Design and Analysis of a Flow Control Algorithm for a Network of Rate Allocating Servers

Srinivasan Keshav[a], Ashok K. Agrawala[b] and Samar Singh[c]

[a]Computer Science Division, Department of EECS, University of California, Berkeley, Berkeley, CA 94720, USA

[b]Department of Computer Science, University of Maryland, College Park, MD 20742, USA

[c]Department of Computer Science and Engineering, Indian Institute of Technology, New Delhi - 110016, India

Abstract

Packet-switched networks where the router queue-service discipline is based on round-robin can be modeled as networks of rate allocating servers [2, 3, 13]. We model a virtual circuit in such networks as a sequence of D/D/1 queues, prove some interesting properties, and use this simple model to derive a new flow control scheme that we call 2P. 2P uses a novel technique based on short packet bursts to estimate the service rate of the circuit and to adapt the sending rate as the network state changes. We describe an implementation of 2P and analyze its response to network transients. Simulations of the scheme compare it with some well known flow control schemes and show that it performs well in deterministic scenarios.

1. INTRODUCTION

Currently, most computer networks with a connectionless network layer have routers that obey a first-come-first-served (FCFS) queueing discipline. Thus, existing transport layer flow control protocols are optimized for such networks. Recently, some new service disciplines based on round-robin service have been proposed [2, 3, 13]. These disciplines have

---

[a] Supported in part by AT&T Bell Laboratories and in part by National Science Foundation Infrastructure Grant number CDA-8722788 under the Mammoth Project.

[b] Partly supported by RADC and the Defense Advanced Research Projects Agency under contract F30602-90-C-0010 to UMIACS at the University of Maryland.

[c] Partly supported by a UNDP traveling fellowship under the E&R Network program at IIT Delhi.

several interesting features, such as the automatic restriction of misbehaved sources, provision for resource reservation, and the separation of throughput and delay requirements. Networks composed of such servers enable novel flow control algorithms. In this paper, we present the design and analysis of such a new flow control scheme called Packet-Pair or 2P.

Section 2 of this paper briefly describes unifying notions underlying the new service disciplines. We then present a simple deterministic model for networks of such servers and prove some interesting properties. This motivates the design of 2P in Section 4. 2P is analyzed in Section 5 and Section 6 presents simulation results. The discussion in Section 7 summarizes our results.

2. FAIR QUEUEING, VIRTUAL CLOCK AND EARLIEST-DUE-DATE SCHEDULING

We focus attention on the queue service discipline in the routers of a communication network. Three new service disciplines have been presented in the recent past, Fair Queueing (FQ) [2], Virtual Clock (VCL) [13] and Earliest-Due-Date (EDD) [3]. We claim that these disciplines are isomorphic to each other since they all attempt to service an incoming stream of packets in a virtual Time Division Multiplexed (TDM) manner. All three schemes assign incoming packets a priority index that corresponds to the service completion time of the packet *had the server been doing TDM*. Service is in order of increasing priority index, and this has the effect of doing TDM without the attendant inefficiencies. The priority index is called the finish number (in FQ), the virtual service time (in VCL) or the deadline (in EDD), but in each case, it serves the same purpose. Due to this isomorphism, we will call all such servers Rate Allocating Servers (RAS); the reason for this name will shortly become apparent.

One outcome of servicing packets in order of their priority index is that it *decouples* packets from different conversations[1]. All packets from the same conversation are allocated monotonically increasing priority indices, and as long as that conversation is active, to a first approximation, its index is independent of the indices allocated to any other conversation. Put another way, since the indices define the service order, each conversation gets a service rate that is independent of the service rate of any other conversation[2]. Thus, a RAS can be modeled as a server that services each of the incoming conversations at some service rate, and the service rate for some conversation is approximately independent of the service rate for any other conversation.

While VCL and EDD scheduling disciplines were originally presented in the context of a connection-oriented network layer, in our treatment, we will consider their behavior in connectionless network layers. This raises an important point. In connection-oriented networks, during call set-up, a conversation specifies a desired service rate to the servers that lie in its path. This information allows each server to prevent overbooking of its bandwidth and the

---

A conversation is a (source, destination) pair; see the discussion in [2]. We will interchangeably use the term ‘virtual circuit’ or VC.

We use ‘service rate’ to denote the inverse of the deterministic service time, as opposed to a time average.

service rate that a conversation receives is constant. However, in connectionless networks, a server is not allowed to refuse any conversations, and so the bandwidth could be overbooked. We assume that in such a situation, a RAS will divide bandwidth in the same way as a FQ server, that is, equally amongst the currently active conversations. Thus, the rate allocated to a conversation could change with time.

Specifically, there are two reasons why a RAS might change the rate at which it services a conversation:

1) The number of conversations served can vary with time. If the service rate of any conversation is defined to be inversely proportional to the number of active conversations, then the service rate of a conversation changes as the total number of active conversations changes.

2) If some conversation has a low service rate, or has a bursty arrival pattern, there are intervals where it does not have any packets to send, and the RAS will treat such a conversation as idle. Thus, the effective number of active conversations decreases, and the rate allocated to the other conversations increases. When the traffic on that conversation resumes, the allocated rate to each conversation will again decrease.

However, this behavior is much better than that of an FCFS server. In an 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 service rate varies much more rapidly. We will make the important assumption that in RAS networks of interest, the rate of service of a conversation varies infrequently as compared to its round-trip-time delay.

