), this is called framed round robin service. A multilevel framed round robin server is also called a hierarchical round robin server [28]. The network model (and hence the flow control scheme) described in this paper is adequate to describe variants of round-robin service where the packet size per connection is fixed (as in ATM networks), or service is weighted to reflect the packet size (as with Fair Queueing [12], Weighted Fair Queueing, Packetized Generalized Processor Sharing [46], and Virtual Clock [65]). We call this kind of round-robin server a *Rate-Allocating Server*, (RAS) since at each round a connection is allocated a service rate depending on the number of active connections and their weights, and relatively independent of the traffic arrival patterns at the other connections. We say relatively independent, since a connection that is only sporadically active will increase and decrease the number of active connections, and in this way influence the service rate allocated to other connections.

Note that even with these variations in the service rate, a RAS provides a conversation with a more consistent service rate than a first-come-first-served (FCFS) server. In a FCFS server, the service rate of a conversation is linked in detail to the arrival pattern of every other conversation in the server, and so the perceived service rate varies rapidly. For example, consider the situation where the number of conversations sending data to a server is fixed, and each conversation always has data to send when it is scheduled for service. In a FCFS server, if any one conversation sends a large burst of data, then the service rate of all the other conversations effectively drops until the burst has been served. In a RAS, the other conversations will be unaffected. Thus, the server allocates a rate of service to each conversation that is, to a first approximation, independent of the arrival patterns of the other conversations.

2.2. Stochastic Model for a Conversation

We now present a stochastic model for a conversation in a network of RASs. This is an extension of the deterministic model presented in References [53, 58, 59]. We model a conversation in a RAS network as a sequence of regularly spaced packets from a source to a destination over a series of servers connected by links. The servers in the path of the conversation are numbered 1,2,3...n, and the source is numbered 0. The destination is assumed to acknowledge each packet. We also assume, for ease of analysis, that sources always have data to send. This simplification allows us to ignore start-up transients in our analysis. In fact, the start-up costs can be significant, and these are analyzed in [53].

The time taken for service at the ith server at time t is denoted $s_i(t)$, and the (instantaneous) service rate is defined to be $\mu_i(t) = 1/s_i(t)$. s_i includes the time taken to serve packets from all other conversations in round-robin order, the round time. Thus, the service rate is the inverse of the time between consecutive packet services from the same conversation.

The source sending rate is denoted by λ and the source is assumed to send packets spaced exactly $s_0 = 1/\lambda$ time units apart. We define

$$s_b(t) = \max_i (s_i(t) \mid 0 \le i \le n)$$

to be the *bottleneck* service time in the conversation, and b is the index of the bottleneck server. $\mu_b(t)$, the bottleneck service rate, is defined to be $1/s_b(t)$. $\mu_b(t)$ changes due to changes in the number of active conversations. We model these changes over discrete time intervals, so that $\mu(t)$ changes at discrete time steps $1,\ 2,\ ...,\ k,\ k+1\ \cdots$. The choice of step duration is discussed in Section 4.1.2.

If the number of active conversations, N_{ac}, is large, we expect that the change in N_{ac} in one time interval will be small compared to N_{ac}. Hence the change in μ_b in one interval will be small and $\mu_b(k+1)$ will be 'close' to $\mu_b(k)$. We model $\mu_b(k)$ as a random walk. This represents the fact that the cross traffic at a bottleneck point is uncontrollable and suffers random variations. This model is simple, and though it represents the dynamics only to first order, we have found that it is sufficient for effective control in practice. Thus, we define

$$\mu_b(k+1) = \max(\mu_b(k) + \omega(k), 0)$$

where $\omega(k)$ is a random variable. We make some assumptions on the distribution of ω depending on the choice of estimator for $\mu_b(k)$. This is discussed in Sections 4.4 and 4.5.

By modelling the effect of cross traffic as a random noise variable, we have essentially linearized the system, except for the 'max' operator. This makes the control task relatively straightforward. Linearization is possible because RAS service isolates a conversation's service from the details of the arrival pattern of cross traffic. For FCFS service, there is no such isolation, and the associated non-linear control is complex [17, 55].

3. Packet-pair probing

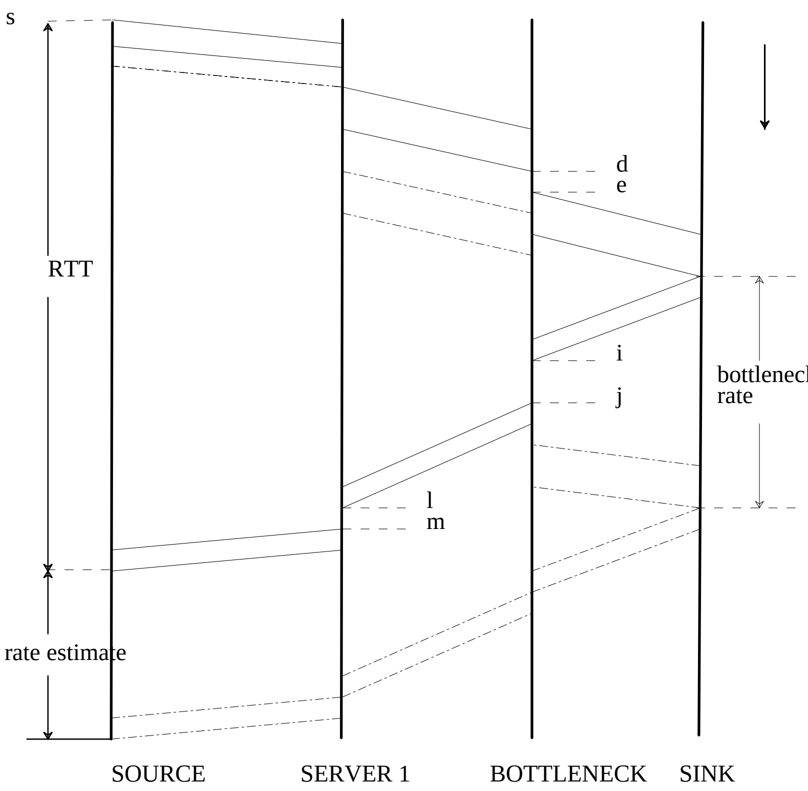

Figure 1: Packet-pair probing

The packet-pair mechanism is to send all data in the form of back-to-back pairs, and then estimate the bottleneck service rate from the spacing of the acknowledgements (Figure 1). The figure presents a time diagram, with time increasing along the vertical axis. Each axis represents either a source, sink or queueing point in a communication network. The parallelograms represent the transmission of a packet and correspond to two kinds of delays: the vertical sides are as long as the transmission delay (the packet size divided by the line capacity). The slope of the longer sides is proportional to the propagation delay. Since the network is store-and-forward, a packet cannot be sent till it is completely received. After a packet arrives, it may be queued for a while before it receives service. This is represented by the space between two dotted lines, such as *de*.

In the Packet-Pair scheme, the source sends out two back-to-back packets (at time s). Let the size of the second packet be P. These two packets are then served by the bottleneck; by definition, if the inter-packet service time is s_b, the instantaneous service rate of the source at the bottleneck, μ_b is P/s_b. Since the acks preserve this spacing s_b, and P is known by the source, it can measure the inter-ack spacing to estimate μ_b.

We now consider possible sources of error in the estimate. The server marked 'SERVER 1' also spaces out the back-to-back packets, so can it affect the measurement of μ_b? A moment's reflection reveals that as long as the second packet in the pair arrives at the bottleneck before the bottleneck ends service for the first packet, there is no problem. If the packet does arrive after this time, then, by definition, server 1 itself is the bottleneck. Hence, the spacing out of packets at servers before the bottleneck server is of no consequence, and does not introduce errors into the scheme. Another factor that does not introduce error is that the first packet may arrive when the server is serving other packets and may be delayed, for example, by *de*. Since this delay is common to both packets in the pair, this does not affect estimation of service capacity.

What does introduce an error is the fact that the acks may be spread out more (or less) than s_b due to different queueing delays for each ack along the return path. In the figure note that the first ack has a net queueing delay of $ij + lm$, and the second has a zero queueing delay. This has the effect of increasing the estimate of μ.

Note that this source of error will persist even if the inter-ack spacing is noted at the sink and sent to the source using a state exchange scheme [50]. Measuring μ_b at the sink will reduce the effect of noise, but cannot eliminate it, since any server that is after the bottleneck potentially introduces noise in the measurement.

We model this error in measurement as an observation noise. Since the observed value of s_b can be either increased or decreased by this noise, with equal probability in either direction, we expect that the noise distribution is symmetric about 0.

4. Flow Control Design

We now present a flow control algorithm that makes uses of the probed values of μ_b. The philosophy behind the design is to make the necessary simplifications and assumptions that enable a analytical design, then use simulations to refine the design and test the validity of the assumptions. In this section, we present the theoretical framework for the design, and in Section 9 we present validation by extensive simulation.

4.1. Design Strategy

The design strategy for the flow control mechanism is based upon the Separation Theorem [2]. Informally, the theorem states that for a linear stochastic system where an observer estimates the system state and then uses this estimate to do control, the eigenvalues of the state estimator and the controller are separable. The theorem allows us to use any technique for state estimation, and implement control using the estimated state $\hat{x}$ instead of the actual state x. Thus, we will derive a control law assuming that all required estimators are available; the estimators are derived in Section 4.5. We first present a few preliminary considerations.

4.1.1. Choice of Setpoint

The aim of the control is to maintain the number of packets in the bottleneck queue, n_b at a desired setpoint. Since the system has delay components, it is not possible for the control to stay at the setpoint at all times. Instead, the system will oscillate around the setpoint value. We assume for the moment that each queueing point allocates B buffers to each conversation, and that B is a static quantity known to the source. (This assumption will be relaxed in Section 7.) Then, the choice of the setpoint reflects a tradeoff between mean packet delay, buffer usage, packet loss and bandwidth loss (which is the bandwidth a conversation loses because it has no data to send when it is eligible for service).

Consider the distribution of n_b for the controlled system, given by $N(x) = Pr(n_b = x)$. $N(x)$ is bounded on the left by 0 and on the right by B and contains information about three things:

1) Pr(loss of bandwidth) = Pr (RAS server schedules the conversation for service $\mid n_b = 0$). Assuming that these events are independent, which is a reasonable assumption, we find that Pr(loss of bandwidth) is proportional to $N(0)$.

2) Similarly, Pr (loss of packet) = Pr (packet arrival $\mid n_b = B$), so that the density at B, denoted $N(B)$, is proportional to the probability of a packet loss.

3) the mean queuing delay is given by

$$E(s_b) \int_0^B xN(x)\,dx,$$

where, a packet takes an average of $E(s_b)$ units of time to get service at the bottleneck.

If the setpoint is small, then the distribution of $N(x)$ is driven towards the left, the probability of bandwidth loss increases, the mean packet delay is decreased, and the probability of packet loss is decreased. Thus, we trade off bandwidth loss for lower mean delay and packet loss. Similarly, if we choose a large setpoint, we will trade off packet loss for a larger mean delay and lower probability of bandwidth loss. In the sequel, we assume a setpoint of $B/2$. The justification is that, since the system noise is symmetric, and the control tracks the system noise, we expect $N(x)$ to be symmetric around the setpoint. In that case, a setpoint of $B/2$ balances the two tradeoffs. Since any other choice of setpoint can be chosen without loss of generality, in Section 7, we show how modifying the setpoint can deal with the case where B is unknown.

Work by Mitra *et al* has shown that asymptotic analysis of product form queueing networks can be used to derive an optimal value of the setpoint [42, 43]. While their ideas are not directly applicable because of their assumptions of FCFS scheduling, Poisson cross traffic and an exponentially distributed packet service time distribution, to a first approximation, their results may be used to determine the choice of the optimal setpoint in the control system.

4.1.2. Assumptions Regarding Round Trip Time Delay

We assume that the propagation delay, R, is constant for a conversation. This is usually true, since the propagation delay is due to the speed of light in the fiber and hardware switching delays. These are fixed, except for rare rerouting.

We assume that the round trip time is large compared to the spacing between the acknowledgments. Hence, in the analysis, we treat the arrival of the packet pair as a single event, that measures both the round trip time and the bottleneck service rate.

Finally, we assume that the measured round trip time in epoch k, denoted by $RTT(k)$, is a good estimate for the round trip time in epoch $k+1$. The justification is that when the system is in equilibrium, the queue lengths are expected to be approximately the same in successive epochs. In any case, for wide area networks, the propagation delay will be much larger than the additional delay caused by a change in the queueing delay. Hence, to a first approximation, this change can be ignored.

4.2. Controller Design: Discrete Time Control

We initially restrict control actions to only once per round trip time (RTT) (this restriction is removed later). For the purpose of exposition, we divide time into *epochs* of length RTT (= propagation delay R + queueing delays) (Figure 2). This is done simply by transmitting a specially marked packet-pair, and when it returns, taking control action, and sending out another marked pair. Thus, the control action is taken at the end of every epoch.

Consider the situation at the end of the kth epoch. At this time we know $RTT(k)$, the round trip time in the kth epoch, and $S(k)$, the number of packets outstanding at that time. We also *predict* $\hat{\mu}_b(k+1)$, which is the estimator for the average service rate during the $(k+1)$th epoch. (If the service rate is 'bursty', then using a time average for μ may lead to problems. For example, if the average value for μ is large, but during the first part of the control cycle, the actual value is low, then the bottleneck buffers could overflow. In such cases, we can take control action with the arrival of every probe, as discussed in Section 4.3.)

Figure 2 shows the time diagram for the control. The vertical axis on the left is the source, and the axis on the right is the bottleneck. Each line between the axes represents a packet pair. Control epochs are marked for the source and the bottleneck. Note that the epochs at the bottleneck are time delayed with respect to the source. We use the convention that the end of the kth epoch is called 'time k', except that $n_b(k)$ refers to the number of packets in the bottleneck at the *beginning* of the kth epoch. Estimators are marked with a hat.

We now make a few observations regarding Figure 2. The distance *ab* is the RTT measured by the source (from the time the first packet in the pair is sent to the time the first ack is received). By an earlier assumption, the propagation delay for the $(k+1)$th special pair is the same as for the kth pair. Then $ab = cd$, and the length of epoch k at the source and at the bottleneck will be the same, and equal to $RTT(k)$.

At the time marked 'NOW', which is the end of the kth epoch, all the packets sent in epoch $k-1$ have been acknowledged. So, the only unacknowledged packets are those sent in the kth epoch itself, and this is the same as the number of outstanding packets $S(k)$. This can be approximated by the sending rate multiplied by the sending interval, $\lambda(k)\,RTT(k)$. So,

$$S(k) = \lambda(k)\,RTT(k) \tag{1}$$

The number of packets in the bottleneck at the beginning of the $(k+1)$th epoch is simply the number of

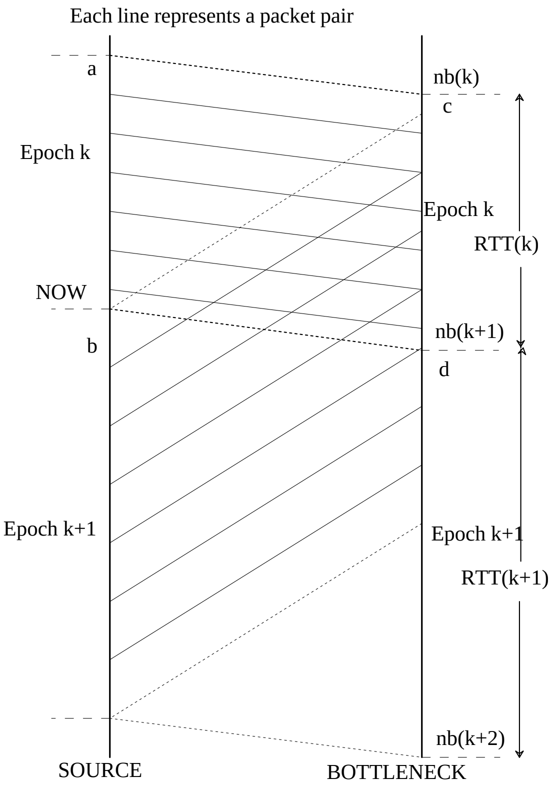

Figure 2: Time scale for control

packets at the beginning of the kth epoch added to the number that came in minus what was served in the kth epoch subject to the condition that n_b lies in [0, B]. Since $\lambda(k) RTT(k) = S(k)$ packets were sent in, and $\mu(k) RTT(k)$ packets were serviced in this interval, we have

$$n_b(k+1) = \begin{cases} \text{if } n_b(k) + \lambda(k) RTT(k) - RTT(k)\mu(k) < 0 \textit{ then } 0 \\ \text{if } n_b(k) + \lambda(k) RTT(k) - RTT(k)\mu(k) > B \textit{ then } B \\ \quad \textit{otherwise} \;\; n_b(k) + \lambda(k) RTT(k) - RTT(k)\mu(k) \end{cases}$$

The introduction of the MAX and MIN terms in the state equation makes the system nonlinear at the boundaries. However, note that if the setpoint is chosen to lie in the interior of the range [0,B], then the system is linear around the setpoint. Hence, for small deviations from the setpoint, we have

$$n_b(k+1) = n_b(k) + (\lambda(k) - \mu(k)) RTT(k) \tag{2}$$

Equations (1) and (2) are the fundamental equations in this analysis. They can be combined to give

$$n_b(k+1) = n_b(k) + S(k) - \mu(k) RTT(k) \tag{3}$$

Since $n_b(k+1)$ is determined by what we sent in the kth epoch, there is no way to control it. Instead, we control $n_b(k+2)$. We have

$$n_b(k+2) = n_b(k+1) + (\lambda(k+1) - \mu(k+1)) RTT(k+1) \tag{4}$$

From (3) and (4):

$$n_b(k+2) = n_b(k) + S(k) - \mu(k) RTT(k) + (\lambda(k+1) - \mu(k+1)) RTT(k+1) \tag{5}$$

The control should set this to $B/2$. So we set (5) to $B/2$, and obtain $\lambda(k+1)$.

$$B/2 = n_b(k) + S(k) - \mu(k) RTT(k) + (\lambda(k+1) - \mu(k+1)) RTT(k+1) \tag{6}$$

This gives $\lambda(k+1)$ as

$$\lambda(k+1) = \frac{1}{RTT(k+1)}[B/2 - n_b(k) - S(k) + \mu(k)RTT(k) + \mu(k+1)RTT(k+1)] \tag{7}$$

Replacing the values by their estimators (which will be derived later), we have

$$\lambda(k+1) = \frac{1}{\hat{RTT}(k+1)}[B/2 - \hat{n}_b(k) - S(k) + \hat{\mu}(k)RTT(k) + \hat{\mu}(k+1)\hat{RTT}(k+1)] \tag{8}$$

From an earlier assumption, we set $\hat{RTT}(k+1)$ to $RTT(k)$. This gives us:

$$\lambda(k+1) = \frac{1}{RTT(k)}[B/2 - \hat{n}_b(k) - S(k) + (\hat{\mu}(k) + \hat{\mu}(k+1))RTT(k)] \tag{9}$$

This is the control law. The control tries to get the buffer to $B/2$ at the end of the next epoch.

