VirtualClock: A New Traffic Control Algorithm for Packet-Switched Networks

LIXIA ZHANG
Xerox Palo Alto Research Center

One of the challenging research issues in building high-speed packet-switched networks is how to control the transmission rate of statistical data flows. This paper describes a new traffic control algorithm, *VirtualClock*, for high-speed network applications. VirtualClock monitors the average transmission rate of statistical data flows and provides every flow with guaranteed throughput and low queueing delay. It provides firewall protection among individual flows, as in a TDM system, while retaining the statistical multiplexing advantages of packet switching. Simulation results show that the VirtualClock algorithm meets all its design goals.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*packet networks*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol architecture*; C.4 [**Computer Systems Organization**]: Performance of Systems:—*performance attributes*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Data traffic control, performance guarantee, rate-based flow-control algorithms, statistical multiplexing, time-division-multiplexing

1. INTRODUCTION

High-speed packet-switched networks introduce challenges in data traffic control. One is that, due to the large product of bandwidth and signal propagation delay over a path, a large amount of data can be stored in the "pipe" at any given time, which makes network congestion control difficult. Another is the stringent performance requirements raised by new applications, such as digitized voice and video; contrasted with conventional reliable data transfer applications, these new applications often require guaranteed throughput and bounded transmission delay, possibly with a relaxed requirement for error recovery [7].

Window-based flow control mechanisms have been widely used for traffic control in packet-switched networks [9, 22] and have served well for reliable data transfer applications in low-speed network environments. The service requirements introduced by new applications, however, make it difficult, if not impossible, for window-based control mechanisms to support them

Author's address: Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

effectively. For instance, a window-based flow control system uses returned acknowledgments both to initiate loss recovery and to regulate further data transmissions, while a digital video application may be willing to tolerate minimal packet losses but not retransmission delays. Moreover, although some video encoders generate data at variable rates, these data flows cannot be controlled by individual acknowledgments, since acknowledgments may incur a rather large and variable delay that are not compatible with the frequencies and regularity of packet generation in most video or audio communication [6].

For these reasons, alternative methods such as rate-based data traffic control algorithms have become a focus of research in recent years [1, 2, 13]. However, there exists a number of difficulties in designing rate-based traffic control algorithms. Among them are how to monitor and control the transmission rate of statistical data flows and how to enforce network resource usage to prevent interference among different users without sacrificing the statistical multiplexing feature of packet switching.

In this paper we introduce *VirtualClock* as a new traffic control algorithm for high-speed network applications. VirtualClock controls the *average* transmission rate of statistical data flows, enforces each user's average resource usage according to the specified throughput, provides firewall protection among individual flows, and supports multilevel priority services. The algorithm has been tested extensively through simulation.

VirtualClock was designed as part of a new network architecture, the Flow Network [25]. In this paper we first give a brief description of the Flow Network architecture to provide the reader the background for the VirtualClock design. In Section 3 we describe the VirtualClock algorithm and discuss its design and fundamental properties. Then in Section 4 we present simulation results to demonstrate the effectiveness and performance of the VirtualClock algorithm. Section 5 compares the VirtualClock algorithm with a few others that have been proposed for network traffic control, mainly a fair-queueing [3, 8], a schedule-based approach [14], and the Leaky-Bucket algorithm [17, 20]. We conclude the paper with a discussion on future research issues.

2. BACKGROUND: FLOW NETWORK

The goal of the Flow Network is to provide users with guaranteed service qualities. The design of the Flow Network architecture focuses on three major issues: where to put the control mechanisms, what kind of control mechanisms to use, and whether control should be based on resource reservation or feedback. In contrast to end-point control with a stateless network model, this design lets the network play an active role in traffic control, because a stateless network cannot *ensure* service quality. Instead of a window control mechanism, this design controls packet flows by controlling average transmission rate, because network resources are measured in rate [24]. And instead of relying on feedback control to adjust users' transmission speed, this design proposes a reservation-based control system for applications that require guaranteed performance, and uses feedback control only to adjust the disparity between the users' reservations and the actual packet transmissions obtained from network measurement results.

In a Flow Network, an abstract entity, the *flow*, is defined to represent users' data transmission requests. Below we give a brief description of the flow.

2.1 What Is a Flow?

In the Flow Network model, a flow is a stream of packets that traverse the same route from the source to the destination and that require the same grade of transmission service. The diversity in various applications' service requirements suggests that, together with a data transmission task, the user must also submit a service specification to the network. Therefore each flow is associated with a specific set of flow parameters (although the values of the parameters can be dynamically adjusted during the session). This specification enables the network to check whether adequate resources are available before accepting new transmission requests. It also enables the network to provide meaningful feedback information in case the actual data flow goes beyond the expected region, to help the flow source adjust its transmission or parameter. Furthermore, this specification serves as a contract between the network and the user: it is used as criteria that the network service must meet, as well as a constraint that the user's transmission behavior must adhere to.