Network model

Since RAS servers provide conversations with nearly fixed service rates, as a simplification, we model a RAS server deterministically, that is, as a server that provides each incoming conversation with a constant service rate. While this is a major simplification, we believe that is appropriate as a first step. Further, congestion depends on the transient behavior of the network, and while existing stochastic models do not deal adequately with transients, they are easily handled by a deterministic analysis. Thus, a deterministic approach seems appropriate, and we use it here to analyze the transient response of 2P. We do realize the inadequacy of the model, and are currently investigating a stochastic extension to the model.

Other authors too have implicitly or explicitly used some deterministic analysis when discussing the performance of transport protocols [4, 5, 8] For example, Jain has explicitly modeled a virtual circuit as a series of D/D/1 queues in [6]. Waclawsky and Agrawala have developed and analyzed a similar deterministic model for studying the behavior of window protocols on a virtual circuit [10, 12].

We model a virtual circuit in a RAS network as a series of servers (routers or switches) connected by links. A packet is a point object that starts out from the source and traverses the links and servers until it reaches the destination. The time taken to traverse a link is zero, while the time taken to get service (at each server) is finite but deterministic.

We number the servers in the path of the VC as 1,2,3..., and the source is numbered 0. If the *i*th server is idle when a packet arrives, the time taken for service is s_i, and the service rate is defined to be $\rho_i = 1/s_i$. If there are other packets from that VC at the server, the packet waits for its turn to get service (we assume a FCFS queueing discipline per VC). We assume a work-conserving discipline, which implies that a server will never be idle whenever

it has a packet ready to be served. The source sending rate is denoted by ρ_0 and the source is assumed to send data spaced apart exactly $s_0 = 1/\rho_0$ time units apart.
We define

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

to be the *bottleneck* service time in the circuit, and b is the index of the bottleneck server. The ordered set SL is defined as $\{ sl_i \in 0, 1, ..., b \mid \text{if } j > i, \rho_{sl_j} < \rho_{sl_i} \}$, that is, SL is the ordered subset of successively strictly slower servers from the source to the bottleneck. Each element of SL is called a *rate-throttle*. This notation is summarized at the end of the paper in Section 10.

To complete the picture, we assume another set of links and servers that constitute a return path from the destination back to the server. This is the path taken by acknowledgment packets (acks). We assume that every packet is acknowledged, so the destination is just another server, and the returning ack is modeled as the same packet looping back to the source. Strictly speaking, this assumption is not required, but we make it for the sake of convenience.

We now prove some lemmas about the properties of such VCs. Similar results and a more detailed analysis can be found in [11, 12].

Lemma 1 : (Basic lemma) Consider data arriving at an initially idle server *j* at a rate *r*.

(a) If $r \le \rho_j$, there is no queueing at *j*, and the departure rate from server *j* is *r*.

(b) If $r > \rho_j$, there is queueing at *j*, and the departure rate from server *j* is ρ_j.

Proof :(a) Initially, since the server is idle, its queue is empty. If the first packet arrives at time t_0, it will depart at time $t_0 + s_j$. Packets in the arriving stream are spaced $1/r$ time units apart. Thus, the next packet arrives at time $t_0 + 1/r$. Since $\rho_j \ge r$, $1/r \ge 1/\rho_j$ and $t_0 + 1/r \ge t_0 + s_j$, so the next packet arrives only after the first one has left. Thus, there is no queueing at the server. Simple induction on the sequence number of the arriving packet gives us the result on queueing.

The departure rate of the packets is constrained only by the arrival rate, and hence the output stream from the server has a rate *r*.

(b) Since the departure of the first packet happens after the arrival of the next packet, the second packet will be queued in the server. If there is a queue already, and a packet arrives before the departure of the previous packet, it will only add to the queue. Induction on the packet sequence number gives us the queueing result.

Since the departure stream from the server has a inter-packet spacing of s_j, the output stream is at rate ρ_j.

Lemma 2 : (Composition) Consider two adjacent servers *j* and $j+1$. If data enters server *j* at a rate *r* such that $\rho_j \ge r > \rho_{j+1}$ queueing occurs only at server $j+1$.

Proof :Since $r \le \rho_j$, there is no queueing at server *j* (Lemma 1). Hence, the departure rate of packets from server *j*, as well as the arrival rate at server $j+1$ is *r*. Since $r > \rho_{j+1}$, there is queueing at server $j+1$ (Lemma 1).

Lemma 3 : (Single bottleneck) If data enters a segment of the VC numbered k, k+1, .. L, at a rate r such that $\rho_L < r < \rho,\ \rho \in \{\rho_k, \rho_{k+1}, \cdots, \rho_{L-1}\}$, then queueing occurs only at L.

Proof :Since $r < \rho_k$, there is no queueing at server k and the departure rate from server k is r (Lemma 1). We can thus delete server k from the chain, and repeat the argument for the servers k+1, k+2, ... L. For the servers L-1, L, we use Lemma 2 to get the desired result.

Lemma 4 : (Chain of rate-throttles) Queueing can happen only in elements of SL.

Proof: Break up the server chain 1,2, ... , b into sub-chains 1,2, ... sl_1; $sl_1 + 1, \cdots, sl_2$; ...; such that only $sl_i \in$ SL. Consider the first such chain. If $\rho_0 < \rho_{sl_1}$, there is no queueing at sl_1. Hence, to get the worst possible scenario, we assume that $\rho_0 > \rho_{sl_1}$. In that case, from Lemma 3, the only queueing at the first chain will be at sl_1 (if ρ_0 is very large, sl_1 could just be 1). By definition of SL, the departure rate from sl_1, ρ_{sl_1}, satisfies the requirements for Lemma 3, so there will be queueing at sl_2, and at no other node in that subchain. From an induction on the sequence number of the subchain, we get the desired result.