Note that the control law requires us to maintain two estimators: $\hat{\mu}_b(k)$ and $\hat{n}_b(k)$. The effectiveness of the control depends on the choice of the estimators. This is considered in Sections 4.4 and 4.5.

4.2.1. Stability Analysis for Discrete Time Control

For the stability analysis of the controlled system, $\lambda(k)$ is substituted into the state equation from the control law. We will assume that the estimators are perfect, and so replace the estimators in (9) with the true values. Since we know $\lambda(k+1)$, we use the state equation (2) one step ahead in time. This gives

$$n_b(k+2) = n_b(k+1) + (\lambda(k+1) - \mu(k+1))RTT(k+1) \tag{10}$$

Substitute (8) in (10) to find the state evolution of the controlled system.

$$n_b(k+2) = n_b(k+1) - \mu(k+1)RTT(k+1) + \frac{RTT(k+1)}{RTT(k)}[B/2 - n_b(k) - S(k) + (\mu(k) + \mu(k+1))RTT(k)]$$

By assumption, $RTT(k)$ is close to $RTT(k+1)$. So, moving back two steps in time,

$$n_b(k) = n_b(k-1) - \mu(k-1)RTT(k-1) + B/2 - n_b(k-2) - S(k-2) + (\mu(k-2) + \mu(k-1))RTT(k-2)$$

Taking the Z transform of both sizes, we get

$$n_b(z) = z^{-1}n_b(z) - z^{-2}\mu(z)RTT(z) + B/2 - z^{-2}n_b(z) - z^{-2}S(z) + 2z^{-4}\mu(z)RTT(z)$$

Considering n_b as the state variable, the characteristic equation is

$$z^{-2} - z^{-1} + 1 = 0$$

If the system is to be asymptotically stable, then the roots of the characteristic equation (the eigenvalues of the system), must lie inside the unit circle on the complex Z plane. Solving for z^{-1}, we get

$$z^{-1} = \frac{1 \pm \sqrt{3}\, i}{2}$$

Hence the eigenvalues lie on the unit circle, and the controlled system is *not* asymptotically stable.

However, we can place the pole of the characteristic equation so that the system is asymptotically stable. Consider the control law

$$\lambda(k+1) = \frac{\alpha}{RTT(k)}[B/2 - \hat{n}_b(k) - S(k) + (\hat{\mu}(k) + \hat{\mu}(k+1))RTT(k)]$$

This leads to a characteristic equation

$$\alpha z^{-2} - z^{-1} + 1 = 0$$

and roots

$$z^{-1} = \frac{1 \pm \sqrt{4\alpha - 1}\, i}{2\alpha}$$

The poles are symmetric about the real axis, so we need only ensure that

$$|z^{-1}| > 1 \;\; => \;\; \alpha < 1$$

This means that if $\alpha < 1$, the system is asymptotically stable. By the Separation Theorem, since the system and observer eigenvalues are distinct, this stability result holds independent of the choice of estimators.

The physical interpretation of α is simple: to reach $B/2$ at the end of the next epoch, the source should send exactly at the rate computed by (9). If it does so, the system may be unstable. Instead, it sends at a slightly lower rate, and this ensures that the system is asymptotically stable. Note that α is a constant that is independent of the system's dynamics and can be chosen in advance to be any desired value smaller than 1.0. The exact value chosen for α controls the rise time of the system, and for adequate responsiveness, it should not be too small. Simulations indicate that a value of 0.9 is a good compromise between responsiveness and instability. Similar studies are mentioned in [14].

4.3. Controller Design: Continuous Time Control

This section describes how the frequency of control can be increased by using information about the propagation delay. Note that $\hat{n}_b$, the estimate for the number of packets in the bottleneck queue, plays a critical role in the control system. The controller tracks changes in $\hat{n}_b(k)$, and so it is necessary that $\hat{n}_b(k)$ be a good estimator of n_b. $\hat{n}_b(k)$ can be made more accurate if additional information from the network is available. One such piece of information is the value of the propagation delay.

The round-trip time of a packet has delays due to three causes:

- the propagation delay from the speed of light and processing at switches and interfaces
- the queueing delay at each switch, because previous packets from that conversation have not yet been serviced
- the phase delay, introduced when the first packet from a previously inactive conversation waits for the server to finish service of packets from other conversations

The propagation delay depends on the geographical spread of the network, and for WANs, it can be of the order of a few tens of milliseconds. The phase delay is roughly the same magnitude as the time it takes to send one packet each from all the conversations sharing a server, the round time. The queueing delay is of the order of several round times, since each packet in the queue takes one round time to get service. For future high speed networks, we expect the propagation and queueing delays to be of roughly the same magnitude, and the phase delay to be one order of magnitude smaller. Thus, if queueing delays can be avoided, the measured round-trip time will be approximately the propagation delay of the conversation.

An easy way to avoid queueing delays is to measure the round-trip time for the first packet of the first packet-pair. Since this packet has no queueing delays, we can estimate the propagation delay of the conversation from this packet's measured round trip time. Call this propagation delay R.

The value of R is useful, since the number of packets in the bottleneck queue at the beginning of epoch $k+1$, $n_b(k+1)$, can be estimated by the number of packets being transmitted ('in the pipeline') subtracted from the number of unacknowledged packets at the beginning of the epoch, $S(k)$. That is,

$$\hat{n}_b(k+1) = S(k) - R\hat{\mu}_b(k)$$

Since S, R and $\hat{\mu}_b(k)$ are known, this gives us another way of determining $\hat{n}_b(k+1)$. This can be used to update $\hat{n}_b(k+1)$ as an alternative to equation (2). The advantage of this approach is that equation (2) is more susceptible to parameter drift. That is, successive errors in $\hat{n}_b(k+1)$ can add up, so that $\hat{n}_b(k+1)$ could differ substantially from n_b. In the new scheme, this risk is considerably reduced: the only systematic error that could be made is in μ. There is another advantage to this approach: it enables control actions to be taken at the arrival of every packet-pair, instead of once per RTT. Thus, the controller can react at the earliest possible moment to changes in the system.

In the system described thus far, we limited ourselves to once per RTT control because this enables the simple relationship between $S(k)$ and $\lambda(k)$ given by equation (1). If control actions are taken faster than once per RTT, then the epoch size is smaller, and the relationship is no longer true. The new relationship is much more complicated, and it is easily shown that the state and input vectors must expand to include time delayed values of μ, λ and n_b. The faster that control actions are required, the larger the state vector, and this complicates both the analysis and the control. In contrast, with information about the propagation delay R, control can be done as quickly as once every packet-pair with no change to the length of the state vector.

When control is done once every probe, it is easier to work in continuous time. We also make the fluid approximation [1], so packet boundaries are ignored, and the data flow is like that of a fluid in a hydraulic system. This approximation is commonly used [3, 6, 40, 55], and both analysis [43] and simulations show that the approximation is a close one, particularly when the bandwidth-delay product is large.

Let us assume that the input rate λ is held fixed for some duration J. Then,

$$n_b(t+J) = n_b(t) + \lambda(t)J - \mu(t)J \tag{11}$$

where μ is the average service rate in the time interval [t, t+J], and n_b is assumed to lie in the linear region of the space. Also, note that the amount of information in the bottleneck buffer can be estimated by the difference between the amount of unacknowledged data and the amount of data in flight:

$$n_b(t) = S(t) - R\mu(t) \tag{12}$$

The control goal is to have $n_b(t+J)$ be the setpoint value $B/2$. Hence,

$$n_b(t+J) = n_b(t) + \lambda(t)J - \mu(t)J = B/2 \tag{13}$$

So,

$$\lambda(t) = \frac{B/2 - S(t) + R\hat{\mu}(t) + J\hat{\mu}(t)}{J} \tag{14}$$

which is the control law. The stability of the system is easily determined. Note that $\dot{n}_b(t)$ is given by

$$\dot{n}_b(t) = \lim_{\delta \to 0} \frac{n_b(t+\delta) - n_b(t)}{\delta} = \lambda(t) - \mu(t) \tag{15}$$

From equation (13),

$$\dot{n}_b = \frac{B/2 - n_b(t)}{J} \tag{16}$$

If we define the state of the system by

$$x = n_b(t) - B/2 \tag{17}$$

then, the equilibrium point is given by

$$x = 0 \tag{18}$$

and the state equation is

$$\dot{x} = \frac{-x}{J} \tag{19}$$

Clearly, the eigenvalue of the system is -1/J, and since J is positive, the system is both Lyapunov stable and asymptotically stable. In this system, J is the pole placement parameter, and plays exactly the same role as α in the discrete time system. When J is close 0, the eigenvalue of the system is close to $-\infty$ and the system will reach the equilibrium point rapidly. Larger values of J will cause the system to move to the equilibrium point more slowly. An intuitively satisfying choice of J is one round trip time, and this is easily estimated as $R + S(k)\mu(t)$. In practice, the values of R and $S(k)$ are known, and $\mu(t)$ is estimated by $\hat{\mu}$.

4.3.1. Fuzzy Prediction

Having derived the control law, and proved its stability, we now need to determine stable estimators for the system state. In earlier work, we studied the Kalman estimator [32]. However, this estimator requires *a priori* knowledge of the variances of the system and observation noise. Since these values are hard to determine, we turn to a novel technique for state estimation, fuzzy prediction.

The basis of fuzzy prediction is the exponential averaging predictor given by:

$$\hat{\theta}_{k+1} = \alpha\hat{\theta}_k + (1-\alpha)\tilde{\theta}_k$$

The predictor is controlled by a parameter α, where α is the weight given to past history. The larger it is, the more weight past history has in relation to the last observation. The method is called exponential averaging, since the predictor is the discrete convolution of the observed sequence with an exponential curve with a time constant α

$$\hat{\theta}_k = \sum_{i=0}^{k-1} (1-\alpha)\tilde{\theta}_i \alpha^{k-i-1} + \alpha^k \hat{\theta}_0$$

The exponential averaging technique is robust, and so it has been used in a number of applications. However, a major problem with the exponential averaging predictor is in the choice of α. While in principle, it can be determined by knowledge of the system and observation noise variances, in practice, the variances are unknown. It would be useful to automatically determine a 'good' value of α, and to be able to change this value on-line if the system behavior changes. Our approach uses fuzzy control to effect this tuning [63, 67].

Fuzzy exponential averaging uses the assumption that a system can be thought of as belonging to a spectrum of behavior that ranges from 'steady' to 'noisy'. In a 'steady' system, the sequence $\{\theta_k\}$ is approximately constant, so that $\{\tilde{\theta}_k\}$ is affected mainly by observation noise. Then, α should be large, so that the past history is given more weight, and transient changes in $\tilde{\theta}$ are ignored.

In contrast, if the system is 'noisy', $\{\theta_k\}$ itself could vary considerably, and $\tilde{\theta}$ reflects changes both in θ_k and the observation noise. By choosing a lower value of α, the observer quickly tracks changes in θ_k, while ignoring past history which only provides old information.

While the choice of α in the extremal cases is simple, the choice for intermediate values along the spectrum is hard to make. We use a fuzzy controller to determine a value of α that gracefully responds to changes in system behavior. Thus, if the system moves along the noise spectrum, α adapts to the change, allowing us to obtain a good estimate of θ_k at all times. Moreover, if the observer does not know α *a priori*, the predictor automatically determines an appropriate value.

Since α is linked to the 'noise' in the system, how can the amount of 'noise' in the system be determined? Assume, for the moment, that the variance in ω is an order of magnitude larger than the variance in ξ (this assumption is removed later in this section). Given this assumption, if a system is 'steady', the exponential averaging predictor will usually be accurate, and prediction errors will be small. In this situation, α should be large. In contrast, if the system is 'noisy', then the exponential averaging predictor will have a large estimation error. This is because when the system noise is large, past history cannot predict the future. So, no matter what the value of α, it will usually have a large error. In that case, it is best to give little weight to past history by choosing a small value of α, so that the observer can track the changes in the system.

To summarize, the observation is that if the predictor error is large, then α should be small, and vice versa. Treating 'small' and 'large' as fuzzy linguistic variables [62], this is the basis for a fuzzy controller for the estimation of α.

The controller implements three fuzzy laws:

If proportional error is low, then α *is high*
If proportional error is medium, then α *is medium*
If proportional error is high, then α *is low*

The linguistic variables 'low', 'medium' and 'high' for α and proportional error are defined in the usual way in Figure 3.
The input to the fuzzy controller is a value of proportional error, and it outputs α in three steps. First, the proportional error value is mapped to a membership in each of the fuzzy sets 'low', 'medium', and 'high' using the definition in Figure 3. Then, the control rules are used to determine the applicability of each outcome to the resultant control. Finally, the fuzzy set expressing the control is defuzzified using the centroid defuzzifier.

The error $|\tilde{\theta} - \hat{\theta}|$ is processed in two steps before it is input to the fuzzy system. First, it is converted to a proportional value, error $= \dfrac{|\tilde{\theta}_k - \hat{\theta}_k|}{\tilde{\theta}_k}$. Second, it is not a good idea to use the absolute error value directly, since spikes in $\tilde{\theta}_k$ can cause the error to be large, so that α drops to 0, and all past history is lost.

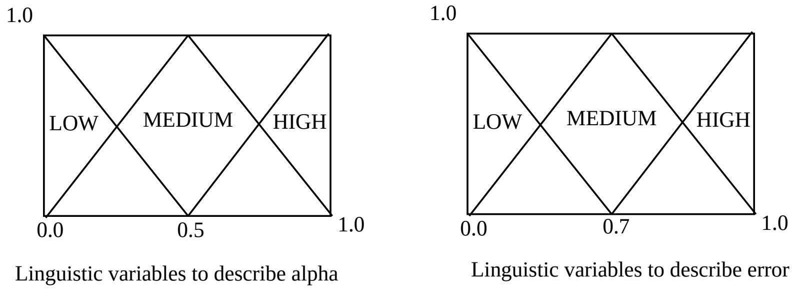

Figure 3: Definition of linguistic variables

So, the absolute error is smoothed using another exponential averager. The constant for this averager, β, is obtained from another fuzzy controller that links the change in error to the value of β. The idea is that if the change in error is large, then β should be large, so that spikes are ignored. Otherwise, β should be small. β and change in error are defined by the same linguistic variables, 'low' and 'high', and these are defined exactly like the corresponding variables for α. With these changes, the assumption that the variance in the observation noise is small can now be removed. The resulting system is shown in Figure 4. Further details of the prediction system and a performance analysis can be found in Reference [37].

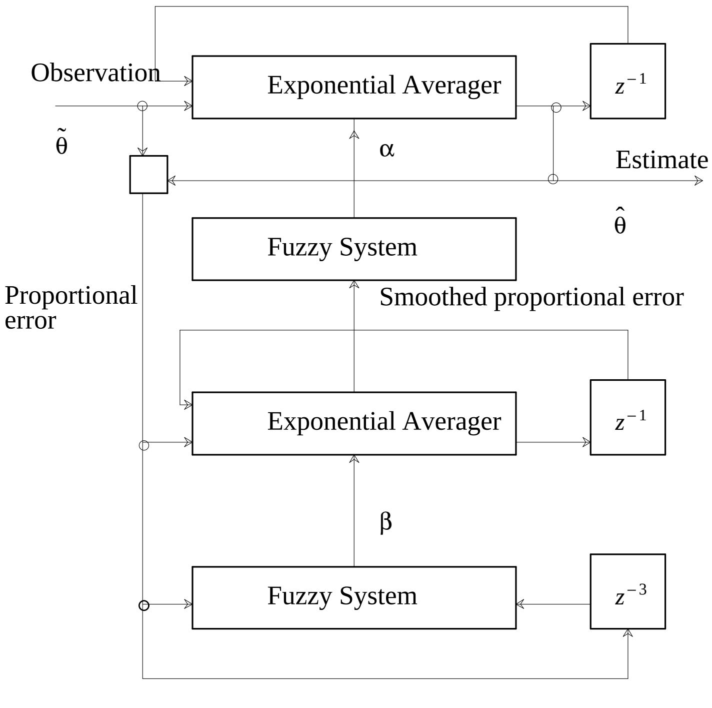

Figure 4: Fuzzy prediction system

5. Startup

Thus far we have discussed the situation where the rate control mechanism has had some time to collect past history and has a steady stream of user data to do continuous rate probing. Unfortunately, there are at least three situations where this situation does not hold. First, the transfer length may be too short for the source to accumulate enough history to get a good estimator for μ. Second, the source may be intermittent, so that there are long idle periods, during which the stored state information becomes aged and useless. Finally, during the startup phase of a conversation, there is no information about the network. All the three cases share a common problem, that is, the source does not have enough state about the network to make an informed decision. Thus, the control law described in the earlier sections is not directly applicable. We call this the startup problem, and describe several solutions to the problem in this section.

Before we begin, note that having a good startup algorithm is an essential part of flow control. If the startup is too aggressive, then on startup, every conversation will overflow its buffers, which is undesirable. On the other hand, a conservative startup can lead to poor performance for short transfers. Since a large class of applications, such as Remote Procedure Calls, consist of short transfers compared to the round trip bandwidth delay product, having a good startup scheme is crucial.

Several startup schemes have been described in the literature. The DECbit scheme is to open the window linearly till the operating point is reached [48]. This is a poor choice when the bandwidth delay product is large. For example, when the operating window is 200 packets, this scheme will take 200 round trip times to reach that size. A faster way to reach the operating point is the exponential + linear scheme proposed by Jacobson and Karels [26]. However, the choice of when to change phase from exponential to linear increase is rather *ad hoc*. Finally, Kanakia and Mishra have proposed a linear increase in the sending rate till the operating point is reached (i.e a constant acceleration in the sending rate) [29]. This scheme performs better than the DECbit scheme, but the choice of the acceleration is crucial, and not addressed.

Our proposal is to use an adaptive exponential rise to the nominal operating point. Given a current nominal operating point, the rate control will exponentially increase the sending rate to this point. However, as new information about the operating point arrives, the asymptote of the exponential rise is adjusted on the fly (Figure 5).