A flow differs from packets in datagram networks in that the flow is treated as one stream of packets, rather than as an aggregate of independent entities, so that the network can allocate resources to a flow and can monitor the flow's behavior. A flow differs from a virtual connection in virtual-circuit networks in that it is not concerned with data integrity. Instead, a flow is characterized by its service requirements; it is associated with the allocation and deallocation of network resources that are required to deliver the data of that flow within the specified performance bounds.

Although describing input traffic entails much work on the part of a user, no service guarantee can be provided if the user does not specify the characteristics of the expected data traffic. Facing random packet traffic and unforeseen future applications, we see nothing superior to a description from users as a proper estimate on their own transmission behavior. Next we discuss what parameters are used to describe the transmission of a statistical data flow.

2.2 Choosing Flow Parameters

We need a set of parameters that can adequately describe the required behavior of a statistical data flow and can be conveniently used in flow measurement and control. Packet traffic is characterized as bursty and random; how should one describe the "burstiness" and "randomness" attributes of a packet flow? How should one express the flow's throughput rate?

Considering each flow as a statistical process, one may describe the throughput by using the average and the variance of the number of packet arrivals over a unit time period. Knowing the average rate helps the network control the resource utilization. Measuring the average, however, is not easy; a major difficulty lies in how to choose a proper average time period. One cannot easily derive a proper average period from the variance of arrivals unless a random process characterization of the data source is also given, which does not seem to be a feasible requirement. Generally speaking, not all applications' data generation patterns can fit into well-known random process models, although we might be able to find a feasible model for a particular application.

Taking a simple and pragmatic approach, we choose two parameters to describe a statistical data flow: *average rate* (AR) and *average interval* (AI). That is, over each AI time period, dividing the total amount of data transmitted by AI should result in AR.

How to choose the AI value for a flow is an important question. The possible range of the AI value is

$$1/\mathrm{AR} \le \mathrm{AI} \le \text{total flow duration}.$$

If the AI value were set at its lower bound, the flow's client would be transmitting at a constant rate as in a circuit-switching network. If it were set to the total duration, the client would be entitled to transmit data in any arbitrary manner, as in an uncontrolled datagram network. Thus, the choice of the AI value must be made in a way that strikes a balance between a tight controllability and the maximal possible tolerance for burstiness in packet flows.[1] Restated, a well-engineered AI value should be small enough to give the network effective control, but large enough to accommodate variations in packet arrivals within each average interval, so that the average rate measured over each AI period will remain relatively constant.

Tolerating flow burstiness also implies the need for adequate buffer space at each switch node, so that packet losses can be maintained at a negligible level. The Flow Network design assumes an adequate buffer space available at each switch to tolerate reasonable flow burstiness. To avoid extraordinary demand on the buffer space, the network also sets an upper limit on the AI value each flow may choose, based on the link bandwidth and the control feedback delay [25].

In the Flow Network, data transmissions follow three logical phases: flow set-up, data transmission, and flow tear-down. To start a flow, the flow source first sends a set-up request that contains, among other information, the average rate (AR) and average interval (AI) that describe the flow's data transmission behavior. The route of a flow is chosen at set-up time.[2] Resources are allocated along the chosen route during the flow set-up. When the

[1] In this paper we use the word burstiness to mean both that packet transmissions are not evenly spaced and that the timing and length of each burst are random

[2] Although there exist intrinsic interactions between network routing and traffic control, they are beyond the scope of this paper. We approach the traffic control problem under the assumption that a network-routing service exists that can provide a proper route upon each data transmission request.

transmission finishes, either end of the flow can send a tear-down message, which deallocates the resources.

2.3 Datagram Traffic

One might argue that there will always exist transaction-oriented applications that exchange only one or a few packets at a time, such as requests to network name servers or time servers. Such datagram traffic does not match the *flow* model well—it is infeasible to require flow setup or resource allocation activities for just a few packets. The Flow Network design provides an escape from the reservation requirement for datagram traffic: datagram traffic is assumed to desire a best-effort service, and therefore no resource reservation is necessary. The design considers the aggregation of all datagram traffic as a whole, estimates the total volume, and preserves resources accordingly. This strategy will work well so long as datagram traffic composes only a small portion of the total load. It does, however, open a possibility of unexpected load fluctuation. The network, therefore, must be able to ensure committed performance for established flows even in the presence of heavy datagram traffic.