Lemma 5 : (Probing) If a source sends packets spaced s_0 time units apart, and $\rho_0 \geq \rho_b$, the acks will be received at the the source at intervals of s_b time units.

Proof :By definition of the bottleneck, and Lemmas 1 and 4, the departure rate of packets at the bottleneck is ρ_b. Since acks are created for each packet instantaneously, the acks will be spaced apart by s_b.

Define Δ_j to be $\rho_{sl_{j-1}} - \rho_{sl_j}$.

Lemma 6 : (Burst dynamics) If a source sends a burst of K packets at a rate $s_0 >> \rho_i$, for all i, then the queue at sl_j builds up at the rate Δ_j, reaches its peak at time $t_j = \sum_{i=0}^{j-1} s_i + \frac{K}{\rho_{sl_{j-1}}}$, and decays at the rate ρ_{sl_j}.

Proof : Consider the situation at sl_j. It receives packets at a rate $\rho_{sl_{j-1}}$, and serves them at the rate of ρ_{sl_j}. Thus, the queue builds up at the rate Δ_j. The queue reaches the maximum size when the last packet from the previous rate-throttle arrives. Since this is at a rate of $\rho_{sl_{j-1}}$, the time to receive K packets is $K/\rho_{sl_{j-1}}$. To this we add $\sum_{i=0}^{j-1} s_i$, which is the time for the first packet to arrive, to get the desired result. Finally, the queue will decay at the service rate of the rate-throttle, i.e. ρ_{sl_j}.

Note that in our model, it is not possible to have more than one bottleneck. While queueing may occur at more than one node, the service rate of the circuit is determined by the lowest indexed server with a service rate of ρ_b, and this will be the bottleneck.

3. RATE PROBING SCHEMES

How should one design a flow control scheme for a RAS network? Given the fact that in these networks a conversation will obtain a slowly time-varying service rate at the bottleneck server, a simple flow control scheme would be to *probe* the server to determine its current service rate for that conversation, and then send data at that rate. Sending it any slower would result in loss of throughput and any faster would result in queueing at the bottleneck. Thus, it is clear that we are interested mainly in what have been called rate-based flow control schemes [1]. Note that rate-based flow control schemes are explicitly enabled by RAS networks, as opposed to FCFS networks.

Rate based flow control

Our first attempt at designing a rate based flow control scheme modifies an idea described by Clark et al. for NETBLT [1]. If a source sends data at a rate ρ_s, and receives acknowledgments at a rate ρ_b, then our control scheme is:

if $\rho_s > \rho_b$, *decrease* ρ_s, *else increase it.*

The idea is that the rate at which acknowledgments are received is approximately the rate which the RAS has allocated to the conversation. We would like to match the sending rate to this rate.

The increase and decrease policies are multiplicative, that is the algorithm is

if $(\rho_s > \rho_b)$ $\rho_s = \alpha . \rho_s$ *else* $\rho_s = \beta . \rho_s$

where $\alpha < 1$ and $\beta > 1$. We expect that as the service rate changes, this adaptive scheme will converge on the new rate, and the system should stabilize at the correct rate.

However, there are a few problems. Note that a source cannot determine an increase in available capacity except by sending at a slightly increased rate and looking at the ack stream. Thus, a sudden large increase in the service rate can be adjusted for only after several round trip times. This is undesirable, particularly in high speed networks, where the bandwidth delay product can be large. Similarly, it takes a few round trip times to adjust to a sharp decrease in service rate. In the meantime, the bottleneck queue builds up. Since after a decrease, the source sends at very nearly the service rate, the built up queues never shrink, and the network becomes more prone to packet loss. Finally, the rate probe tends to push the network towards congestion, since the source is always trying an increased sending rate, until the rate can no longer be supported. These problems point to a need for a better rate control algorithm, such as 2P.

4. THE 2P SCHEME

We describe 2P in three stages. First, we describe the heuristic basis for the algorithm. Next, we give a full description, and finally present the implementation details.

4.1. Theory

2P is based on three observations, that we state informally:

> *If two packets are sent to a RAS at a rate faster than* ρ_b*, then the inter-ack spacing is* $s_b = 1/\rho_b$ *(Probing Lemma).*

If a source sends a pair of packets back-to-back, the spacing between the acknowledgments (acks) for these packets is an accurate probe of the service rate.

> *If a source is sending data to a RAS, and has a rate allocation* ρ_b *and a round trip propagation delay R, it operates at the knee of the network load-throughput curve when it has* $R.\rho_b$ *packets outstanding (and the pipeline is full)*[3].

Since $V = R/s_b$, and R and s_b could change with time, it is necessary to periodically measure these quantities. We measure s_b using observation 1 above. R is the time between sending out a packet and receiving an ack when all the queues in the VC are empty. This can be approximated by measuring r_t, the round trip time, though r_t will have a component due to the queueing delay.

> *If the pipeline depth* V *can increase or decrease by at most* ΔV *in any interval of time* r_t*, then keeping* ΔV *packets in the bottleneck queue's buffers will ensure that usually the bottleneck will not be idle.*

If an increase in R or ρ_b increases the pipeline depth by ΔV, bottleneck bandwidth will be lost until the source reacts to the change. Since a source will take at least R time units to react to the change, we want to have enough slack in the buffer to take up any transients. If the bottleneck queues ΔV packets, when V increases, the buffer will be drained, and no loss of throughput occurs. Thus, 2P tries to ensure that at any given time, at least ΔV packets are present in the bottleneck queue. We assume a buffer capacity of at least $2\Delta V$ per conversation at every switching node.