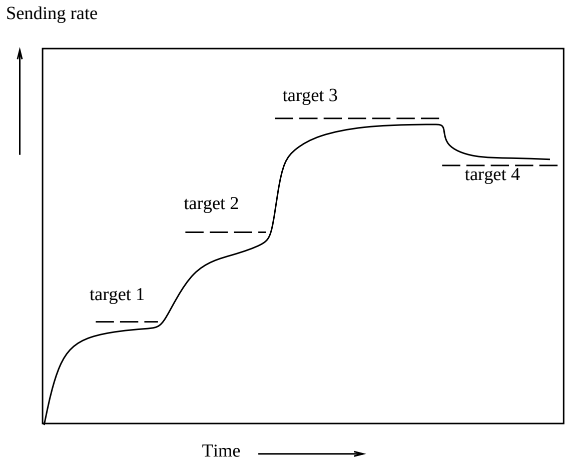

Figure 5: Adaptive Exponential Startup

In the figure, starting at time 0, the sending rate exponentially rises to the first target operating point. However, soon after reaching this asymptote, fresh information about the next target arrives, so that the sending rate moves exponentially to the new target. The sending is quickly adjust to information as it arrives. The idea is that the information obtained at startup is not very accurate, so we do not want to rise to it immediately. Nor do we want to be overly conservative. By choosing an exponential rise, there is some room to maneuver, without giving poor performance to short transfers.

Each target operating point is chosen by applying the control law to the current estimator for μ. At startup, a source sends a packet pair probe and waiting for the first pair of acks to arrive. This gives enough information for the control law to choose the first target operating point. During the second round trip time, more packet pairs are sent and as the estimators get better, the target operating points can change. The source does an adaptive exponential rise to the targets as and when they change. Note that there is a one round trip time delay at startup. This can be avoided if the network administrator can guarantee that each new connection is guaranteed some nominal bandwidth. Then, during the first round trip time, the connection can send at this bandwidth, revising its estimate for actual available bandwidth at the end of the first round trip time.

It is important to determine when to end the startup phase. In other words, we need to know when that the operating point has been reached. This information is available by looking at the estimator for the bottleneck buffer size $\hat{n}_b$. When the sending rate is below the operating point, $\hat{n}_b(k+1)$ will be close to zero, since the service rate will exceed the sending rate. However, as we approach the operating point, and the buffer setpoint is reached, $\hat{n}_b(k+1)$ will rise above zero, and reach the chosen setpoint. Thus, a simple way to know that the operating point has been reached is to check for $\hat{n}_b(k+1) > 0$. In practice, to allow for measurement errors, we actually check for $\hat{n}_b(k+1) \geq 2$.

Having presented the adaptive exponential rise scheme, we now discuss the solutions to the three problems raised at the beginning of this section. For short transfers and startup for long transfers, we use the adaptive exponential rise scheme until $\hat{n}_b(k+1) \geq 2$. (With short transfers, the transfer may complete before this condition is reached.) For the problem of intermittent transfers, note that the end of a tranmission burst causes the value of $\hat{n}_b$ to drop to zero. Thus, when a new burst starts, the flow control automatically goes into slow-start. Network state information is automatically updated during this startup.

The adaptive exponential rise technique is also useful whenever the state estimators are suspected to be inaccurate. Such a situation arises when the number of active sources at a bottleneck suddenly decreases. Consider a scenario where three sources share a line (Figure 6a). Note that each source sends a pair of packets, and the output is perfectly interleaved. Thus, each source correctly estimates its capacity as 1/3 of the line capacity. Now, if one of the three terminates (Figure 6b), for the next round trip time the other two send data slower than their available capacity, draining their bottleneck queue. If this leads to a zero length queue, then it is possible that pairs arriving from the same conversation would be served back to back before the arrival of pairs from the other conversation. For example, the pairs from source 'a' are served back-to-back since the server is otherwise idle. The same occurs for source 'c'. Both conversations incorrectly estimate their share of the line at 100%, and in the next round trip time, would send at too high a rate.

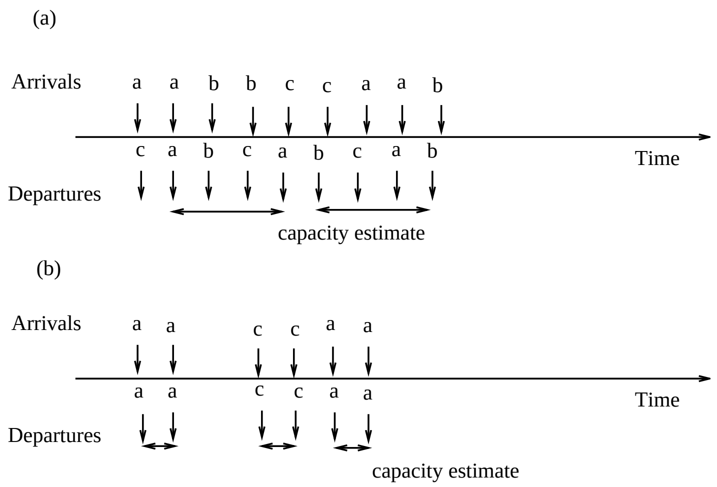

Figure 6: If a source stops transmission, the others get wrong rate estimates

The real problem here is that the sources receive poor estimates of bottleneck capacity because their bottleneck buffers are empty. If the buffers were non-empty, then even when conversation 'b' terminated, packets from 'c' would be available to interfere with packets from 'a' and thus correctly inform 'a' that its share of capacity is 50%. If the endpoints notice that their $\hat{n}_b(k+1)$ is small, and act cautiously on the estimators received when this situation holds, this problem can be avoided.. Thus, the source should check for the condition $\hat{n}_b(k+1) < 2$ at all times (not just during startup), and do an adaptive exponential rise if this condition holds. Simulations show that this is solution works well in practice.

6. Timeout and Retransmission Strategy

Timeout and retransmission strategy is usually considered to be a part of error control rather than flow control. However, if timers are too small there can be numerous retransmissions, which can lead to network congestion. A poor choice in the other direction will lead to long pauses, wasting the available bandwidth [64]. Thus, the choice of timers and of which packets to retransmit is intimately related to flow and congestion control.

Three considerations guided the design of the timeout and retransmission strategy for packet-pair. First, we believe that timers should be a mechanism of last resort. In the normal case, losses should be detected without using timers. However, if timers are needed, then the choice of the timeout value should be chosen intelligently, based on current network state. Second, the sender should try to continuously keep sending data in order to keep probing the network state. That is, it is desirable to keep the pipeline as full as possible. Finally, loss detection and correction should be made orthogonal to flow control.

To attain the last objective, each transport layer connection maintains a transmission queue that buffers both incoming user data and data to be retransmitted. The transmission queue is partitioned into a high priority zone at the head, and a low priority zone at the tail of the queue. Data from the application layer is placed at the tail of the low priority zone, and the packets awaiting retransmission are added to the tail of the high priority zone of the queue. The rate control process removes data from this queue at a rate specified by the control law or startup procedure (Figure 7). The idea is that since retransmissions and data share the same queue, it is not possible to violate the sending rate due to retransmissions. The transmission queue thus effective decouples the rate and error control processes.

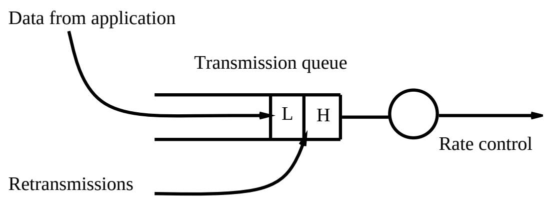

Figure 7: Transmission queue

To detect losses without using timers, we monitor the sequence numbers carried by the acknowledgments. We assume that, as in TCP, this sequence number is that of the last in-sequence packet seen so far, and that every packet is acknowledged. In addition, every acknowledgement carries the offset between the last in-sequence packet seen, and the sequence number of the packet that generated the acknowledgment. For example, if the receiver receives sequence numbers 1,2,3,4,6,7,8... the acknowledgements will be (1,0), (2,0), (3,0), (4,0), (4,2), (4,3), (4,4)... where each tuple is the sequence number of the acknowledgment, and the corresponding offset. A non-zero offset indicates that the packet with the sequence number one larger than the sequence number must have been lost (or was received out of order). Assuming that most packets are delivered in order (which is true for virtual circuit oriented networks, such as ATM), the sender should retransmit the packet with that sequence number. For the example above, the sender presumes that packet 5 must have been lost and will retransmit it. This idea of retransmitting packets before they time out is called fast retransmit, and is due to Jacobson [26]. Fast retransmits are a good indication of receiver state, since even if one or more acknowledgments are lost, the information is repeated, and even one duplicate acknowledgment sequence number is enough to inform the sender about the loss of a data packet.

In contrast to Jacobson's approach, where three duplicate acks automatically trigger retransmission of the flow control window, we use the offset information to do intelligent retransmission. When an ack with a non-zero offset is received, the sender notes that the packet with sequence number (ack sequence number + offset) has reached safely. It then retransmits every packet in the range [last acknowledgement sequence number, ack sequence number + offset] that has not been retransmitted already, and has not been received correctly (this would have been detected earlier). Continuing with the example, when the sender receives (4,2), it will retransmit 5 and note that 6 was correctly received. When (4,3) is received, since 5 has been retransmitted, and 6 has been received, no retransmissions occur. Thus, with a single loss, a single retransmission will occur. If a large chunk of outstanding packets are lost, they will all be retransmitted. The scheme guarantees that no packet that has been correctly received and acknowledged is ever retransmitted. In this sense, it is optimal.

If the retransmitted packet is also lost, then the scheme described above will fail. This is handled by two other schemes called soft timeout and timeout respectively. In the soft timeout scheme, every two round trip times, the source checks if any progress has been made in the last consecutive acknowledgement received (round trip times are measured without using timers, as described in the discussion of buffer setpoint probing in Section 7). If no progress has been made, then the packet with sequence number (last_ack + 1) is retransmitted. Thus, in the usual case, the hole in the transmission window is corrected without having a timer. Of course, if more than one packet is lost in the window, a timeout is needed.

In order to minimize the OS requirements of an implementation, timeouts are done using a single shared timer, instead of a per-packet timer. The timer is re-initialized at the start of every packet transmission. On a timeout, the entire flow control window is put in the retransmission queue, except for packets that have been received correctly. Thus, if there are multiple losses in a round trip time, these are automatically retransmitted. With this scheme, the sender keeps the pipeline full the extent possible. Note that the retransmission scheme combines the robustness and low overhead of a go-back-n protocol with the transmission efficiency of a selective retransmission scheme. Also note that the actions on a duplicate ack and a timeout are identical, except that the duplicate ack scheme will not retransmit a packet that has already been retransmitted, but the timeout scheme will.

The timeout value must be well chosen to deal with the case where a whole window as well as its acknowledgments are lost (for example, on a mobile host that has moved from one cell to another) or when a fast retransmission is lost. We expect the round trip time to be $R + S(k)\hat{\mu}_b(k)$. Thus, we set the timer to be 1.5 ($R + S(k)\hat{\mu}_b(k)$), in the expectation that if an ack is not heard from in this time, something is wrong. Since we have good estimators for R, $S(k)$ and $\hat{\mu}_b(k)$, this enables us to set timeouts fairly accurately. If necessary, the multiplier can be tuned to achieve the best results for a particular network.

The control law assumes that a correct count of the number of outstanding packets, $S(k)$, is available. It is important to know $S(k)$ accurately, since the estimate for the number of packets in the bottleneck directly depends on this value. If there are no packet losses, then $S(k)$ can simply be incremented on every packet transmission, and decremented with every ack. However, if a packet is lost, then the increment to $S(k)$ is made on transmission, but the corresponding decrement is not made. Thus, if $S(k)$ is not corrected to deal with lost packets, it will slowly increase with time, causing a systematic error in the rate control. To deal with this, the sender decrements $S(k)$ whenever a packet is retransmitted either from a fast retransmission or a timeout. This correctly accounts for the lost packet, and prevents drift in $S(k)$.

7. Buffer Management Strategy

In Section 4.1.1 we assumed that each intermediate point would reserve B buffers per conversation. If that is indeed feasible, then the preceding treatment is sufficient. However, we expect there to be many networks where intermediate queueing points do not reserve buffers per conversation. In this case, the endpoint has two strategies. One is to minimize losses by choosing a small setpoint. As discussed in Section 4.1.1, this will lead to a lower effective throughput, but the loss rate would also be lower. This would be suitable for risk-averse applications such as remote procedure calls.

The other strategy is to choose the largest setpoint that can be supported by the network without causing excessive packet losses for the conversation. This can be done by choosing a small setpoint, and increasing the setpoint additively once every few round trip times till a loss occurs, which would trigger a fast retransmit and a multiplicative decrease in the setpoint. By this dynamic probing of the setpoint, a risk-taking client can choose a setpoint that is the largest that can be supported at any given time. This maximizes the available bandwidth at the risk of incurring packet losses and additional queueing delay. In our work, we have found that increasing the setpoint by 0.1 every four round trip times, and a multiplicative decrease factor of 0.6 is effective in most circumstances. Note that this additive-increase-multiplicative-decrease algorithm is modeled on the DECbit scheme, but is rather different in its aim. In the DECbit scheme, the window size is increased and decreased to modify the operating rate. Here, we choose the sending rate based on the packet-pair information. The buffer setpoint is modified by risk-taking applications only to maximize the usage of buffers at the bottleneck.

8. Implementation

This section presents details on the implementation of packet-pair flow control. Feedback flow control algorithms are typically implemented at the transport layer of the protocol stack [54]. If a data source is not trusted, or incapable of flow control, this functionality could also be implemented at the Network Interface Unit, which is interposed between a traffic source and a public B-ISDN network. In either case, we assume the system to be able to support per-connection state, give access to a real-time clock, and provide at least one timer. We present the state diagram for the protocol, describing the actions at each state.

We describe only the sending side of the packet-pair protocol. Each packet is acknowledged by the receiver, and the sequence number of the acknowledgment is the last in-sequence packet seen by the receiver so far. A connection is assumed to have a transmission queue where it buffers packets awaiting transmission (Figure 7). A flow chart describing the implementation of the protocol is presented in Figure 8.

State diagram

The system is usually in blocked state, waiting for an input. This can be one of ACK, TICK, INT, TIMEOUT and USER_INPUT. When a packet containing an acknowledgement (ACK) arrives, and has a non-zero sequence number offset, we know that at least one packet in the current send window has been lost. The sender first notes that sequence number (last_ack + offset) has been correctly received, It then scans the current send window, and places all packets in the range [last_ack +1, last_ack + offset] that have not been retransmitted already at the end of the high priority portion of the transmission queue, to be retransmitted eventually. If the ack signals the end of two round trip times in which no progress has been made, then the packet with sequence number (last_ack + 1) is retransmitted (soft timeout).

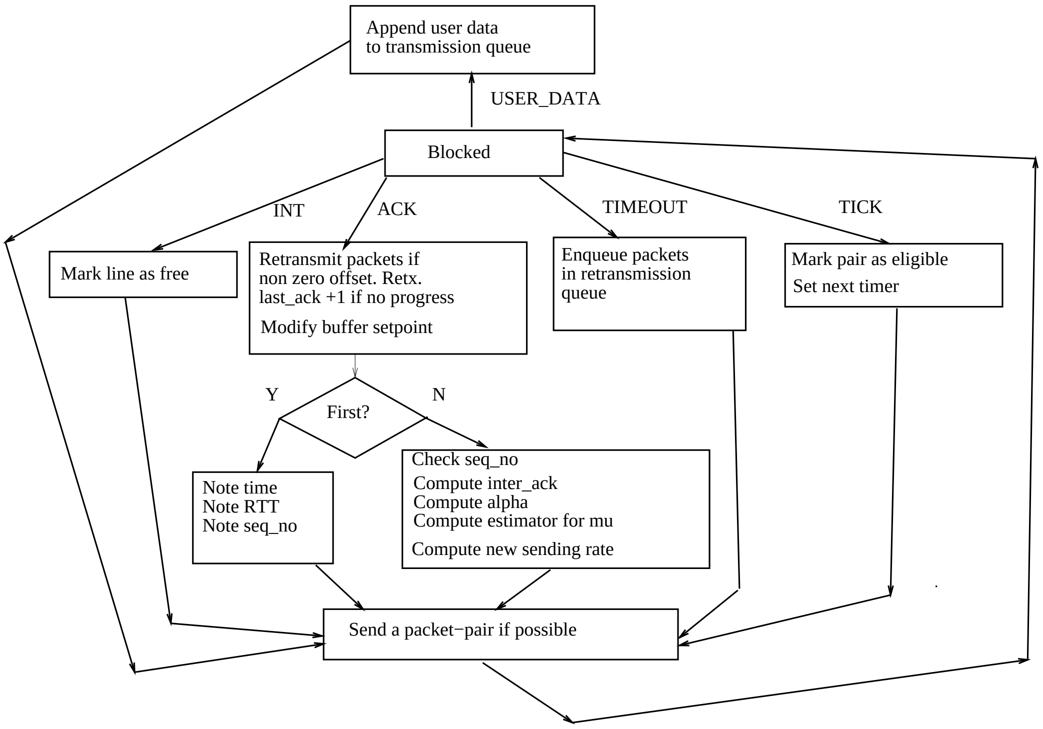

Figure 8: Finite state machine description of the protocol

If the ACK is the first of a pair (this is a field in the transport layer header), then the current time is noted. By comparing the transmission time of this packet to the current time, the round trip time is computed and stored in a state variable. The sequence number of this packet and the current time are also stored. The arrival of the ack reduces the count of outstanding packets by one.

If the ack's header claims that it is the second of a pair, we validate its sequence number by testing if it is one larger than the sequence number of the last ack seen. If so, this is a valid ack pair, and the inter-ack time is the current measurement of the bottleneck service time. This is compared with the current estimate of the service time, and the proportional error is fed into the fuzzy controller to get a new value of α for the exponential averager (Section 4.5). Using this α and the new observation of s_b, the new value for μ and $\hat{n}_b$ are computed. These are plugged into the continuous time control law (Equation 14), to obtain the new sending rate. This rate is stored in a state variable, and is used by the per-connection timer then next time it is loaded. Note that if multiple updates to the sending rate occur before the current rate timer expires, the latest value of the sending rate is automatically used.

To dynamically modify the setpoint, the source maintains two state variables, `rtt_count` which counts round trip times, and `rtt_seq`, which stores a sequence number. `rtt_seq` is initially 0. When an ack arrives, if the sequence number is greater than `rtt_seq`, then `rtt_seq` is set to the sequence number of the next packet to be sent, and `rtt_count` is incremented. Since the ack for this packet will arrive in approximately one round trip time, this method gives us a cheap but approximate round trip time counter. When `rtt_count` reaches a chosen value, the setpoint is increased, and the counter reset to zero. If there is a packet loss, the setpoint is multiplicatively decreased. Thus, the dynamic setpoint probing is quite simple to implement. By testing to see if the ack has a sequence number greater than or equal to `rtt_seq`, the scheme is tolerant of packet loss. The errors introduced by the fact that the next packet is not sent immediately are small, and tolerable for our purpose.