Note that this scheme avoids wasted bandwidth but adds a delay (of $\Delta V.s_b$) to every packet served. A user can adjust the bottleneck queue size to obtain a range of delay *vs.* lost throughput tradeoffs. We denote the target bottleneck queue size by n_b, and in the rest of the paper, n_b is assumed to be ΔV.

4.2. Algorithm

There are three phases in the operation of 2P: start-up, queue priming and normal transmission.

At start-up, we do not know the value of s_b. Since we do not want to overload the bottleneck with packets, some sort of 'slow-start' is desirable. We combine this with an initial measurement of the VC parameters by sending a *packet-pair*, two packets sent as fast as possible (back-to-back). The round-trip time of the first packet gives us R_e, the estimator for R and the inter-arrival time of the two packets gives us s_e, the estimator for s_b.

Once the source computes V_e, an estimator of $V = Re/se$, it can decide what n_b should be. Deciding n_b *a priori* is possible, but not desirable, since we might want n_b to be some

---

A plot of throughput versus network load reveals that throughput increases linearly with load, until queueing occurs. The point at which queueing is incipient is called the knee [5]. At the knee, there is no queueing at the bottleneck, and the maximum possible throughput is achieved.

fraction of V_e. During queue priming, the source sends out a burst of n_b back-to-back packets so that the n_b packets accumulate in the bottleneck queue.

During normal transmission the source transmits packet-pairs every $2s_e$ time units and updates s_e to the inter-arrival time between paired acks. R_e, the estimate for R, is updated to $r_t - n_b . s_e$. To react immediately to changes in V_e, we recompute R_e / s_e on the arrival of every pair. Let V_{new}, V_{old} be the old and new values of V_e using the old and new estimates respectively. If $V_{new} < V_{old}$, we calculate the quantity

$$n_{skip} = \max(\lceil (V_{old} - V_{new})/2 \rceil ,\ 0)$$

where $\lceil z \rceil$ computes the smallest integer greater than or equal to z. The source then skips n_{skip} transmission slots with a duration of the new value of s_e, and then continues to send pairs of packets at regular intervals of $2s_e$. If $V_{new} > V_{old}$, the source immediately transmits a burst of $V_{new} - V_{old}$ back-to-back packets.

In practice, intermittent bursts of more than two packets may be useful to account for statistical variations in s_b. We will investigate this in future work.

4.3. Implementation

Figure 1 is the state diagram for a 2P implementation.
A 2P source usually is in the ‘receive’ state, waiting for an interrupt, one of

a) A signal indicating receipt of an acknowledgment packet. (ACK)
b) A ‘tick’ indicating that at the current sending rate, the next packet is due to be sent. (TICK)
c) A signal indicating that the output line is now free. (INT)
d) A signal indicating that the last packet sent out has been timed out. (TIMEOUT)

There are two important state variables. `linefree` indicates that the output line from the source is free. `num_in_burst` is the number of packets enqueued in the output queue that belong to a burst. As long as `num_in_burst` is positive, a packet will be dequeued and sent on the arrival of every INT.

When an ACK arrives, if the ack is the first of a pair, R_e is updated. If it is the second of a pair, s_e is updated, and the source computes V_e and n_{skip}. If $V_{new} > V_{old}$, a burst of $V_{new} - V_{old}$ packets are queued on the output queue. When an acknowledgment is received, some of the packets it acknowledges may have been timed out, and may be enqueued waiting to be sent out. So, at this point, all queued retransmissions that have become invalid are discarded. If the acks received are for the first pair, 2 packets are enqueued on the output queue and the TICK timer is set to $2s_e$.

If a TICK is received and n_{skip} is non-zero, it is decremented, the TICK timer is reloaded with $2s_e$, and we return to the receive state. Else, two packets (from the client) are enqueued on the output queue. If the line is free, one of the packets is dequeued and sent, else the source waits for an INT to arrive.

When an INT arrives, if there are burst packets to be sent, one of them is dequeued and transmitted, else the source marks the line as free and returns to the receive state. We assume one retransmission timer per packet, and selective retransmission. Each time a packet is sent, its retransmission timer is set to $X(R_e + n_b . s_e)$, where X is some small integer, and can be used as a tuning parameter (we used $X = 3$). On a TIMEOUT the timed out packet is placed in the output queue, waiting to be retransmitted. The new timeout value for