The adaptive exponential rise uses a state variable called the `mean_send_rate`, which is an exponentially averaged value of the actual sending rate. The control variable for the exponential averaging is supplied extraneously and the system is quite insensitive to this choice. We used a value of 0.75. The adaptive rise is implemented as an exponential increase from the current `mean_send_rate` to the target operating point specified by the control law. Specifically, if the current target is T, the current `mean_send_rate` is m, then the next sending rate is chosen to be $INC\_FACTOR*(T - m)$, and m is updated by the exponential averager. We used an INC_FACTOR of 0.2. To deal with startup, if the current value of $\hat{n}_b(k+1)$ is less than 2, then the sending rate is chosen to be value computed above, else it is the value computed by the control law (i.e. the target operating point itself). At startup, the `mean_send_rate` is set to 0, and $\hat{n}_b(k+1) < 2$ holds, so that the system automatically does an adaptive rise.

In Section 4.3 we mentioned that J is the pole placement parameter and can be used to control the speed with which the system reacts to changes. A large value of J will lead to slow reaction, and a small J leads to a quick reaction, with possible oscillations. A good value for J is $\hat{RTT}$, though it can be chosen arbitrarily. We use two values of J in our implementation. If $\hat{n}_b(k+1)$ is above the setpoint, then the buffer is overfull, and we would like to react to it quickly in order to prevent packet loss. If $\hat{n}_b(k+1)$ is below the setpoint, we would like to use a less aggressive reaction rate. This is achieved by choosing two constants $ATTACK\_UP$ and $ATTACK\_DOWN$ such that if $\hat{n}_b(k+1) \le B/2$ then $J = ATTACK\_UP*\hat{RTT}$ else $J = ATTACK\_DOWN*\hat{RTT}$. We found $ATTACK\_UP = 1.5$ and $ATTACK\_DOWN = 0.8$ works well for a variety of simulated situations. When a TIMEOUT event occurs, all currently unacknowledged packets are placed in the transmission queue. This is similar to the go-back-n policy, except that some or all of these packets can be removed from the queue if acknowledgements are received before the queue can be emptied.

When an INT event occurs, this indicates that the output line is free, and a state variable is set to note this. When a USER_INPUT event occurs, the users data is appended to the tail of the transmission queue. If the transmission queue is full, the user process can be optionally be put to sleep. When a TICK event occurs, this indicates that the rate timer has expired, and a state variable is modified to indicate that it is valid to send a packet pair. The TICK timer is reloaded with a value computed using the control law by the latest ACK pair.

At the end of event processing, the source checks to see if it is possible to send another packet pair. This is true if the rate-timer has enough credits, and the output trunk is free, and either the receiver's flow control window is not full, or there has been a timeout or duplicate retransmission event. If all these conditions are true, two packets are removed from the head of the transmission queue and a packet pair is sent. As each packet is sent, the TIMEOUT timer is reloaded with a value computed from the current estimate of the propagation and queueing delays, and the number of outstanding packets is increased by one. If any of these conditions fail, then the source falls out of the test into the blocked state.

Dealing with Singletons

In the discussion above, we have implicitly assumed that each user write has enough data for an integral number of packet-pairs. This is because in the 'test' state, if there is only one packet's worth of data to send and the user sends no more data, then this packet will never be transmitted, which is an error. Thus, if a user calls the transport layer with, say, only 1 byte of data, (such as for a 'telnet' application), then the transport layer has to somehow deal with this singleton packet.

Clearly, since the user may hand the transport layer only one byte, the transport layer cannot assume that it can always fragment user data and send out this data as a pair. Thus, an alternate solution would be to accompany small packets with a dummy packet, whose only purpose would be to provide the pair to probe the bottleneck state. However, this is inefficient.

An alternate solution, which we have adopted, is to allow some packets to be singletons. These packets do not contribute to state probing. When the transport layer receives a packet, it starts off a If the timer goes off, and the packet that led to the timer being set is still in the transmission queue, it is transmitted as a singleton. This saves transmission efficiency at the cost of an extra timer and some user delay for singleton packets. If this is unacceptable for some applications, the timer can be set to zero, so that all singletons would be sent out right away. In our implementation, we chose the singleton timer (based on anticipated application sensitivity to delay) to be 100 ms.

Implementation Details

Table 1 shows the state variables used in the implementation, their meanings, and initial values. Table 2 shows the control parameters, their meanings, and their suggested values.

This concludes our description of the packet-pair scheme. We now evaluate its effectiveness by means of simulations.

9. Simulation Results

In this section, we present simulation results to measure the performance of packet-pair in a variety of situations. We start with simple scenarios where we explore the dynamics of the system, then move on to more complicated scenarios where we test the limits of packet-pair control.

The simulation scenarios have some network elements in common. We study the behavior a source sending data to a sink where the source and sink are separated by a WAN link (Figure 9). Without loss of generality, we assume that resource contention occurs at the destination LAN. The data packet size is 500 bytes (at the transport layer), and the ack packet is assumed to be 40 bytes long. These correspond to mean packet sizes observed on the Internet [10]. The source LAN speed is assumed to be the same as the WAN link speed, and the destination LAN speed is assumed to be one-tenth of the WAN link speed. The line speeds and propagation delays are chosen so that the bandwidth delay product is 100 packets and the round trip time is 100 time units (TUs). The bottleneck router has a buffer capacity of 100 packets. Since no congestion or queueing happens at the first router, the abstracted network is shown in Figure 10.

The pipeline depth of 100 packets and a round trip time delay of 100 TUs model a variety of bandwidth delay products. Table 3 shows sample LANs, MANs and WANs to which our results would be directly applicable. In this table, bandwidth refers to the sustained WAN bandwidth that would be available to a network interface unit (or single end system). Note that both high speed LANs as well as medium speed WANs fall in the same bandwidth delay regime, and can be modeled using similar parameters. We also show sample parameters for ATM networks where flow control is done on an ATM cell (instead of AAL5 frame) basis. Given the current economics of LAN interconnection (45 Mbps connections cost about 1000 dollars per mile per month), we feel that the simulation parameters are fairly representative.

Variable	Initial value	Meaning
seq_no	0	sequence number counter
last_sent	-1	highest sequence number sent so far
num_outstanding	0	number of packets outstanding
time_of_last_ack	-1	time when last ack recvd
timeout	2s	timeout value
alpha	1.0	exp. av. const for rate estimate
nbhat	0	estimate of number of packets in bottleneck
se	0	bottleneck rate estimator
re	0	estimator of RTT
reclean	0	clean estimate of propagation delay
send_rate	0	computed sending rate
mean_send_rate	0.0	smoothed sending tate
inter_ack	0	current inter-ack time value
last_ack	-1	last ack seen so far
first_of_pair	-1	seq number of first ack of pair
num_dup_acks	0	number of duplicate acks seen so far
packets_sent	0	packets sent so far
total_dup_acks	0	number of packets retransmitted so far from dupacks
B	B_INITIAL	current buffer setpoint
rtt_count	0	how many RTTs have gone by
rtt_seq	0	an ack with this sequence number indicates one RTT
num_retx[WINDOW_SIZE]	0	number of retransmissions for this sequence no.
recvd_ok[WINDOW_SIZE]	0	1 if packet with that seq. no. was correctly recvd.

Table 1: State variables and their meaning

Variable	Suggested value	Meaning
ATTACK_UP	1.5	attack rate if bottleneck underfull
ATTACK_DOWN	0.8	attack rate if bottleneck overfull
TIMEOUT_MULTIPLIER	1.5	this times RTT estimate is timeout value
INC_FACTOR	0.2	factor controlling send rate increase
MS_ALPHA	0.75	exponential averaging constant for mean_send_rate
B_DEC_FACTOR	0.75	multiplicative decrease factor for setpoint
B_ADD_INC_FACTOR	2.0	additive increase constant for setpoint
B_INITIAL	5	initial value of setpoint
B_MIN	2	smallest value of setpoint
B_COUNT	2	how many RTTs to wait before changing setpoint

Table 2: Control parameters

Network type	Packet size	Bandwidth	Delay	Buffer size
LAN	500 bytes	400 Mbps	1 ms	50 KBytes
MAN	500 bytes	40 Mbps	10 ms	50 Kbytes
WAN	500 bytes	6.7 Mbps	60 ms	50 Kbytes
ATM LAN	53 bytes	42 Mbps	1ms	5.3 Kbytes
ATM MAN	53 bytes	4.2 Mbps	10 ms	5.3 Kbytes
ATM WAN	53 bytes	706 Kbps	60 ms	5.3 Kbytes

Table 3: Simulation bandwidth delay regime

9.1. Base case dynamics

The simplest possible dynamics are when a single source sends packets through a single bottleneck which has adequate buffers. This is the best possible case for the flow control algorithm. We use this case to study the dynamics of 'free-running' flow control.

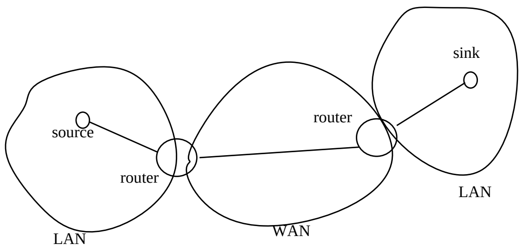

Figure 9: Simulation scenario

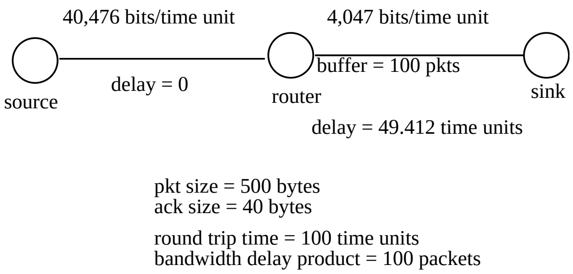

Figure 10: Abstracted scenario

In the first test we study the behavior of a risk-averse source by turning off setpoint probing and observing the behavior of the flow control algorithm in isolation. Figure 11 shows the setpoint, the queue length at the bottleneck buffer, its estimated value, and the number of outstanding packets versus simulation time measured in round trip times (RTTs). During startup, the bottleneck queue length is close to zero. As the sending rate matches the service rate, the queue length rises exponentially to the setpoint of 20 packets, where it stays till the end of the simulation. It is clear that $\hat{n}_b$ is a good estimator of the queue length. The number of outstanding packets is around 120, which is the pipeline depth of 100 added to the buffer set-point of 20.

A more detailed view of the queueing is shown in Figure 12. Note that the packets arrive as pairs, and depart with a spacing of around 1 time unit, which is the service rate at the bottleneck. The arriving pairs exactly synchronize with the departing pairs so that the oscillation around the bottleneck is exactly 2 packets in amplitude, with a mean of the buffer setpoint of 20 packets. The buffer size estimate also

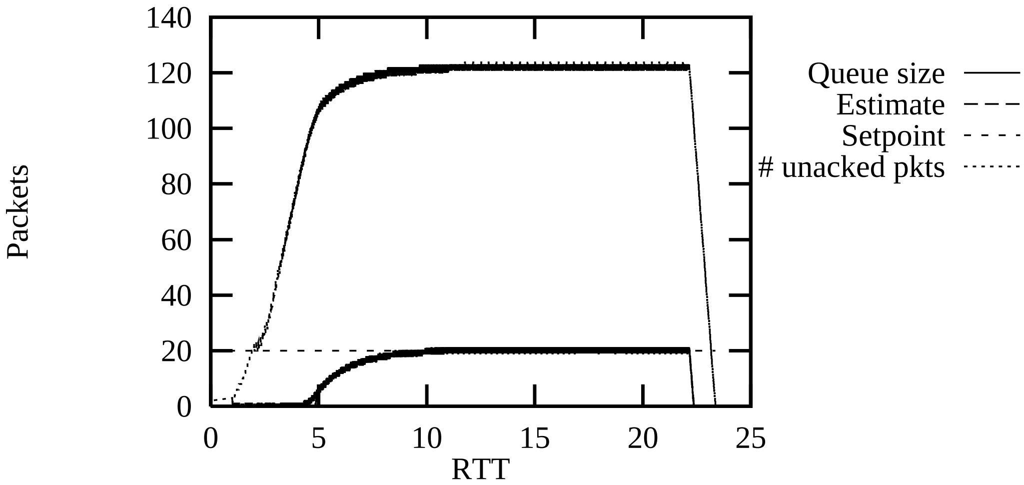

Figure 11: Queue length, setpoint and number of outstanding packets

oscillates around this point.

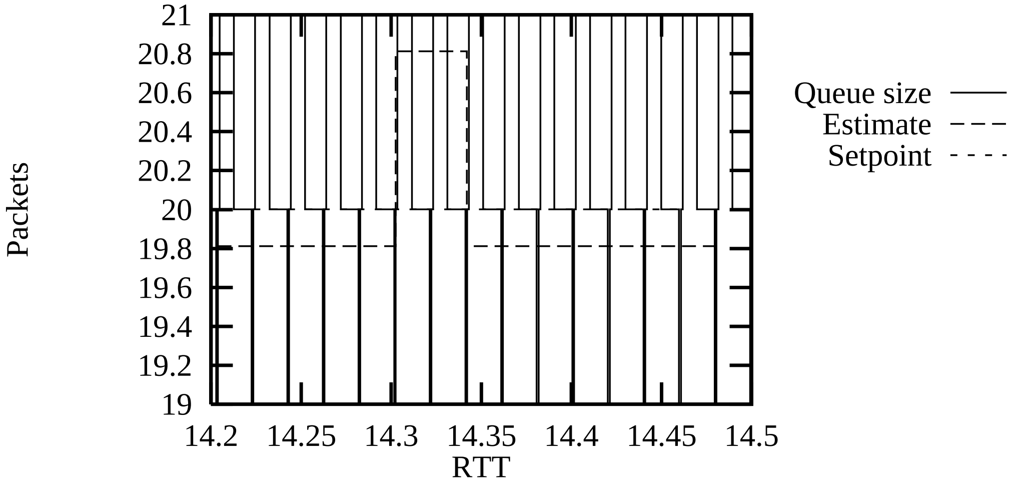

Figure 12: A closer look at the queue size, its estimator and the setpoint

The flow control does not immediately reach the buffer setpoint because of the adaptive exponential rise during startup. The time to reach the setpoint is seen in the switch utilization curve. The utilization is measured over intervals of 25 time units. Note that the 99% utilization mark is reached in about 4.5 round trip times. The utilization stays at 100%, since the bottleneck queue is never empty.

Finally, the sequence numbers and acknowledgement numbers rise smoothly, with a slope equal to the service rate at the bottleneck (Figure 14).

In the next experiment, we enable buffer setpoint probing. Setpoint probing is not a great advantage

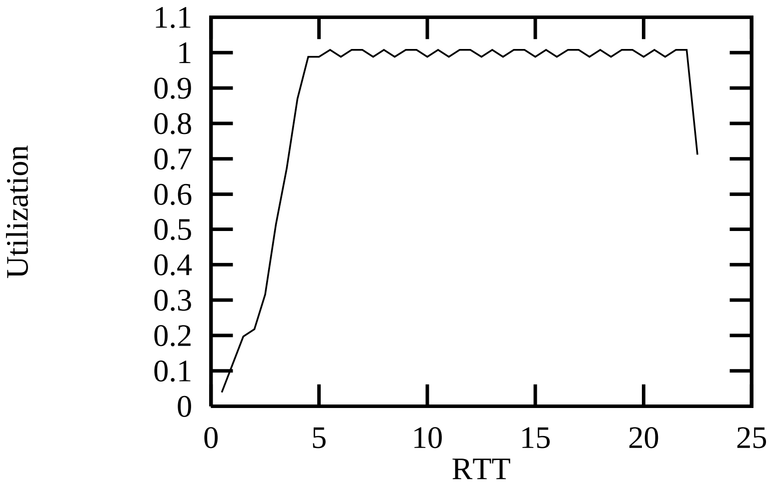

Figure 13: Switch utilization

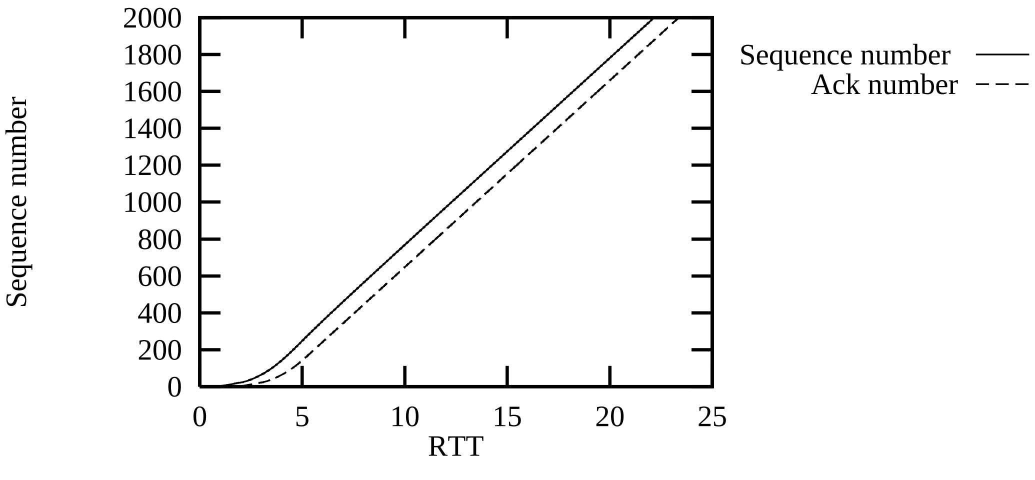

Figure 14: Sequence number and acknowledgement number space

when there is only a single source. In fact there is no need for probing with only a single source since there is no source of variation in the buffer share. Probing is more useful when the number of sources, and therefore the fair buffer share, changes with time. Figure 15 shows dynamic setpoint probing where four sources share a bottleneck. Note that the sources increase their setpoint till the sum of the setpoints reaches 100, at which some packet losses occur, and the four sources immediately reduce their setpoint. The setpoints oscillate around 25, which is the fair share of buffers. Though not shown here, the queue lengths exactly follow the setpoints. Because of the retransmission strategy described in Section 6, the losses affect the sources minimally. This is illustrated by the sequence number trace in Figure 16. Note that each loss is corrected by a single retransmission, and losses do not affect the sending rate, reflecting the decoupling of the flow control and retransmission strategy.

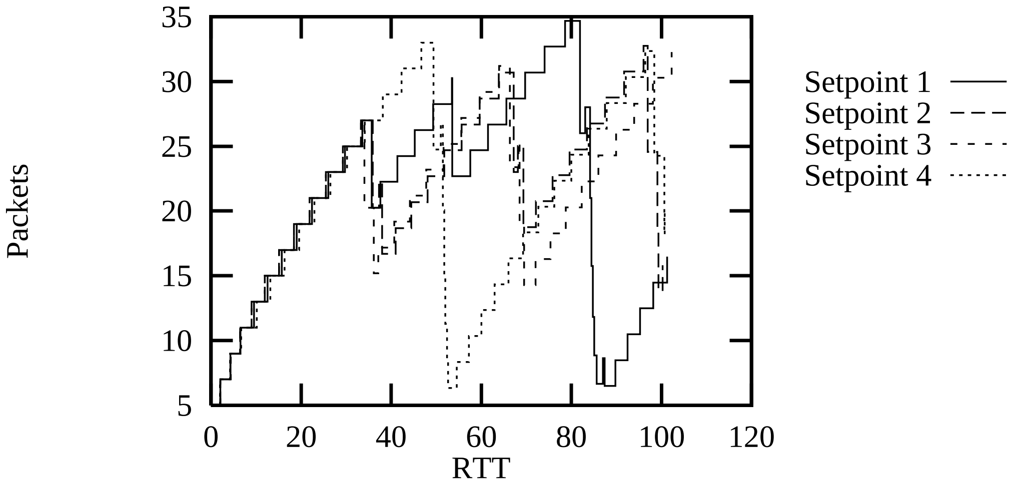

Figure 15: Dynamic setpoint probing for four sources sharing 100 buffers

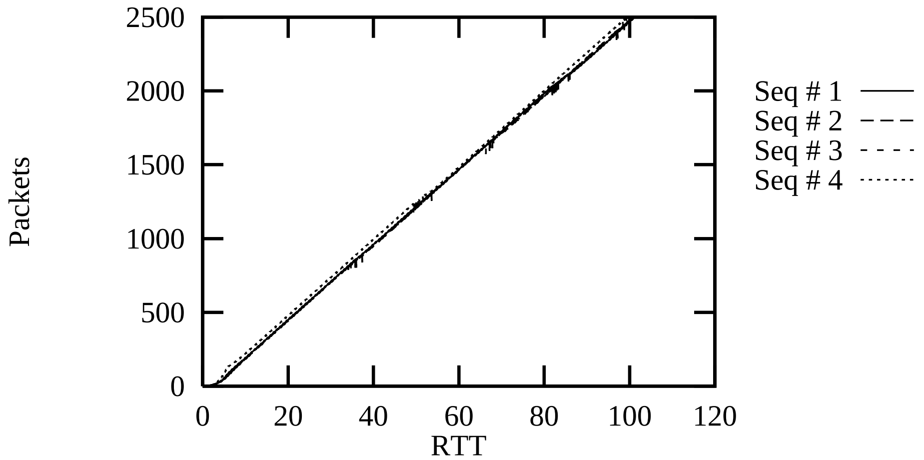

Figure 16: Sequence numbers for the four sources

During probing, the setpoint is increased linearly by 2 every 2 round trip times, and if a loss occurs, the setpoint is multiplicatively decreased by a factor of 0.75. If the setpoint increase factor were increased, the increase interval were decreased, or the decrease factor made closer to one, the frequency of setpoint probing would increase. However, this would induce more losses. One can view each loss event as giving some information about the highest achievable setpoint. Better information is achieved only by increased losses. Thus the goal of loss-free transfer and setpoint probing are mutually contradictory. We prefer to somewhat conservative in inducing losses, as evidenced by the choice of probing parameters. It is possible to trade off losses for better probing. As an example, in the scenario above, the four sources each lost three to six packets during the course of the simulation. With additive increase in the setpoint of 1 every two round trip times, there is only one loss per source, which would be a more conservative tradeoff.

Another problem with dynamic probing is that the instantaneous setpoints achieved by the different sources are not necessarily fair. Given a buffer of size K and N sources, we would like the setpoints to converge to a fair buffer share of K/N. Though instantaneous bandwidth shares are guaranteed to be fair by the round-robin scheduling discipline, since loss events cause drastic changes in the setpoint, over short intervals of time, the buffer share is unfairly distributed. However, since Fair Queueing follows the policy of dropping a packet from the longest queue, over long enough periods, the buffer share will also be fairly distributed.

The same problems are present in a more severe form in the 4.3 BSD implementation of TCP, where induced losses are used to probe for the window size [26]. However, in their case, since the probing is for the flow control window, uneven losses lead to unfair bandwidth allocation. In our case, the unfairness in setpoints does not lead to unfair allocations of bandwidth.

9.2. Dynamics of Interaction Between Sources

In the next set of scenarios, we study the behavior of a source as additional sources start up and share the bottleneck bandwidth. Two cases are of interest. In the first, we study the effect of adding a source to an existing set of sources. In the second, we study the effect of adding N additional sources simultaneously, as N varies over some range.

For the first case, the source under study starts at time 0, and new sources start up every R round trip times, until 10 sources are active. As each source starts up, we see the incremental effect of a source when N sources are already active, and N ranges from 1 to 9. The behavior of the system depends on the choice of R. If R is less than 1, then the source under study does not have time to react to a change, before the next change occurs. If R is more than 1, then the source has a chance to adjust its sending rate before seeing the next change. Thus, we study two sub-cases - one where R = 0.5, and another where R = 2.

Figure 17 shows the sending rates versus time for sources 1 (the source under study), 2, 6 and 10 when R = 0.5. Source 1 sends 1500 packets and the other 9 sources send 500 packets each. The start times are staggered 50 time units apart starting at time 500. Source 1 initially has a sending rate of 1, which completely uses the bottleneck. As the setpoint for 1 increases, its queue occupancy also rises. At time 500, source 2 starts up, and every 50 time units after that a new source starts up. Since each source does an adaptive exponential rise, which takes about 5 round trip times to complete, the effect of the additional sources on source 1 is not as drastic as it might be (Figure 17). The last source starts up at time 950, and by time 1000, source 1 has brought its rate down to below its fair share of 0.1, so that its queues drain to the fair buffer share of 10. From time 2000 onwards, the system is quite stable, with all the sources sharing bandwidth equally. As sources terminate, their share is distributed to the remaining sources, which appears as a rise in the sending rates towards the end of simulation time.

Figure 18 shows the sending rates versus time for the case where R = 2. As before source 1 stabilizes by time 500 slightly above the bottleneck rate of 1.0, so that its queue size increases with the increasing setpoint. As source 2 starts up, source 1's sending rate decreases to around 0.5, then drops incrementally as more sources become active. By the time source 10 starts up at time 2100, source 1 is nearly at its fair share rate of 0.1 and is not affected much by the new source. Two conclusions are clear from the this and the previous experiment. First, the degree to which source 1 is affected by a new source depends on the number of sources already active. Second, the variation in sending rate for a new source also decreases as the number of active sources increases. This is easy to explain - if there are N sources active, the addition of a new source changes the bottleneck service rate from $1/N$ to $1/(N+1)$, a change of $1/N(N+1)$. As N increases, this fraction decreases by a factor of N^{-2}, and so the system behaves in a more stable fashion. In this sense, packet-pair scales well with N. This is desirable.

We now turn our attention to the behavior of a source as N sources are simultaneously added to the system. As N increases, the source under study suffers a bandwidth loss of $1 - 1/N$, so that the ratio of new service rate to old decreases linearly with N. We study two cases, one where the buffer size is 100 buffers

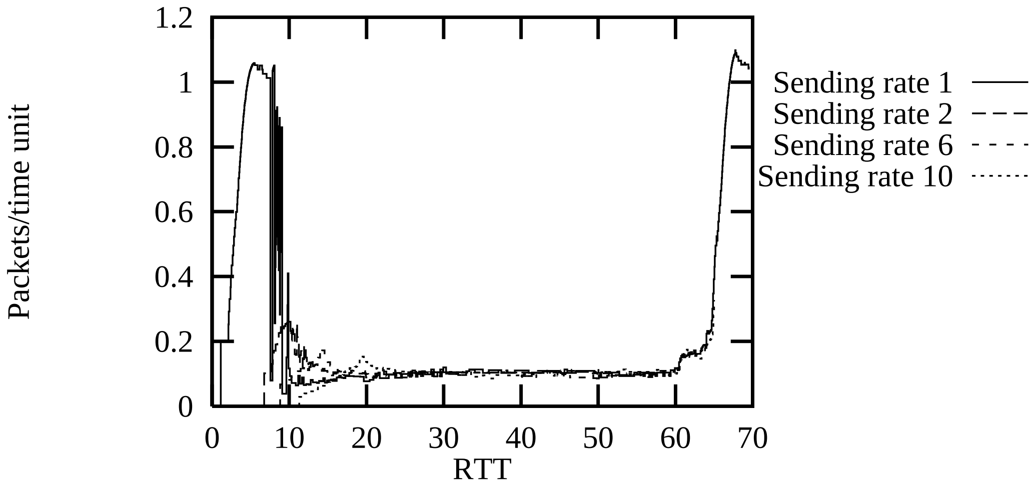

Figure 17: Sending rates as new sources are added every 0.5 RTT

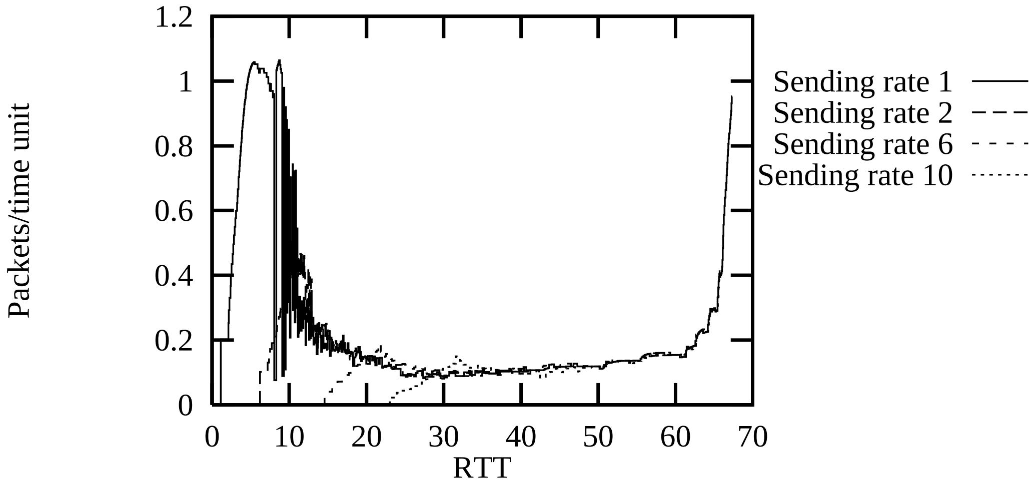

Figure 18: Sending rates as new sources are added every 2 RTT

as before, and we look at the number of losses suffered by source 1, and another where the buffer size is infinite, and we study the peak buffer size attained by source 1.

We first consider the behavior of source 1 for the case where nine other sources start up at time 1000. This behavior of source 1 goes through nine distinct phases, explained below. Figure 19 shows the sending rate in packets/second and the inter-ack spacing seen by the packet pair probe sent from source 1, and Figure 20 shows its queue at the bottleneck. Recall that the service rate is approximately 1 packet per second. Thus, when all 10 sources are active, the inter-ack spacing is 10, and the sending rate ought to be around 0.1.

In the first phase, from time 0 to 1000, the source sees an inter-ack spacing of 1 and sends at 1 packet per second, after the slow start period discussed earlier. At time 1000, phase 2 begins when all the other sources send their first probe, and source 1's service rate drops to 0.1. However, in the period before this is seen, the sending rate stays at 1.0, thus building up the queue. At this time, packet losses are possible. Phase 3 begins about one round trip time from the end of phase 1, when the new service rate is known, and the estimate for the bottleneck queue ($\hat{n}_b$) suddenly increases. Thus, to allow the bottleneck queue to drain, the sending rate rapidly drops to nearly zero. At the end of the third phase, the source sends out a new probe. However, this probe encounters none of the other sources, since they are all in their slow start phases, and source 1 imagines that it is alone in the network (the inter-ack spacing is seen to be 1). So, it begins transmitting at the full rate (phase 4). However, the other sources soon attain their final sending rates, and source 1's queue starts building up (this is the second queue buildup, with packet losses again possible). When source 1 finally sees the inter-ack spacing of 10, and all the sources are simultaneously active, source 1 sends at its correct rate of 0.1 (phase 5), and the bottleneck queue is at its correct value of 10 packets. Since packets lost in phase 1 and retransmitted in phase 4 are also lost, every two round trip time, the source attempts several soft timeouts (the downward spikes in Figure 21). When the competing sources complete, the service rate suddenly increases to 1, and so source 1's queue rapidly drains. This is phase 6. In the seventh phase, source 1 detects that the service rate has increased to 1.0, and the queue increases to the setpoint. The source is now able to send at the full rate till the end of its transmission, unless the soft timeouts are unable to fill in the lost packets in the transmission window. In this case, the source will be able to send only until the receive window fills up, at which point it must wait for a timeout before the packet is retransmitted for the second time, and the receive window opens up again. Thus, in the eighth phase, the source is idle, waiting for a timeout, and in the ninth phase, it begins a new slow-start. Phases 8 and 9 do not occur in this example, but have been observed for larger values of *N*.

Figure 21 shows the loss behavior for source 1. Note that almost the entire window is lost at time 1000, and this is retransmitted in phase 4, where some packets are again lost. The source recovers from this loss during phase 5 using soft timeouts.

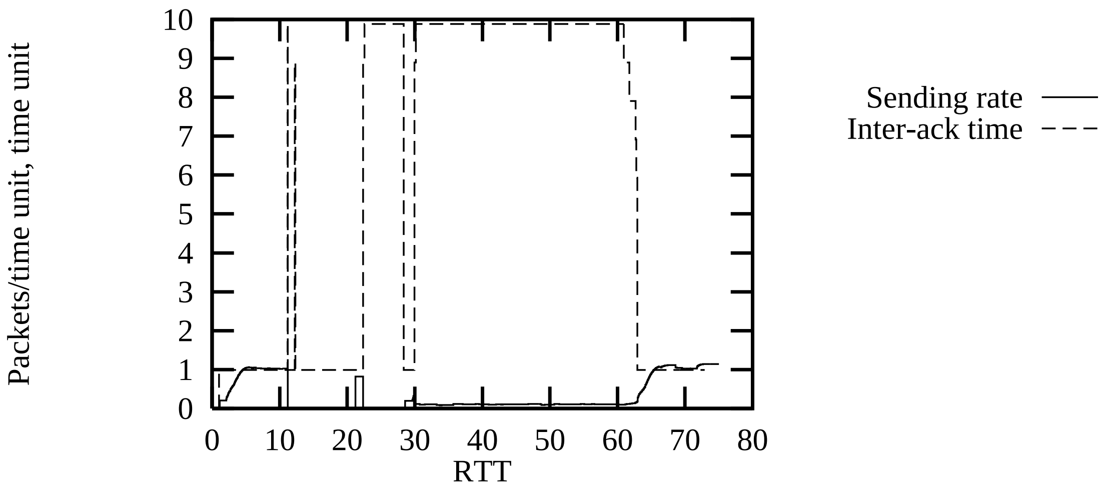

Figure 19: Sending rate and inter-ack probes for source 1

This scenario illustrates the difficulty in designing a good retransmission strategy for a rate-based flow control algorithm. One needs to design not just the rate probing algorithm, but also the choice of timeout, slow-start strategy, what information is sent back to the source by the receiver, and problems with incorrect probes. We believe that we have achieved a reasonably good mix in packet-pair flow control.

We now consider some quantitative metrics of packet-pair performance as *N* increases. Table 4 shows

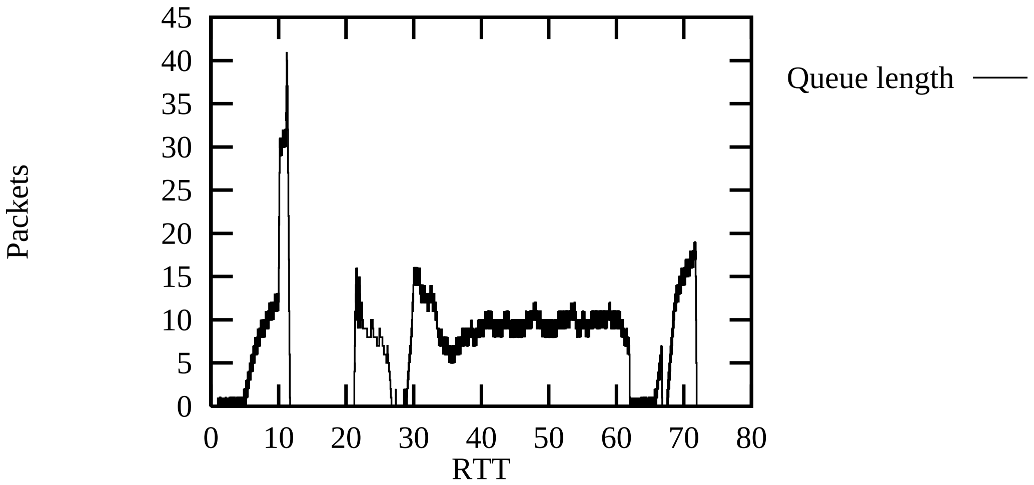

Figure 20: Queue length at the bottleneck for source 1

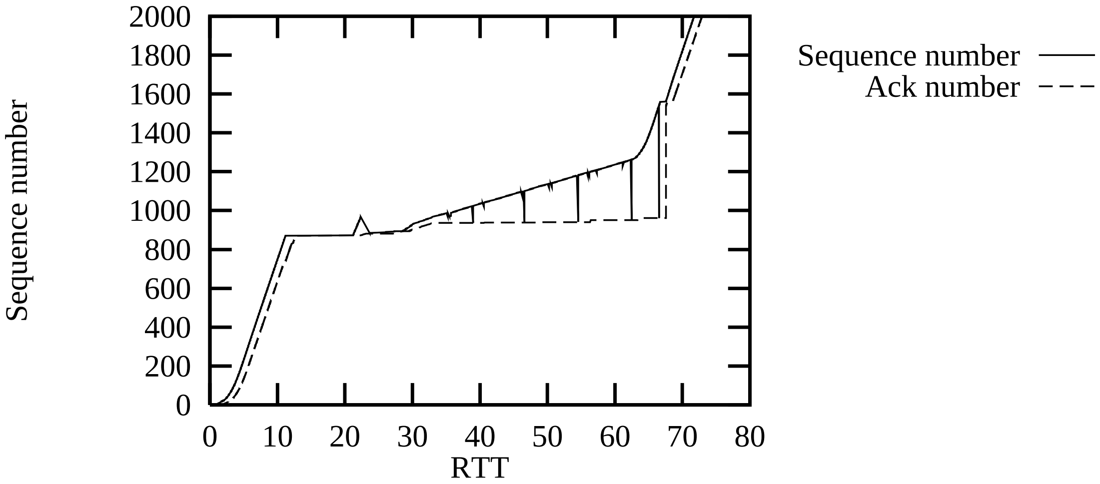

Figure 21: Sequence and acknowledgement space for source 1

the number of loss rate, retransmission rate, the completion time, and the peak buffer occupancy as a function of N. Note that the completion time increases monotonically. This is to be expected, since the total number of packets served increases as N increases. The loss rate and retransmission rate match exactly, which shows that the retransmission strategy is optimal, as claimed. The loss rate (expressed as mean number of packets lost per 1000 time units over the length of the simulation) generally increases with N. This is because the buffer share decreases as $1/N$. The peak buffer occupancy shows an interesting trend, first decreasing and increasing by turns. The explanation is that these peaks are achieved in different phases, and that in each phase, the peak is achieved for a different N.