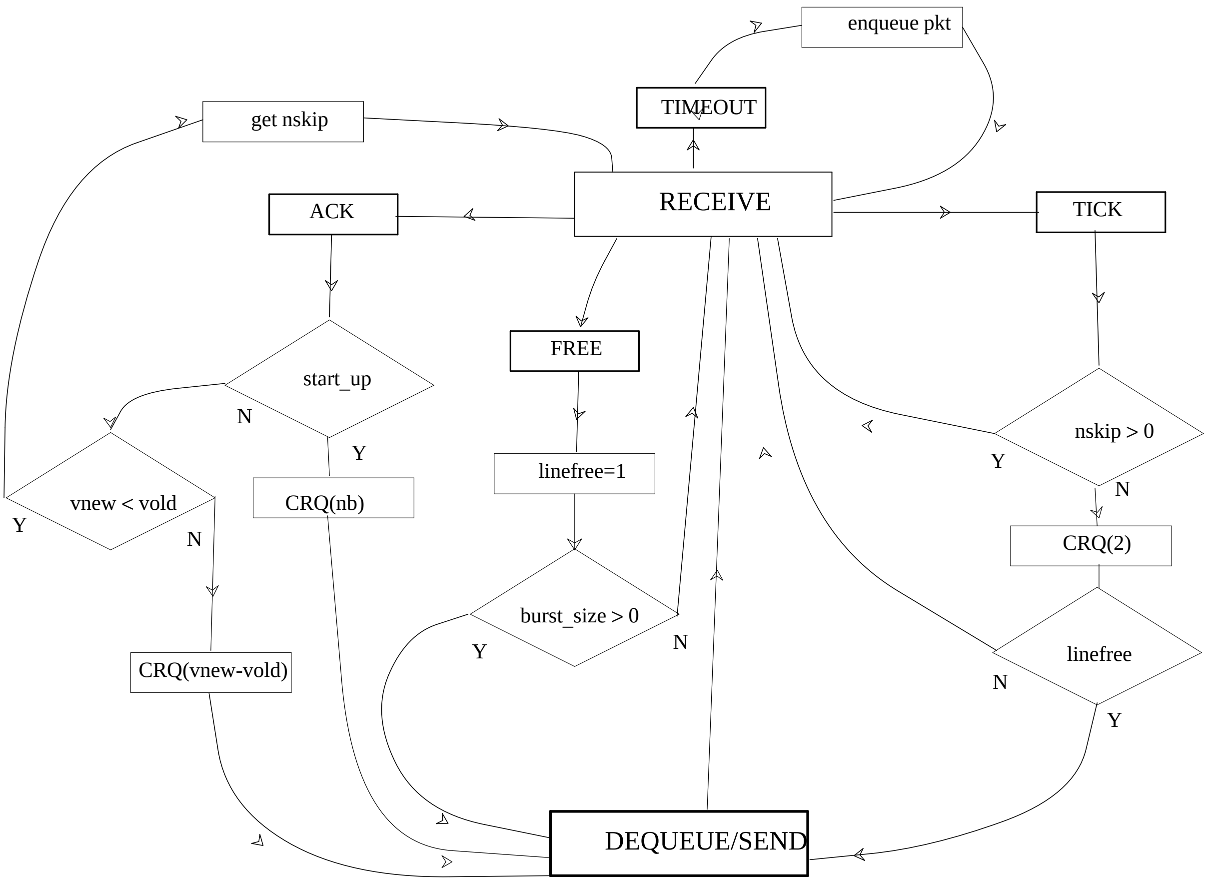


Figure 1: State Diagram for 2P Implementation. CRQ(x) means accept x packet from the user and enqueue them on the output queue for transmission.

the packet is twice the old value.

5. ANALYSIS OF 2P

We will analyze the the behavior of 2P in the steady state (that is, when R and s_b do not change), and its response to transient changes in the virtual circuit. We will make four simplifying assumptions:

- Flow control is being done on behalf of an infinite source, that always has some data ready to send.
- Changes in V are bounded from above by ΔV, and that the source knows or can estimate this value.
- Each server reserves $B \geq 2\Delta V$ buffers for each source.
- Transients are assumed to be due to a sharp, rather than a gradual, change in the system state. We assume that the value of a parameter, such as R, is constant until time t_0, at which point it changes discontinuously to its new value. We denote the value of $R(t)$ before the change as $R(t_0 - \varepsilon)$, and after as $R(t_0 + \varepsilon)$. We define $s_b(t_0 - \varepsilon)$ and $s_b(t_0 + \varepsilon)$ similarly.

Optimal flow control

We introduce the notion of optimal flow control and show that in the steady state, 2P is optimal. An optimal transmission flow control scheme should *always* operate at the knee of the load-throughput curve, so that maximum throughput is achieved with minimum delay [5]. As the conditions at the server change, the source flow control must adapt itself to the change. However, if we consider the speed-of-light propagation delay in this control loop for wide-area networks, it is clear that no realizable flow control scheme can always operate at the knee. Hence, we propose a weaker definition of optimality that is suitable for high throughput applications.

Let the bottleneck have B buffer spaces available for a source. Then, a flow control scheme is optimal in the interval $[T_0, T_1]$ if in every time interval $[t_1, t_2] \in [T_0, T_1]$, there are no buffer overflows, and there is no loss of bandwidth at the bottleneck node. To be precise, at the bottleneck node, if the buffer occupancy at time t_1 is k,

$$0 \leq \int_{t_1}^{t_2} (\rho_0(t - d_1) - \rho_b(t))\, dt \leq B - k$$

where $\rho_0(t)$ is the source sending rate at time t, $\rho_b(t)$ is the bottleneck service rate at time t and d_1 is the propagation delay from the source to the bottleneck.

Steady state behavior of 2P

In the steady state, 2P will keep n_b packets in the bottleneck queue, and send packets at exactly the service rate, ρ_b. Proposition 1 proves optimality of 2P in the steady state.

Proposition 1: Let the transmission at the source start at time T_0 and end at time T_1. If V is constant in $(T_0, T_1]$ then 2P is an optimal flow control scheme in $[T_0 + 2R(0), T_1]$.

Proof:At time $T_0 + R(0)$, the source knows ρ_b. Since $\Delta V = 0$, $n_b = 0$, and priming the queue is not necessary. Thus, the source will immediately start to send a packet-pair every $2s_b$ time units. The first pair reaches the bottleneck latest by $T_0 + 2R(0)$. Since service is at the rate of ρ_b, there is no build up of the queue. Clearly, no bandwidth is lost, and optimality conditions are trivially satisfied in $[T_0 + 2R(0), T_1]$.

Remark

Note that many schemes described in the literature do not satisfy this weak notion of optimality even in the steady state. The Jacobson-Karels TCP modifications [4] drop packets if the maximum possible window size is larger than the buffer capacity at the bottleneck queue. The DECbit scheme keeps the queues at an average queue length of 1, and so will lose bandwidth when V increases [5]. As we showed earlier, NETBLT causes queues to build up whenever V decreases, and they are never adjusted for. Hence, a sequence of decreases in V will cause NETBLT to drop packets. Jain's delay based scheme [6] will respond poorly if V decreases due to a decrease in R, since it interprets the decrease in delay as a signal to increase the window size, which will cause further queueing, and possible packet loss. A more detailed analysis of these schemes can be found in [9].

Response to transients

In our model, the only network parameters visible to a source are ρ_b and r_t. Thus, a flow control scheme can react to a change only in either of these variables, and we will study the response of 2P to these changes. For each change, we study the packet or bandwidth loss, and the time taken to return to steady state.

Note that r_t itself depends on R and on the queueing delay in the bottleneck node. In the steady state, the queueing delay is constant, and r_t changes only if R or ρ_b change. Thus, we need only consider changes in R and ρ_b. In either case, the effect is to change $V = R.\rho_b$. We will denote the time at which the change occurred by t_0, and the change in V by $\delta V \leq \Delta V$.

5.1. Increase in R

5.1.1. Loss of bandwidth

R could increase if the VC is rerouted, and some new servers are added to the VC's path. There are two cases: either the path increase occurs before the bottleneck node, or after. If the increase happens after the bottleneck, then some server downstream of the bottleneck will have an increased idle time, and there is no loss in bottleneck bandwidth.

If the increase happens before the bottleneck, then for some period of time, the bottleneck could be idle, since packets that should have arrived at the bottleneck will now be sent to the new servers instead. Since we bound the increase in V by ΔV, if the bottleneck queue has ΔV packets, the buffered packets will be transmitted in the interim and there is no loss of bandwidth.

Another subtle possibility for bandwidth loss is if the first packet of a packet-pair reaches the bottleneck after time delay D_1 while the second one reaches there after a delay D_{1new}. The inter-arrival time of the pair at the bottleneck is then $\xi = D_{1new} - D_1$. If this is larger than s_b, the protocol will react by skipping some pairs of packets. However, 2P will recover as soon as the next pair of acks arrive.

Proposition 2 The loss of bandwidth will be $2n + \xi / s_b$ packets, where

$$n = \max\left(\left\lceil \frac{1}{2} \left(\frac{R_{old}}{s_b} - \frac{R_{new}}{\xi} \right) \right\rceil, 0 \right)$$

Proof: The second term, ξ / s_b, is just the bubble size in the pipeline. Since 2P mistakenly estimates that the pipeline depth is R_{new} / ξ, instead of R_{new} / s_b, it skips n slots, and in each slot it loses 2 packets, leading to a net loss of $2n$ packets.

5.1.2. Recovery time

Suppose R increased at time t_0. This information reaches the source latest by time $t_0 + R(t_0 + \varepsilon)$. The source will immediately send a burst of δV packets. Since the source is sending at ρ_b, all but the bottleneck server will be idle when this burst is initiated. By the Burst dynamics lemma, we see that the burst will fully arrive at the bottleneck at time $t_0 + \sum_{i=0}^{b-1} s_i + \frac{\delta V}{\rho_{sl_{b-1}}}$. This replenishes the bottleneck queue and so the steady state is regained. Subsequent increases in R are handled as before.

5.2. Increase in ρ_b

5.2.1. Loss of bandwidth

ρ_b could increase if the number of conversations at the bottleneck decreases. The increase in ρ_b could lead to some other node in the VC's path to become the bottleneck. We need to consider two cases, depending on whether the bottleneck migrates or not (that is, whether or not some other node becomes the bottleneck).

Assume that the bottleneck does not migrate. Due to the increase in ρ_b, the bottleneck will serve packets faster than they arrive, and until the first packet transmitted at the new rate arrives, there could be a loss of bandwidth. Now, the increase in ρ_b becomes known to the source by time $t_0 + \sum_{i=b}^{n} s_i$, and the first packet from the burst reaches the server by $t_0 + R(t_0)$. So, if there are $(s_b(t_0+\varepsilon)-s_b(t_0-\varepsilon))R(t_0) \leq \Delta V$ packets in the bottleneck buffer, there will be no loss of bandwidth.

If the bottleneck migrates downstream from the old bottleneck, then packets queued at the old bottleneck will arrive at the new bottleneck and will form a queue there. It is easy to show that the only loss of throughput happens for the brief interval where the first packet from the old bottleneck is in transit to the new bottleneck.

If the bottleneck migrates upstream, then the δV packets queued at the old bottleneck cannot compensate for the bubble in the pipeline. The new bottleneck will be idle till the first packet from the compensatory burst arrive, and the loss could be as large as δV packets. Note that no flow control algorithm can prevent this loss, since the source must take at least $R(t_0)$ time to react to the change in s_b. Thus we have shown that no feasible flow control algorithm can satisfy even our weak notion of optimality in non-steady state operation.

5.2.2. Recovery time

The recovery time is just the time for δV packets to accumulate at the bottleneck, and this is the same as in the case of the increase in R, that is $\sum_{i=0}^{b-1} s_i + \frac{\delta V}{\rho_{L_{b-1}}}$.

5.3. Decrease in R

5.3.1. Loss of packets

R could decrease if packets are sent through a shorter route, but through the same bottleneck. We will assume that the packets in transit are not lost, but are received at the bottleneck. The effect of this change is to put an additional δV packets in the buffers of the bottleneck queue. Since in the steady state there are ΔV packets in the bottleneck, and we assume that we can buffer $2\Delta V$ packets, there is no packet loss.