As N increases, the peak in phase 2 increases, and is maximum for $N = 10$ at 41. The peak for phase 4 depends on the slow start behavior of the other sources, and peaks at $N = 7$ with an occupancy of 76. When $N = 2$, the peak occupancy of 83 is achieved because the loss in service rate is small, and the queue achieved at the end of phase 2 is not much larger than the target setpoint. Thus, phase 3, where the sending rate slows down to drain the queue does not occur. So, the phase 4 buildup adds to the phase 2 buildup, leading to a large queue. For $N = 3$ and $N = 4$ also there are no losses, but phase 3 is longer, and the phase 4 buildup is smaller since the rate probes are more likely to interfere with competing sources than when $N = 2$. Thus, the peak phase 4 buildup is achieved when $N = 2$. To summarize, for N in [2,4], there are no losses, and the peak is for $N = 2$, when there is least interference. For N in [5,7], there are packet losses in phase 2, as well as queue draining in phase 3, and the subsequent phase 4 buildup peaks when $N = 7$. Finally, the phase 2 buildup peaks when $N = 10$.

N	Loss Rate (pkts/1000 TUs)	Retransmission Rate (pkts/1000 TUs)	Completion Time (TU)	Peak Buffer Occupancy (pkts)
2	0	0	2761	83
3	0	0	3291	31
4	0	0	3803	35
5	0.6	0.6	4435	40
6	0.5	0.5	4941	44
7	3.7	3.7	5542	76
8	9.0	9.0	6519	35
9	9.9	9.9	6921	38
10	8.9	8.9	7172	41

Table 4: System with limited buffers

We now consider peak buffer occupancy when the buffer is very large (10,000 packets) (Table 5). As before, when $N = 2$, phase 3 is missing, and the phase 2 and 4 buildups add up, so that the peak queue occupancy of 83 is reached at that time. Phase 4 buildup is much larger than phase 2 buildup in all cases, and this peaks for $N = 9$ where it is 101. $N = 10$ achieves the largest phase 2 buildup (41), but this is smaller than its phase 4 buildup of 86.

N	2	3	4	5	6	7	8	9
Peak Occupancy	83	31	35	40	44	82	84	101

Table 5: System with unlimited buffers

We draw several conclusions from the study of packet-pair dynamics. First, the dynamics are influenced by a number of factors, and it is important to achieve a synergy between retransmission and flow control strategies. Second, the scheme behaves nearly optimally even when there are drastic changes in the bottleneck capacity. The estimator for the bottleneck queue length tracks the real queue length quite closely, due to the fuzzy prediction algorithm. The setpoint probes allow the sources to share bottleneck buffers fairly. Third, the peak buffer occupancy is achieved in one of two phases of queue buildup. The phase 2 buildup is as expected, and reflects the lack of information about the rate change for one round trip time. The amount of buildup is exactly the minimum predicted by multiplying the change in the service rate by the round trip time [33]. However, phase 4 buildup because of improper rate probing when several sources are in their startup phase, was somewhat unexpected. As it turns out, this effect dominates peak buffer occupancy. Fourth, we the simulations show that packet-pair is robust to large changes in service rate, large numbers of packet losses, improper rate information and small number of available buffers in situations where the bandwidth delay product is as large as 100 packets. This robustness is necessary for any practical rate-based flow control mechanism.

9.3. Goodput in a Public Data Network

We now study a scenario that models a public data network. Here, N on-off sources regulated by packet-pair flow control compete for a single bottleneck server. Each source is modeled as a pair of processes. The producer process places some number of packets in the transmission queue during the exponentially distributed on time and then is idle for the exponentially distributed off time. The consumer process is the packet-pair flow regulator, which empties out the transmission queue into the network. The producer process has a nominal rate λ, which is the mean rate at which it places data into the transmission queue. The actual source rate can be less than λ if so determined by the flow control algorithm. The metric of interest is to study the throughput and loss behavior of the system as the sum of the nominal source rate increases and then goes beyond the bottleneck capacity. Since the sources are stochastic in nature, not all the sources would be active all the time, leading to possible congestion.

Since this is a large public network, we choose to simulate 10 active sources in all the scenarios. We change the loading by increasing the sum of the nominal rates from 0.5 of the bottleneck capacity to 5.0 of this capacity. Two types of sources are studied - open and closed sources. An open source puts its data in the transmission queue independent of the state of the queue. Thus, the sum of the on and off duration is independent of the network loading. This models a file transfer type application. A closed source enters its off time only when the transmission queue is empty. Thus, the actual sum of on and off times varies with the load of the system. The open system models a file transfer type of application, the closed system models a transaction processing environment.

We measure the performance of packet-pair in this scenario by the goodput, which is the retransmission rate subtracted from the transmission rate, as a function of the sum of the nominal loads of all the sources. Losses (and retransmissions) are fairly shared by all sources due to the fair buffer loss property and the fair bandwidth allocation property of Fair Queueing. So, we simply add the retransmissions incurred by all the sources, and subtract this from the throughput to determine the goodput. Each of 10 sources sends 2000 packets during the course of the simulation, so RTX total retransmissions corresponds to a mean goodput of $1 - RTX/20{,}000$. Table 6 shows the mean goodput per source as the nominal rate increases for the open and closed systems. In both cases, the bottleneck has 100 buffers, and the bandwidth delay product of the network is 100 packets.

Nominal Load	Open System		Closed System	
	Retransmissions	Goodput	Retransmissions	Goodput
0.5	0	1.0	0	1.0
0.75	0	1.0	0	1.0
0.9	205	0.99	50	0.99
1.0	506	0.97	394	0.98
1.5	387	0.98	739	0.96
2.0	398	0.98	845	0.96
5.0	382	0.98	235	0.99

Table 6: Goodput in a public data network

Note that in all cases, the goodput is above 95%, which is excellent, considering that the sources share a single round trip time worth of buffers, and the delay is substantial. Further, though this is not shown, for all the cases, each sources maintains approximately 10 packets in the buffer at all times, thus achieving the fair setpoint. To our mind, these simulations indicate that packet-pair is suitable for best effort traffic for B-ISDN networks.

9.4. File Transfer Benchmark Tests

In this section, we study the behavior of packet-pair in a set of benchmarks that are designed to stress the flow control algorithm in a number of ways [31]. We compare the behavior of packet-pair to the optimal flow control, defined below. We use sources with finite amounts of data (such as a file) to send and use the file transfer time as the primary metric to compare the performance of various schemes. The file transfer time is defined as the interval between the instant at which the first byte of data is available at the transport-level of the sender and the instant at which the sender receives the acknowledgement that the destination has received the last byte of the file. By carefully designing the topology of the network and the experimental environment, this measure subsumes other traditional performance measures such as link utilization, fairness and throughput. Several considerations about the design of such a benchmark are presented in [31]. In this work, we simply present the benchmarks and results obtained for packet-pair.

We do not compare packet-pair with other schemes for two reasons. First, all the other schemes have been designed for networks of first-come-first-served servers, and so are optimized for that case. Since the behavior of FCFS servers is not as 'nice' as that of round-robin-like servers, packet-pair has an unfair advantage. A more pragmatic reason is that the number of flow control schemes proposed in the literature is rather large, and it would be impractical to study them all. Further, most schemes are parametrized by a large number of parameters that are not always published in the literature. For these reasons, we only present the benchmark results for packet-pair.

The optimal file transfer time is obtained analytically by assuming that the bottleneck has infinite buffers. Thus, the optimal scheme would be to send the entire file into the bottleneck, where it would be stored, and served at the fair share of the capacity. In a round-robin like network, this transfer time would be lowest achievable.

9.4.1. Benchmark 1: Slow Changes in Cross-Traffic

The configuration used for this experiment is shown in Figure 22. In this scenario, the cross-traffic load seen by the primary connection changes gradually. Connection 1, shown in Figure 22, starts up at time zero and has 2 Megabytes of data to send. The bottleneck has one round trip time worth of buffers. The first cross-traffic stream, connection 2, starts at 100 ms and nine more cross-traffic streams come up at intervals of 44 ms (which is the round-trip propagation delay for all the connections). Since the fundamental time constant for reacting to changes in network state is one round trip time (RTT), the flow control mechanisms are highly stressed when new traffic sources come up spaced apart one RTT. Each cross traffic stream has 500 Kbytes of data to send. The transmission speeds of links and the size of files are chosen such that the cross-traffic load gradually increases and then decreases back to zero during the interval in which connection 1 is transmitting its data. As Table 7 shows, packet-pair achieves a throughput within 10% of optimal in this case.

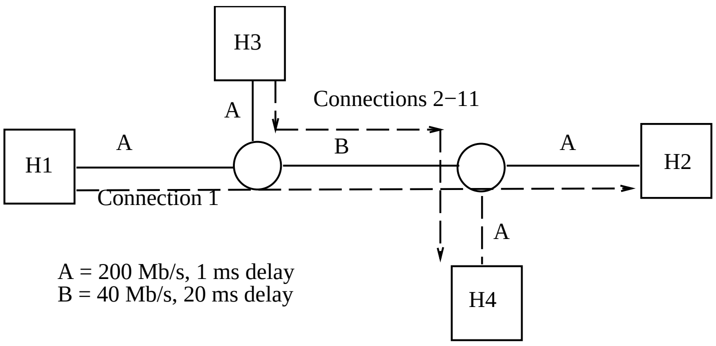

Figure 22: Configuration for Benchmark 1

Optimal	Packet-pair	Slowdown
1444	1576	9.1%

Table 7: Benchmark 1: Completion times (in milliseconds)

9.4.2. Benchmark 2: Sudden Jumps in Cross-Traffic

The configuration used for this experiment is the same as in the previous experiment and is shown in Figure 22. However, in this scenario, the cross-traffic load seen by the primary connection changes very rapidly. This scenario evaluates packet-pair in terms of buffers required at the bottleneck in face of a large perturbation. Connection 1 starts up at time zero and has 2 Megabytes of data to send. The first cross-traffic stream, connection 2, starts at 100 ms and nine more cross-traffic streams come up at intervals of 4 ms. This is much smaller than one RTT, so connection 1 experiences a large drop in available bandwidth within one RTT. At the same time, the interval of 4 ms is large enough to throw off any schemes that simply count the number of new connection requests and accurately allocate bandwidth to existing and new connections. Each cross traffic stream has 500 Kbytes of data to send.

Optimal	Packet-pair	Slowdown
1444	1599	10.73%

Table 8: Benchmark 2: Completion times (in milliseconds)

This benchmark shows that even with large changes in the available bit rate, packet-pair is stable, and shows only a small degradation in performance (10.7% slower than optimal compared to 9.1% slower in Benchmark 1). This confirms the results in Section 9.2.

9.4.3. Benchmark 3: Transfer Time for Short Files

The transfer time is a critical performance measure for small file transfers. The main factor that increases the transfer time for short amounts of data transfers is the ‘‘slow-start’’ mechanism used by many flow control schemes (Section 5). In this experiment, we isolate the slow start mechanism from the effect of lost packets and lost bandwidth due to cross traffic. Hence, there is no cross traffic. The configuration used in this experiment is shown in Figure 23. File sizes of 4, 20 and 200 packets are used in this configuration.

The file transfer times for sending files of 4, 20, 200 and 2000 packets are shown in Table 9. The results indicate that as the number of packets sent, N, increases, packet-pair has an increasing, and then a decreasing slowdown factor. It is easy to explain why the factor decreases with N. Note that if R is the round trip propagation delay, the optimal scheme completes at time $R + N/\mu$. Packet-pair stablizes roughly at time $5R$. So, its completion time would be roughly $5R + N/\mu$. Thus, the slowdown ratio is $(5R + N/\mu)/(R + N/\mu)$, which can be written as $(K + 5)/(K + 1)$, $K = N/(R\mu)$. This function, and thus the slowdown factor, asmyptotically approaches 1 as N increases.

The increase for small N is harder to explain, since it depends in detail on the behavior of packet-pair. Instead, we use the acknowledgment number trace for the scenario with a transfer length of 2000 packets to compute the slowdown factor for every value of N. This is shown in Figure 24. Note that the peak slowdown of about 140% is achieved for a transfer of about 100 packets.

We feel that this order of slowdown is inevitable for any real-life algorithm that has to deal with imperfect knowledge about network state at the time of start up. In fact, all the other proposals for slow-start show similar or worse behavior (recall that for the DECbit algorithm [48], to achieve a window size of 100 would take 100 round trip times, so that packet-pair performs roughly 20 times better).

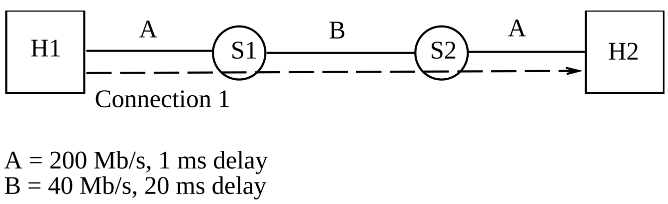

Figure 23: Configuration for Benchmark 3

# Packets	Optimal	Packet-pair	Slowdown
4	44.4	84	89%
20	46	92	100%
200	64	146	128%
2000	244	330	35%

Table 9: Benchmark 3: Completion times (in milliseconds)

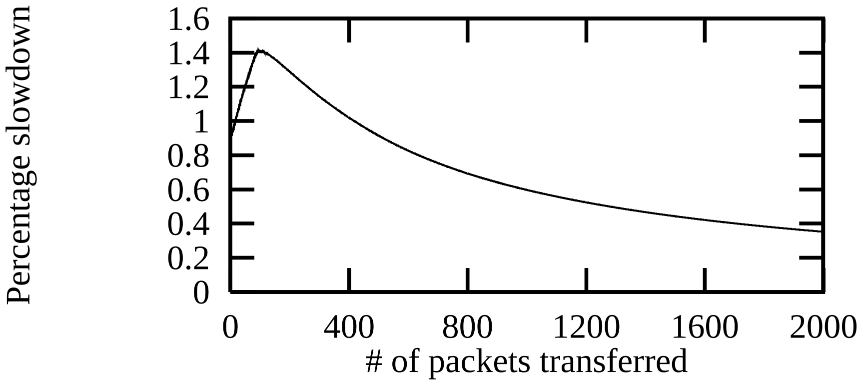

Figure 24: Percentage slowdown as a function of number of packets transferred

9.4.4. Benchmark 4: Fairness

Previous studies of window flow control performance have observed that connections on a longer delay path get a smaller share of the bandwidth of the bottleneck (for example, see [12]). This ‘‘unfairness’’ increases the file transfer times for traffic streams going over longer paths. In this experiment the fairness in bandwidth allocations is measured. The configuration used in this experiment is shown in Figure 25. Two primary connections, 1 and 2, are started at time 0 and both have 2 Megabytes of data to send. Connection 2 goes over a path with approximately twice the end-to-end propagation delay of that of connection 1. Both connections see the same cross-traffic load at the bottleneck link shared by these connections. The starting times and file sizes of the cross traffic streams is the same as in Experiment 1.

The file transfer times for sending files over the two different delay paths are shown in Table 10. Because of the different reaction times to a change, every flow control scheme will discriminate against the connection with the longer delay paths to some degree. Even the optimal file transfer time for Connection 2 is slightly longer than for Connection 1 because of the longer round trip delay. However, packet-pair performs within 6% of optimal for both the long and short connection. This discrimination against longer delay paths can be mitigated to some extent by choosing a larger buffer setpoint for longer paths.

9.4.5. Benchmark 5: Migrating Bottlenecks

There will be only one bottleneck at any one time in the path of a connection. But the site of the bottleneck may migrate with changes in the network traffic. In this experiment, we explore the performance of flow control schemes in the presence of migrating bottlenecks. For simplicity, we have considered a generic case where a bottleneck first develops at a link, shifts to another link and then finally shifts back to the original link. The shifts in the site of the bottleneck are induced by introducing cross-traffic streams with different file transfer sizes over two separate links. The primary connection comes on at time zero with 2

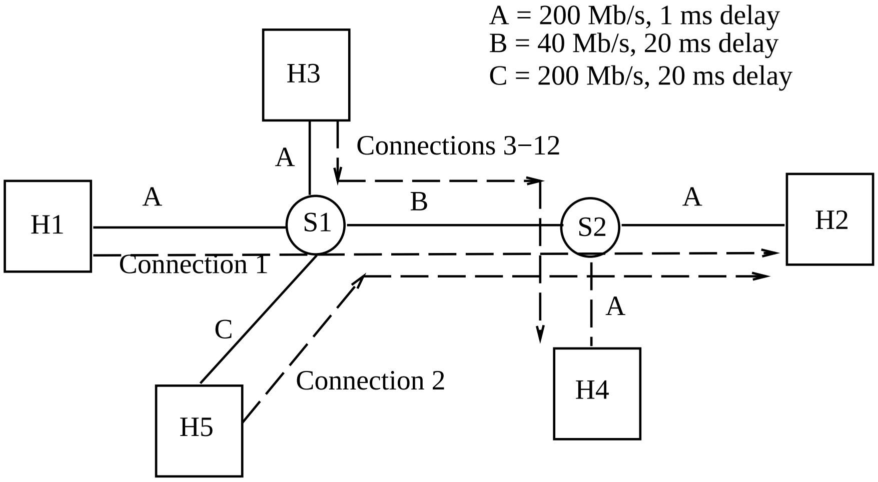

Figure 25: Configuration for Benchmark 4

Optimal		Packet-pair		Slowdown	
Short	Long	Short	Long	Short	Long
1844	1882	1869	1993	1.4%	5.9%

Table 10: Benchmark 4: Completion times (in milliseconds)

Megabytes of data to send. Connection 2 switches on at 100 ms with 1 Megabyte of data to send. After connection 2 switches on, the S1-S2 link becomes the bottleneck for connection 1. At time 250 ms connections 3 and 4 switch on with 250 Kilobytes of data each to send. This causes the bottleneck for Connection 1 to shift to the S2-H2 link. When connections 3 and 4 terminate the S1-S2 link again becomes the bottleneck for connection 1. The configuration used in this experiment is shown in Figure 26.

The file transfer times for sending files are shown in Table 11. Surprisingly, the migrating bottleneck traffic scenario results presents little difficulties to the packet-pair scheme even though it does not get explicit feedback from a bottleneck. The result seems to suggest that the filters used to predict and adapt rates in packet-pair scheme are functioning reasonably well.

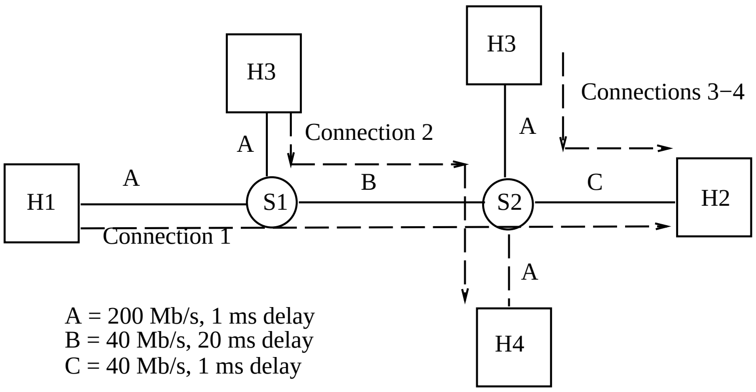

Figure 26: Configuration for Benchmark 5

Optimal	Packet-pair	Slowdown
644	779	21.0%

Table 11: Benchmark 5: Completion times (in milliseconds)

9.4.6. Benchmark 6: Two-Way Traffic

A traffic scenario with traffic flowing in both directions through a switch is important to examine because of the role played by acknowledgement packets in some flow control schemes [66]. The TCP scheme uses acks as a clocking mechanism to inject packets into the network at roughly the bottleneck rate. The packet-pair scheme uses the gap between acknowledgment packets to estimate the service rate at the bottleneck node. Unfortunately, whenever there is congestion or even burstiness in the traffic along the reverse path, there is a chance that two successive acks may get bunched up and thus adversely affect schemes that use the frequency of ack arrivals or inter-ack gaps in their control policies. This experiment is designed to detect any such conditions and to measure its impact on the performance.

The configuration used in this experiment is shown in Figure 27. The configuration is similar to that used in Experiment 1 with all the connections being duplicated and reversed so as to create an exactly symmetrical flow in both directions. Thus, the primary reverse connection is started by the host that is the sink for the primary forward connection. Both the primary forward and reverse connections start at time zero. Cross-traffic streams are also mirrored in a similar fashion.

The file transfer times for sending files in the forward and backward directions are shown in Table 12. These times should be compared to the file transfer time observed in Experiment 1 in order to determine the impact of having a congested reverse path. Note that the optimal file transfer time is longer because the data packets have to share bandwidth with the ack packets. Packet-pair performs better than expected (even better than in Benchmark 1!) because the fuzzy filtering of the spacing between acks, which ignores the noise generated by bi-directional traffic.

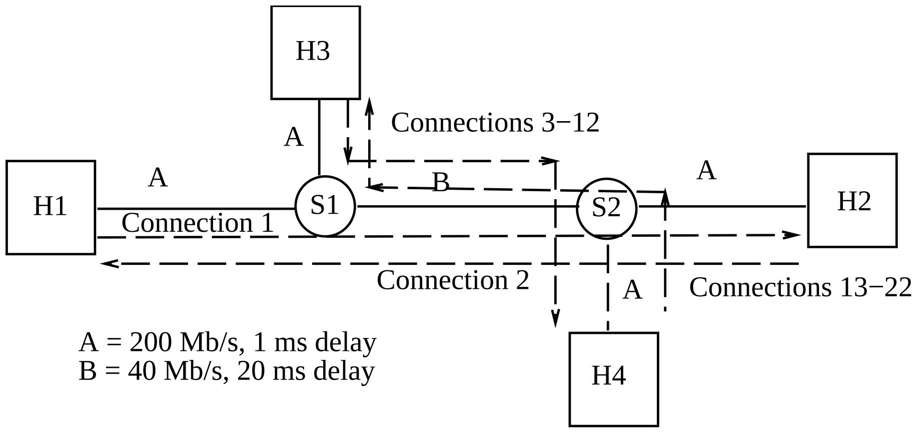

Figure 27: Configuration for Benchmark 6

Optimal		Packet-pair		Slowdown	
Conn 1	Conn 2	Conn 1	Conn 2	Conn1	Conn2
1556	1556	1677	1677	7.8%	7.8%

Table 12: Benchmark 6: Completion times (in milliseconds)

9.4.7. Benchmark 7: Non-compliant Cross-traffic

In a network, one should not expect that all traffic streams will be subject to a single flow control discipline. For example, a real-time traffic source such as a packet video camera is unlikely to follow the same flow control scheme as that followed by a bursty data source such as a file transfer. Furthermore, there will always be some short data transfer applications (such as distributed computation) which will not be subject to any end-to-end flow control. Thus, it seems important to measure the performance of packet-pair when other non-compliant traffic streams are present. The configuration used in this experiment is shown in Figure 22. The configuration is identical to that used in Experiment 1 except that the cross-traffic streams are uncontrolled and the packet arrival process for each cross traffic source is Poisson such that the aggregate arrival rate of all the cross connections is 20-90% of the bottleneck link bandwidth. As before the primary connection comes on at time zero and the cross traffic streams switch on at 44 ms intervals, starting at 100 ms.

The file transfer times are shown in Table 13. The slowdown percentage decreases with the increase in cross traffic intensity. This reflects the time lost due to slow start, which is independent of simulation length. Since this factor plays a smaller role as the optimal transfer time increases, the slowdown from optimal correspondingly decreases. These performance results show that that packet-pair does not do much worse even with non-compliant cross traffic. Thus we believe packet-pair to be robust and well-behaved even in the presence of non-compliant cross-traffic streams.

Cross traffic Intensity	Optimal	Packet-pair Mean	Packet-pair Variance	Slowdown
0.2	544	681	8.5	25.2%
0.4	710	934	7.9	31.5%
0.6	1044	1331	19.5	27.5%
0.8	2044	2236	26.8	9.3%
0.9	4044	4112	134	1.7%

Table 13: Benchmark 7: Completion times (in milliseconds)

9.4.8. Benchmark 8: Fewer Data Buffers at a Switch

In this experiment, we study the effect of having fewer buffers at the switch. While we believe that having at least one round trip time worth of buffering (shared among all conversations) is a minimum, it may sometimes be the case that this amount of buffering is not available. Here the scenario is as in Experiment 1, except that the switch has only one tenth of a round trip time worth of buffers.

Table 14 shows that despite having very few buffers, packet-pair is able to perform quite well. With one RTT worth of buffering, the completion time was 1576ms, with a tenth of that, the completion time deteriorated only by around 110ms to 1686ms. This is because of the robust and efficient retransmission strategies developed in Section 6.

Optimal	Packet-pair	Slowdown
1444	1686	16.76%

Table 14: Benchmark 8: Completion times (in milliseconds)

9.4.9. Benchmark 9: A Torture Test for Rate-based Schemes

This scenario is designed to be a ’torture test’ for rate-based schemes, in that it stresses what are believed to be the weakest aspects of rate-based flow control. The configuration used in this experiment is shown in Figure 28. In this scenario, we have Connection 2 with a round trip time of 20.2 ms and Connection 1 with rtt of 0.2 ms sharing a bottleneck link. Cross-traffic is provided by 2 on-off sources that start at times 1.5 ms and 3 ms respectively. Each on-off source sends at 25% of the total bottleneck link’s capacity for 2.5 ms and then goes idle for 1.5 ms. The number of backlogged sources changes thus changes from 1 to 2 to 1 several times in a round -trip time of 1 source (20.2 ms rtt) whereas the other one gets the chance to adjust almost immediately. The file transfer times for the two connections are shown in Table 15. It is clear that even with large discrepancies in the round trip times of the two sources, the difference in their completion times is small (less than 7%). This is due to the efficient filtering of rate information, as well as the continuous time control system implemented in packet-pair.

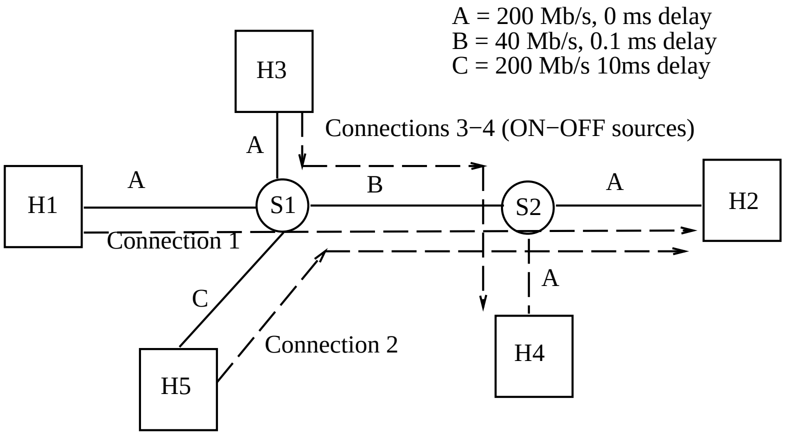

Figure 28: Configuration for Benchmark 9

Conn 1	Conn 2
1113	1188

Table 15: Benchmark 9: Completion times (in milliseconds)

9.4.10. Summary

This set of benchmarks shows that packet-pair is robust in a wide variety of scenarios. We have tested the scheme in scenarios with regulated cross traffic, unregulated cross traffic, small number of buffers, rapidly changing cross traffic and migrating bottlenecks. In all cases, the scheme is stable, and performs no worse than 20% of optimal (except for very small transfers, where the slow-start scheme causes bandwidth to be lost). While considerations of space do not allow us to present detailed results (such as the sequence number trace) for each scenario, the behavior even in cases of extreme stress was similar to that presented in Section 9.1 on the dynamics of packet-pair. Based on these results, we feel that the scheme is suitable for use in public data networks.

9.5. Packet Video Using Packet-pair

In the final test of packet-pair, we used it to carry MPEG compressed video over a simulated network. The experimental method is as in [30], where software coded MPEG is carried through a simulated network with 8 active sources, limited buffering and sudden changes in available bandwidth. The feedback signals from the flow control algorithm are used to modify the quality of the compression algorithm. Thus, on the onset of congestion, the coder is informed, and the subsequent data rate is decreased. Of course, this results in a loss of quality, but the quality degradation is even worse if packet losses occur, and there is no subsequent decrease in sending rate, as would happen with open loop flow control through a leaky bucket. The results of this experiment showed that the perceptual quality of the video stream carried by packet-pair in a congested network are as good as a hop-by-hop congestion control scheme that uses explicit per-virtual circuit rate feedback. Since packet-pair does not use any explicit rate information, the advantage is obvious.

10. Extensions to Packet-Pair

In this section, we discuss some extensions of the packet-pair approach to deal with different environments. The first subsection discusses how packet-pair would fit in ATM environment, Section 10.2 outlines how packet-pair and multicast might interact. Section 10.3 considers the interaction of window flow control with packet-pair rate-based flow control. Finally, Section 10.4 outlines an implementation of packet-pair that runs at the receiver instead of the sender.

10.1. Packet-pair and ATM

The packet-pair scheme is designed to work well in ATM environments. We propose to place packet-pair at the transport layer of a native-mode ATM protocol stack [36]. User data would be handed by the session layer to the transport layer using the transmission queue described in Section 6. The transport layer would then use packet-pair to send out pairs of AAL5 frames. The peer transport layer would send acknowledgements that could then be used to do rate control on the transmission queue.

Note that packet-pair works well in a Fair Queueing environment. Fair Queueing tries to emulate bit-by-bit round robin scheduling, which is closely approximated by cell-by-cell round robin at high transmission speeds. Thus, the rate probing technique works well despite segmentation and reassembly done below the AAL.

Rate probing is effective only when all the queueing points are round-robin like, and there is no multiplexing of data streams in the network. There are high-speed ATM networks already in existence that implement round-robin service [19, 51]. Further, the proposed native-mode ATM stack does not do any multiplexing so as to preserve per-virtual circuit Quality of Service constraints. Thus, we feel that these two restrictions could easily be satisfied by other ATM networks, making it feasible to implement packet-pair in B-ISDN environments.

10.2. Packet-pair in a Multicast Environment

While the discussion so far has been in the context of a point-to-point data transfer, packet-pair can be used in a multicast environment, as long as the acknowledgments from each endpoint can be demultiplexed using some identifier (such as the AAL 3/4 MID field). Given this, it is trivial to determine the slowest link and send data at that rate. The timeout and retransmission strategy use only the information contained in the acknowledgments, and by maintaining a per-destination retransmission queue, we believe that a flow control scheme for multicast networks can be developed.

10.3. Interaction with Window Flow Control

Note that our control system does not give us any guarantees about the shape of the buffer size distribution. Hence, there is a non-zero probability of packet loss. In many applications, packet loss is undesirable. It requires endpoints to retransmit messages, and frequent retransmissions can lead to congestion. Thus, it may sometimes be desirable to guarantee zero packet This can be arranged by having a window flow control algorithm operating simultaneously with the rate-based flow control algorithm described here.

In such a scheme, the rate-based flow control provides us a ‘good’ operating point which is the setpoint that the user selects. In addition, the source has a limit on the number of packets it could have outstanding (the window size), and every server on its path reserves at least a window’s worth of buffers for that conversation. This assures us that even if the system deviates from the setpoint, the system does not lose packets and possible congestive losses are completely avoided.

Note that by reserving buffers per conversation, we have introduced reservations into a network that we earlier claimed to be reservationless. However, our argument is that strict *bandwidth* reservation leads to a loss of statistical multiplexing. As long as no conversation is refused admission due to a lack of buffers, statistical multiplexing of bandwidth is not affected by buffer reservation, and the multiplexing gain is identical to that received in a network with no buffer reservations. Thus, with large cheap memories, we claim that it will be always be possible to reserve enough buffers so that there is no loss of statistical multiplexing.

To repeat, we use rate-based flow control to select an operating point, and window-based flow control as a conservative cut-off point. In this respect, we agree with Jain that the two forms of flow control are *not* diametrically opposed, but in fact can work together [27].

The choice of window size is critical. Using fixed sized windows is usually not possible in high speed networks, where the bandwidth-delay product, and hence the required window can be large (of the order of hundreds of kilobytes per conversation). In view of this, the adaptive window allocation scheme proposed by Hahne *et al* [23] is attractive. In their scheme, a conversation is allocated a flow control window that is always larger than the product of the allocated bandwidth at the bottleneck, and the round trip propagation delay. So, a conversation is never constrained by the size of the flow control window. A signaling scheme dynamically adjusts the window size in response to changes in the network state. We believe that their window-based flow control scheme is complementary to the rate-based flow control scheme proposed in this paper.

10.4. Receiver-based Control

The discussion thus far places the rate control at the sender. While this is reasonable for data-transfer type of applications, we would like to dispense with this for one-way transfers such as for MPEG-compressed video streams. In this case, we would like to implement packet-pair at the receiver. This can be done by making three extensions to packet-pair. First, the sender places a header containing its current state in each TPDU. The receiver thus receives continuous updates about sender state. Second, the receiver measures the inter-packet spacing, and computes the appropriate sending rate. Third, the receiver sends periodic messages to the sender with the optimal sending rate. The frequency of these messages determines the effectiveness of the control. Based on the state exchange work by Sabnani et al [50], we feel that a fairly parsimonious state exchange protocol can give reasonably good performance. However, this is still an area for interesting future work.

11. Limitations

The main limitation of our approach is that it restricts the scheduling discipline to be round-robin-like. Unfortunately, the vast majority of current networks implement the first-come-first-served discipline. However, while our rate probing technique does not work in FCFS networks, other parts of the scheme, such as for optimal retransmission, do carry over to FCFS networks.

Other than this restriction, we believe that our work is quite general in its treatment of feedback based congestion control for ABR traffic. Given the benefits of round-robin-like service for data service, we feel that this work will gain importance in the future.

12. Related Work

This work is related to several threads of research in congestion control. Broadly speaking, congestion control can be classified as open-loop or closed-loop. Closed loop (or feedback) schemes, such as packet-pair, are generally subdivided into dynamic window or rate-based schemes.

12.1. Dynamic Window Schemes

The first two dynamic window schemes to attract wide attention were those proposed for TCP by Jacobson and Karels [26], and for DEC Network Architecture by Jain, Ramakrishnan and Chiu [48]. Both schemes made seminal contributions to this area, but are best suited for FCFS networks with small bandwidth-delay products. Neither scheme exploits the linearization property of round-robin service, and because they take control actions only once every few round-trip times, are unsuitable for networks with a large bandwidth-delay product. Further, as this product increases, the startup algorithms embodied in the algorithms make both packet losses and small data transfers expensive [35]. The Jacobson-Karels approach uses packet losses as an indication of congestion. As a consequence, the congestion indicator is seen only when the congestion has already taken a toll. This reduces the achievable goodput. Both schemes assume that all sources are cooperative, that is, they respond correctly to congestion signals. Thus, a non-cooperative source (as in Benchmark 7) will cause a compliant source to reduce its own load to zero. This is clearly undesirable for public networks!

A dynamic window scheme that uses out-of-band signalling to achieve zero packet loss while minimizing buffer usage has been proposed by Hahne, Kalmanek and Morgan [23]. This scheme is efficient in its use of bandwidth and buffers, but uses a fairly complicated signaling protocol. In contrast, packet-pair achieves good performance using passive probes.

Mitra, Seery et al have proposed an adaptive window scheme for networks with large bandwidth-delay products [42, 43]. However, their results depend on the cross traffic being Poisson, which may not unlikely to hold true for current or future networks [39] Further, they do not make use of predictive control, so propagation delays lead their control to oscillate even under stable conditions.

Dynamic window schemes have been mathematically analyzed by Bolot [6], Fendick, Rodrigues and Weiss [15] and Mukherjee and Strikwerda [45]. These analyses give some insight into the performance of generic dynamic window schemes.

12.2. Rate-based Flow Control

Several rate-based flow control schemes have been proposed in the literature. One of the earliest scheme was for NETBLT [11], which was based on heuristics, and not thoroughly studied. In NETBLT, the transmitter sends data at some rate for a few round trip times, and the receiver clocks the data to see if it was received at that rate. If it did, the sender increases its rate, else it cuts back, This suffers from the problem that an increase in available capacity is known only by inducing congestion. We avoid this by using packet-pair probes.

An early approach to hop-by-hop predictive control was proposed by Ko, Mishra and Tripathi [38]. This work shares several objectives with packet-pair, and was independently developed at around the same time. The differences with our approach are that their congestion indicator is explicitly communicated to the source, and the smoothing of these estimates uses an exponential averager with an α chosen *a priori*, instead of using a fuzzy predictor. Further, they appeal primarily to intuitive heuristics, and do not use a formal control-theoretic model, to develop their control.