5.3.2. Recovery time

The source knows the reduced value of R latest by time $t_0 + R(t_0 + \varepsilon)$. At this point, it will skip $(V_{old} - V_{new})/2$ transmission slots, taking a time of $1/2(R(t_0 - \varepsilon)/s_b - R(t_0 + \varepsilon)/sb)2sb, = R(t_0 - \varepsilon) - R(t_0 + \varepsilon)$. The first packet after resumption of normal transmission reaches the bottleneck latest by time $t_0 + R(t_0 - \varepsilon) - R(t_0 + \varepsilon) + R(t_0 - \varepsilon) = t_0 + R(t0 - \varepsilon)$. Since the excess packets accumulated at the bottleneck is exactly $(R(t_0 - \varepsilon)/s_b - R(t_0 + \varepsilon)/s_b)$, they will all be cleared in this time, and the system reaches the steady state at time $t_0 + R(t_0)$

5.4. Decrease in ρ_b

5.4.1. Loss of packets

The source knows of the decrease in ρ_b latest by $t_0 + R(t_0)$, and will skip $\delta V/2$ transmission slots. The bottleneck could accumulate an additional δV packets in this time. Since there are $2\Delta V$ buffers, there is no packet loss.

5.4.2. Recovery time

The source detects the decrease in ρ_b by time $t_0 + \sum_{i=b}^{n} s_i$, and will skip some transmissions. The first packet sent at the new rate is received at the bottleneck after a time $\sum_{i=1}^{b-1} s_i$, so that the first packet arrives at the bottleneck at time $t_0 + R(t_0) + skiptime$. During the first $R(t_0)$ time units, the bottleneck queue will accumulate packets in excess of ΔV, but exactly these many packets will be serviced during *skiptime*. Hence, at $t_0 + R(t_0) + skiptime$ the steady state is attained.

The source skips transmission for the time during which the bottleneck services the excess packets accumulated in its queue. Hence,

$$skiptime = \max\left(\left\lceil \frac{1}{2} \frac{R(t_0)}{s_b(t_0 - \varepsilon)} - \frac{R(t_0)}{s_b(t_0 + \varepsilon)} \right\rceil, 0\right).\, sb(t_0 + \varepsilon)$$

6. SIMULATION

Our goal in doing a simulation study was to examine the dynamic behavior of 2P in a simple deterministic framework. We compare the behavior of 2P with the NETBLT-like rate control scheme described earlier, the Jacobson-Karels modifications to TCP [4] and the DEC binary congestion avoidance scheme [5]. The simulations were done using the REAL

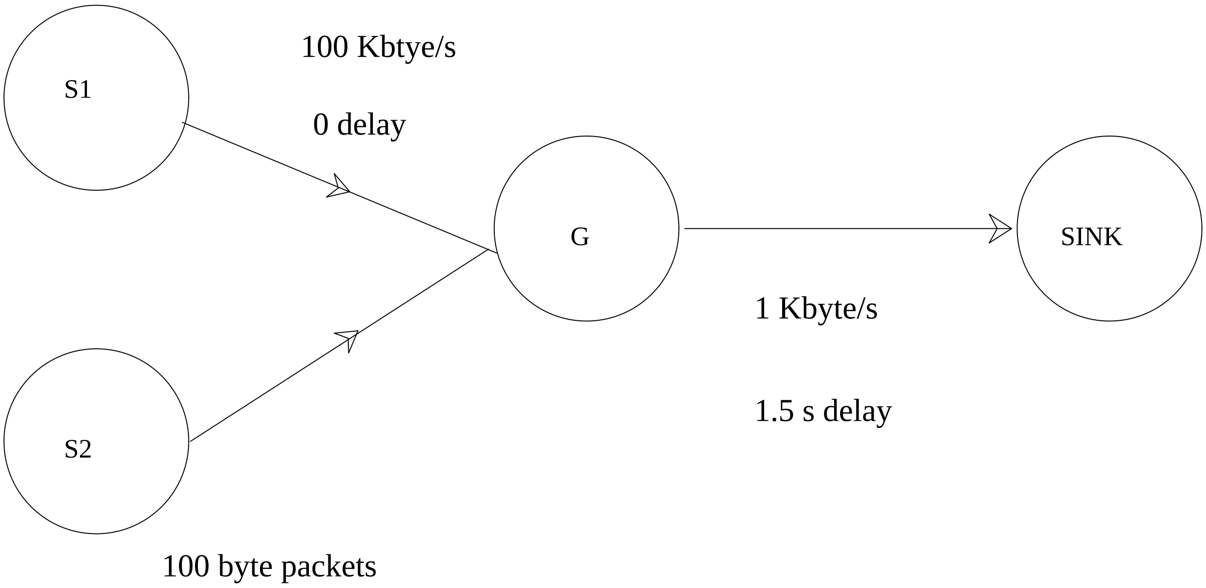


Figure 2: Simulation scenario

simulation package [7].

The simulation scenario is shown in Figure 2. G is a gateway or router, S1 implements 2P, S2 is a background traffic source that uses half the resources at the gateway for 300 seconds, is idle for the next 300 seconds, and restarts 300 seconds later. The sources and the gateway are assumed to be much faster than the line service rate, so the bottleneck is the capacity of the line from the gateway to the sink. The line capacity of the line from the sources to the gateway is 100 Kbytes/sec, and that from G to the sink is 1 Kbyte/sec. The buffer at the gateway has a capacity of 4 packets. The simulations were run for 800 seconds.