Williamson has proposed the Loss-Load scheme [60, 61] that uses the throughput-versus-loss curve to compute an optimal sending rate. This approach has numerous lacunae. It ignores system considerations, such as the the fact that monitoring each connection at each switch poses a rather considerable burden on the switch controller. Also, a source may lose packets even if it is sending below its fair share. Finally, the sender computes its rate by solving an equation of the kth degree, where the overall loss rate is $1/k + 1$. For a loss rate of $10 - 3$, solving such an equation in real-time is impossible given current technology. We feel that packet-pair is better attuned to current networking realities.

Kanakia and Mishra have proposed a hop-by-hop congestion control scheme based on predictive control [29]. This is also similar to our approach. Again, their scheme requires a switch to monitor all the conversations, and there is also the overhead of switch-to-switch transfer of rate information. We have found that the hop-by-hop scheme does perform better than packet-pair, but not by very much [31]. Given that packet-pair uses only passive probing, we feel that the loss in performance may be worth it.

Low et al [41] have studied a scheme with one or more bits of explicit feedback information from each switch. They estimate the network state using these bits, then do a state prediction using Kumar and Varaiya's one-step ahead predictor assuming that the system state density function is Gaussian. Simulations are used to confirm the design. Their scheme may be a viable alternative to ours, but their simulations are not detailed or diverse enough to verify the correctness of their assumptions. Further, they have not shown that their scheme is stable, nor have they taken implementation considerations into account.

Browning has proposed a scheme based on the concept of disturbance accomodation control [9]. His scheme derives a control law very similar to that we derive in Section 4, thus validating some of our work. However, he has not done detailed simulations to confirm his control law.

Park has proposed the 'Warp' congestion control scheme [47]. In this scheme, the receiver detects network state by observing arrival times of data packets from the sender, and uses this to determine an appropriate sending rate. However, this work suffers from an error in the mathematical model (Equation 6 in [47]), since according to this equation, the bottleneck queue size may be negative, which is clearly impossible. Since the entire control is based on this equation, we doubt the correctness of his approach.

Theoretical analyses of rate-based flow control for generic schemes have been done by Benmohamed and Meerkov [4] and by Fendick and Rodrigues [16].

12.3. Studies of Ensembles of Controlled Systems

One body of work has considered the dynamics of a system where users update their sending rate either synchronously or asynchronously in response to measured round trip delays, or explicit congestion signals, for example in References [5, 7, 8, 13, 52]. These approaches typically assume Poisson sources, availability of global information, a simple flow update rule, and exponential servers. We do not make such assumptions. Further, they deal with the dynamics of the entire system, with the sending rate of all the users explicitly taken into account. In contrast, we consider a system with a single user, where the effects of the other users are considered as a system ‘noise’. Also, in our approach, each user uses a rather complex flow update rule, based in part on fuzzy prediction, and so the analysis is not amenable to the simplistic approach of these authors.

12.4. Optimal Flow Control

A control theoretic approach to individual optimal flow control was described originally by Agnew [1] and since extended by Filipiak [17] and Tipper *et al* [55]. In their approach, a conversation is modeled by a first order differential equation, using the fluid approximation. The modeling parameters are tuned so that, in the steady state, the solution of the differential equation and the solution of a corresponding queueing model agree. While we model the service rate at the bottleneck μ as a random walk, they assume that the service rate is a non-linear function of the queue length, so that $\mu = G(n_b)$, where $G(.)$ is some nonlinear function. This is not true for a RAS, where the service rate is independent of the queue length. Hence, we cannot apply their techniques to our problem.

Vakil, Hsiao and Lazar [56] have used a control-theoretic approach to optimal flow control in double-bus TDMA local-area integrated voice/data networks. However, they assume exponential FCFS servers, and, since the network is not geographically dispersed, propagation delays are ignored. Their modeling of the service rate μ is as a random variable as opposed to a random walk, and though they propose the use of recursive minimum mean squared error filters to estimate system state, the bulk of the results assume complete information about the network state. Vakil and Lazar [57] have considered the the design of optimal traffic filters when the state is not fully observable, but the filters are specialized for voice traffic.

Robertazzi and Lazar [49] and Hsiao and Lazar [25] have shown that under a variety of conditions, the optimal flow control for a Jacksonian network with Poisson traffic is *bang-bang* (approximated by a window scheme). It is not clear that this result holds when their strong assumptions regarding memorylessness of service are removed.

12.5. Summary

In summary, we feel that our approach is substantially different from others described in the literature. Only one other scheme has concentrated on the problem of feedback flow control in networks of round-robin-like servers [23], and this uses a rather complex signaling protocol. All the other schemes have either not assumed any particular scheduling discipline, or assumed it to be FCFS, and thus do not exploit the linearization property of round-robin service.

Our use of a packet-pair to estimate the system state is unique, and this estimation is critical in enabling the control scheme. We have described two provably stable rate-based flow control schemes as well as a novel estimation scheme using fuzzy logic. Numerous practical concerns in implementing the scheme have been addressed. We have proposed novel schemes for startup, optimal retransmission and buffer setpoint probing. Finally, we have performed an exhaustive simulation study of the scheme, and shown it to be useful in a wide variety of scenarios.

We have shown that packet-pair responds quickly and cleanly to changes in network state. Unlike some current flow control algorithms (DECbit and Jacobson’s modifications to 4.3 BSD [26, 48]), the system behaves well in situations where the bandwidth-delay product is large, even if the cross traffic is misbehaved or bursty [34]. Implementation and tuning of the algorithm is straightforward, unlike the complex and ad-hoc controls in current flow control algorithms. Even in complicated scenarios, the dynamics are simple to understand and manage. Packet-pair behaves well even under stress, and, more importantly, it is simple to implement and tune. These are not fortuitous, rather, they reflect the theoretical underpinnings of the approach.

13. Acknowledgements

My thanks to Scott Shenker for introducing me to the area of congestion control and for guiding earlier attempts to build rate-based flow control schemes. I would like to thank Domenico Ferrari for his support, constant encouragement and keen questioning. Subsequently, numerous valuable discussions with Chuck Kalmanek, Hemant Kanakia, Sam Morgan and Partho Mishra motivated me to explore the limits of this scheme.

The use of packet-pairs to probe the bottleneck service rate was independently suggested by Ashok Agrawala and Samar Singh. The use of a noise variable to model the bottleneck service rate was suggested by G. Srinivasan. P.S. Khedkar collaborated on the design of the fuzzy controller. The benchmarks in Section 9.4 were jointly designed with Hemant Kanakia and Partho Mishra. The public data network scenario in Section 9.3 was suggested by Sam Morgan. Amy Reibman supplied codec data for the packet-video experiment in Section 9.5 and also converted my results into a videotape. The helpful advice and criticism of Sally Floyd, Debasis Mitra, R.P. Singh, M. Tomizuka, Pravin Varaiya and the anonymous refrees of ACM Sigcomm and ACM Transactions on Computer Systems on several aspects of the work are much appreciated.

14. References

1. C. Agnew, Dynamic Modeling and Control of Congestion-prone Systems, *Operations Research 24*, 3 (1976), 400-419.
2. B. D. O. Anderson and J. B. Moore, *Linear Quadratic Methods*, Prentice Hall, 1990.
3. D. Anick, D. Mitra and M. M. Sondhi, Stochastic Theory of a Data-handling System with Multiple Sources, *Bell System Technical Journal 61* (1982), 1871-1894.
4. L. Benmohamed and S. M. Meerkov, Feedback Control of Congestion in Store-and-forward Networks: the Case of a Single Congested Node, *ACM/IEEE Trans. on Networking (to appear)*, 1994.
5. K. Bharath-Kumar and J. M. Jaffe, A New Approach to Performance-Oriented Flow Control, *IEEE Trans. on Communication COM-29*, 4 (April 1981), 427-435.
6. J. Bolot and A. U. Shankar, Analysis of a Fluid Approximation to Flow Control Dynamics, *Proc. IEEE INFOCOM '92*, 1992, 2398-2407.
7. A. D. Bovopoulos and A. A. Lazar, Decentralized Algorithms for Optimal Flow Control, *Proc. 25th Allerton Conference on Communications Control and Computing*, October 1987. University of Illinois, Urbana-Champaign.
8. A. D. Bovopoulos and A. A. Lazar, Asynchronous Algorithms for Optimal Flow Control of BCMP Networks, Tech. Rpt. WUCS-89-10, Washington University, St. Louis, MO, February 1989.
9. D. W. Browning, Flow Control in High-Speed Communication Networks, *IEEE Trans. Comm. (to appear)*, 1994.
10. R. Caceres, P. B. Danzig, S. Jamin and D. J. Mitzel, Characteristics of Application Conversations in TCP/IP Wide-Area Internetworks, *Proc. ACM SigComm 1991*, September 1991.
11. D. D. Clark, M. L. Lambert and L. Zhang, NETBLT: A Bulk Data Transfer Protocol, RFC-998, Network Working Group, March 1987.
12. A. Demers, S. Keshav and S. Shenker, Analysis and Simulation of a Fair Queueing Algorithm, *Journal of Internetworking Research and Experience*, September 1990, 3-26;. also Proc. ACM SigComm, Sept. 1989, pp 1-12..
13. C. Douligeris and R. Majumdar, User Optimal Flow Control in an Integrated Environment, *Proc. of the Indo-US Workshop on Systems and Signals*, January 1988. Bangalore, India.
14. A. E. Ekberg, D. T. Luan and D. M. Lucantoni, Bandwidth Management: A Congestion Control Strategy for Broadband Packet Networks: Characterizing the Throughput-Burstiness Filter, *Proc. ITC*

Specialist Seminar, Adelaide, 1989, paper no. 4.4.

15. K. W. Fendick, M. A. Rodrigues and A. Weiss, Analysis of a Rate-Based Control Strategy with Delayed Feedback, *Proc. ACM SigComm '92*, 1992.
16. K. W. Fendick and M. A. Rodrigues, An Adaptive Framework for Dynamic Access to Bandwidth at High Speeds, *Proc. ACM SigComm '93*, 1993.
17. J. Filipiak, *Modelling and Control of Dynamic Flows in Communication Networks*, Springer-Verlag, 1988.
18. A. G. Fraser, Designing a Public Data Network, *IEEE Communications Magazine*, October 1991, 31-35.
19. A. G. Fraser, C. R. Kalmanek, A. Kaplan, W. T. Marshall and R. C. Restrick, Xunet 2: A Nationwide Testbed in High-Speed Networking, *Proc. IEEE INFOCOM 1992*, May 1992.
20. E. Gafni and D. Bertsekas, Dynamic Control of Session Input Rates in Communication Networks, *IEEE Trans. on Automatic Control 29*, 11 (1984), 1009-1016.
21. A. G. Greenberg and N. Madras, Comparison of a Fair Queueing Discipline to Processor Sharing, in *Performance '90; Proceedings of the 14th IFIP WG 7.3 International Symposium on Computer Performance Modelling, Measurement and Evaluation*, North Holland, Edinburgh, Scotland, September 1990, 193-207.
22. A. G. Greenberg and N. Madras, How Fair is Fair Queueing?, *Journal of the ACM 3*, 39 (1992).
23. E. L. Hahne, C. R. Kalmanek and S. P. Morgan, Fairness and Congestion Control on a Large ATM Data Network with Dynamically Adjustable Windows, *13th International Teletraffic Congress*, Copenhagen , June 1991.
24. E. L. Hahne, Round Robin Scheduling for Fair Flow Control in Data Communication Networks, LIDS-TH-1631, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Cambridge, MA 02139.
25. M. Hsiao and A. A. Lazar, Optimal Flow Control of Multi-Class Queueing Networks with Partial Information, *IEEE Transactions on Automatic Control 35*, 7 (July 1990), 855-860.
26. V. Jacobson, Congestion Avoidance and Control, *Proc. ACM SigComm 1988*, August 1988, 314-329.
27. R. Jain, Myths About Congestion Management in High-Speed Networks, Technical Report-726, Digital Equipment Corporation, October 1990.
28. C. R. Kalmanek, H. Kanakia and S. Keshav, Rate Controlled Servers for Very High Speed Networks, *Proc. Globecom 1990*, December 1990, 300.3.1-300.3.9.
29. H. Kanakia and P. P. Mishra, A Hop by Hop Rate-Based Congestion Control Scheme, *Proc. ACM SigComm*, 1992.
30. H. Kanakia, P. P. Mishra and A. Reibman, An Adaptive Congestion Control Scheme for Real-Time Packet Video Transport, *Proc. ACM SigComm*, 1993.
31. H. Kanakia, S. Keshav and P. P. Mishra, A Comparision of Congestion Control Schemes, *Proc. Fourth Annual Workshop on Very High Speed Networks*, Baltimore, Maryland, March 1993.
32. S. Keshav, A Control-Theoretic Approach to Flow Control, *Proc. ACM SigComm 1991*, September 1991.
33. S. Keshav, A. K. Agrawala and S. Singh, Design and Analysis of a Flow Control Algorithm for a Network of Rate Allocating Servers, in *Protocols for High Speed Networks II*, Elsevier Science Publishers/North-Holland, April 1991.
34. S. Keshav, Congestion Control in Computer Networks, *PhD thesis, Comp. Sci. Dept. Tech. Rpt. 91/649*, University of California, Berkeley, August 1991.
35. S. Keshav, Flow Control in High Speed Networks with Long Propagation Delays, *Proc. INET'92*, June 1992.
36. S. Keshav and H. Saran, Semantics and Implementation of a Native-Mode ATM Protocol Stack, *Submitted to Infocom '95*, August 1994.

37. P. S. Khedkar and S. Keshav, Fuzzy Prediction of Timeseries, Proc. IEEE Conference on Fuzzy Systems-92, March 1992.

38. K. Ko, P. P. Mishra and S. K. Tripathi, Predictive Congestion Control in High-Speed Wide-Area Networks, in *Protocols for High Speed Networks II*, M. J. Johnson (editor), Elsevier Science Publishers/North-Holland, April 1991.

39. W. E. Leland, M. S. Taqqu, W. Willinger and D. V. Wilson, On the Self-Similar Nature of Ethernet Traffic, *Proc. ACM SigComm '93*, 1993.

40. S. Low and P. P. Varaiya, A Simple Theory of Traffic and Resource Allocation in ATM, *Conference Record, GlobeCom 1991*, December 1991.

41. S. Low, N. Plotkin, M. K. Wong and J. Yee, On the Usefulness of Explicit Congestion Notification in High Speed Networks, *2nd International Conference on Telecommunication Systems, Modeling and Analysis*, March 1994.

42. D. Mitra and J. B. Seery, Dynamic Adaptive Windows for High Speed Data Networks: Theory and Simulations , *Proc. ACM SigComm 1990,* September 1990, 30-40.

43. D. Mitra, Asymptotically Optimal Design of Congestion Control for High Speed Data Networks, *IEEE Trans. on Communications 40*, 2 (Feb 1992), 301-311.

44. S. P. Morgan, Queueing Disciplines and Passive Congestion Control in Byte-Stream Networks, *Proc. IEEE INFOCOM '89*, 1989, 711-729.

45. A. Mukherjee and J. C. Strikwerda, Analysis of Dynamic Congestion Control Protocols - A Fokker-Planck Approximation, *Proc. ACM SigComm '91*, September 1991.

46. A. K. Parekh, A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks, *PhD thesis*, Massachusetts Institute of Technology, February 1992.

47. K. Park, Warp Control: A Dynamically Stable Congestion Protocol and its Analysis, *Proc. ACM SigComm '93*, 1993.

48. K. K. Ramakrishnan and R. Jain, A Binary Feedback Scheme for Congestion Avoidance in Computer Networks, *ACM Trans. on Comp. Sys. 8*, 2 (May 1990), 158-181.

49. T. G. Robertazzi and A. A. Lazar, On the Modeling and Optimal Flow Control of the Jacksonian Network, *Performance Evaluation 5* (1985), 29-43.

50. K. K. Sabnani and A. N. Netravali, A High Speed Transport Protocol for Datagram/Virtual Circuit Networks, *Proc. ACM SigComm 1989,* September 1989, 146-157.

51. H. Saran, S. Keshav and C. R. Kalmanek, A Scheduling Discipline and Admission Control Policy for Xunet II, *Multimedia Systems Journal 2*, 3 (September 1994).

52. S. Shenker, A Theoretical Analysis of Feedback Flow Control, *Proc. ACM SigComm 1990*, September 1990, 156-165.

53. S. Singh, A. K. Agrawala and S. Keshav, Deterministic Analysis of Flow and Congestion Control Policies in Virtual Circuits, Tech. Rpt.-2490, University of Maryland, June 1990.

54. A. S. Tanenbaum, in *Computer Networks*, Prentice Hall, Englewood Cliffs, NJ, 1981.

55. D. Tipper and M. K. Sundareshan, Numerical Methods for Modeling Computer Networks under Nonstationary Conditions, *JSAC 8*, 9 (December 1990).

56. F. Vakil, M. Hsiao and A. A. Lazar, Flow Control in Integrated Local Area Networks, *Performance Evaluation 7*, 1 (1987), 43-57.

57. F. Vakil and A. A. Lazar, Flow Control Protocols for Integrated Networks with Partially Observed Traffic, *IEEE Transactions on Automatic Control 32*, 1 (1987), 2-14.

58. J. G. Waclawsky and A. K. Agrawala, Dynamic Behavior of Data Flow within Virtual Circuits, Comp. Sci.-Tech. Rpt.-2250, University of Maryland , May 1989.

59. J. G. Waclawsky, Window Dynamics, *PhD Thesis*, University of Maryland, College Park, May 1990.

60. C. L. Williamson and D. R. Cheriton, Loss-Load Curves: Support for Rate-Based Congestion Control in High-Speed Datagram Networks, *Proc. ACM SigComm 1991*, September 1991.

61. C. L. Williamson, Optimizing File Transfer Response Time Using the Loss-Load Congestion Control Mechanism, *Proc. ACM SigComm 1993*, 1993.

62. L. A. Zadeh, Fuzzy Sets, *Journal of Information and Control 8* (1965), 338-353.

63. L. A. Zadeh, Outline of a New Approach to the Analysis of Complex Systems and Decision Processes, *IEEE Trans. on Systems, Man and Cybernetics*, 1973, 28-44.

64. L. Zhang, Why TCP Timers Don't Work Well, *Proc. Sigcomm 1986*, 1986, 397-405.

65. L. Zhang, A New Architecture for Packet Switching Network Protocols, *PhD thesis*, Massachusetts Institute of Technology, July 1989.

66. L. Zhang, S. Shenker and D. D. Clark, Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic, *Proc. ACM SigComm 1991*, September 1991.

67. H. J. Zimmerman, in *Fuzzy Set Theory and its Applications*, Kluwer Academic Publishers, 1985.