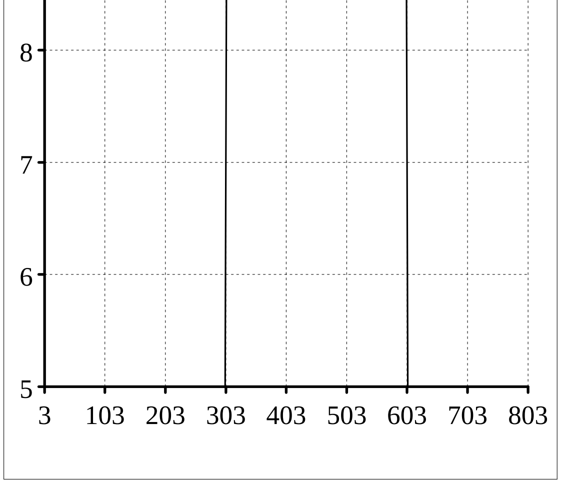
Observations

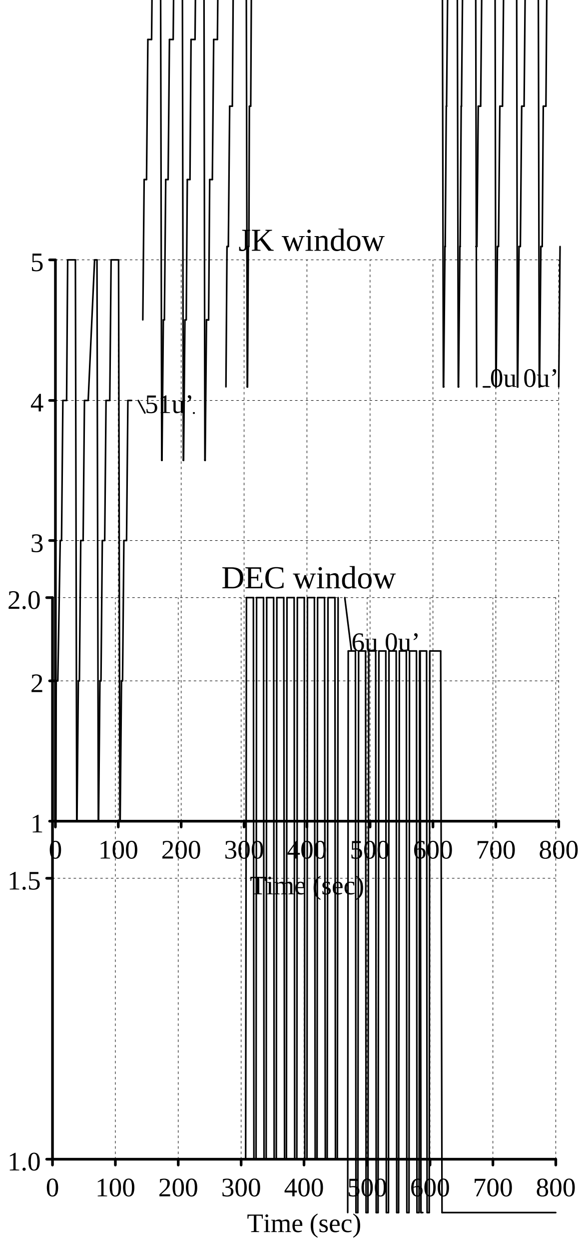
The ideal behavior for a source is to send at half the bottleneck service rate for the first 300 seconds, send at full speed for the next 300 seconds, and return to half speed after that. We see that 2P closely approaches this, with a time lag of 3 seconds (which is the control delay in the network).

NETBLT shows three main features. First, it reaches the correct value (half speed) only after about 180 seconds. The explanation is interesting: NETBLT sends at a rate ρ_0 by setting a 'tick' timer to go off at $1/\rho_0$. At start up, the sending rate is unknown, so NETBLT sends at a small rate ('slow-start'), and the initial tick timer values are large. This means that the adjustment process is slow, since the a large tick prevents the bottleneck from being probed more frequently, and so prevents itself from being reduced. Once the system reaches steady state, however, the tick value is small enough that subsequent adjustments are made rapidly.

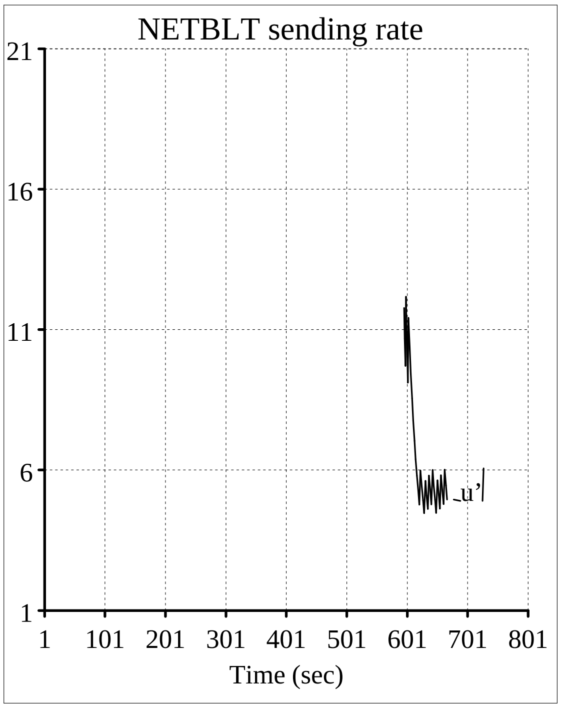
Second, NETBLT has a tendency to oscillate around the correct sending rate. This is because the multiplicative factors tend to overcompensate the sending rate, with the obvious dynamics. Finally, because of the oscillations, the mean queueing delay for NETBLT is higher than that of 2P.

The JK protocol had a maximum window size of 5. When the background source is active, the JK source obtains, on average, only 2 buffers, and so when it tries to push its




JK window

DEC window


NETBLT sending rate