2.4 The Role and Functionality of VirtualClock in the Flow Network

After flow set-up, the VirtualClock algorithm is designed both to ensure resource usage and to monitor flows. More specifically, it provides the following functionalities:

(1) Support for the diverse performance requirements of various applications by enforcing the resource usage according to each individual flow's throughput reservation, while preserving the flexibility of statistical multiplexing of packet-switching networks.

(2) Monitor the average throughput rate of data flows and provide measurement input to other network control functions, as well as provide feedback to flow sources whenever the actual data flows violate the agreement negotiated during the flow set-up.

(3) Provide firewall protection among individual data flows, particularly firewalls between datagram traffic and flows that require performance guarantees. As has frequently been observed in operational networks, network users may sometimes misbehave. For example, a user may transmit data at a high rate without listening to the network control information. Even though in this paper we assume that no user has malicious intention, such misbehavior can still be caused by software or hardware failures, by protocol implementation errors, or even by unforeseen protocol design errors [15, 19]. It is the responsibility of the network control to prevent misbehaving users from interrupting normal service to others.

The VirtualClock algorithm plays a key role in the control of traffic in the Flow Network. In the rest of the paper we will focus mainly on the VirtualClock algorithm, its basic concepts, and properties. Although there are a number of other interesting and important issues in the Flow Network design, such as how to request resource reservations during flow setup or how to dynamically adjust a flow's throughput, they are beyond the scope of this paper and will be explored elsewhere.

3. VIRTUALCLOCK ALGORITHM

The basic idea of VirtualClock was inspired by Time Division Multiplexing (TDM) systems. A TDM system guarantees each user the prescribed transmission rate. It also completely eliminates interference among users, as if there were firewalls in between that protect individually reserved bandwidths. Users, however, are limited to transmission at a constant bit rate; the system tolerates no variations in the speed of data generation. In addition, capacities are wasted when a slot is reserved for a user that has no data to send at that moment. Furthermore, both the channel bandwidth allocated to each user and the total number of users that each network link can accommodate at any time are fixed rather than dynamically adjustable.

Our goal is to achieve both the guaranteed throughput and the firewall protection of a TDM system, while at the same time preserving the statistical multiplexing advantages of packet switching. Therefore, the network should assign "slots" to flows on a demand basis; only when conflicts in demand occur should the network regulate the resource usage to guarantee each flow's reserved throughput. A TDM system regulates the resource usage by using a real-time clock—as the clock ticks, each user channel sends data in turn. A statistically multiplexed system may use a *virtual clock* in a similar way.

To make a statistical data flow resemble a TDM channel, we imagine that arriving packets from the flow are spaced out by a constant interval in virtual time, so that each packet arrival indicates that one slot time period has passed. We can assign to each data flow a *VirtualClock*, which ticks at every packet arrival from that flow. If we set the tick step to the mean interpacket gap (assuming a constant packet size for the moment), the value of the VirtualClock will denote the expected arrival time of the arrived packet. To imitate the transmission ordering of a TDM system, we can let each switch node stamp packets of each flow with the flow's VirtualClock time and order packet transmissions according to the stamp values, as if the VirtualClock stamp were the real-time slot number in a TDM system (see Figure 1). If a flow transmits according to its specified average rate, its VirtualClock reading should fluctuate about real time.

In the rest of this section, we first give an exact description of the VirtualClock algorithm and then explain its major functions and the choices that motivated the design.

3.1 The Algorithm

The actual algorithm is slightly different from the basic idea described above. Each switch along the path of a flow uses two control variables, a VirtualClock and an auxiliary VirtualClock (auxVC), to monitor and control the flow according to the specified AR and AI values. More specifically, each

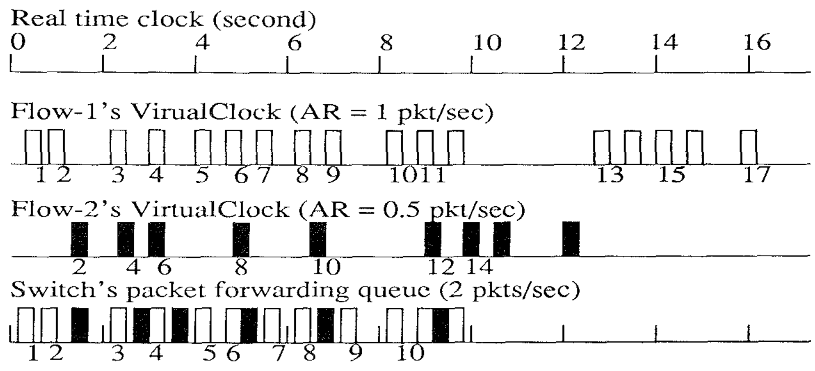
Real time clock (second)
0 2 4 6 8 10 12 14 16
Flow-1's VirualClock (AR = 1 pkt/sec)
1 2 3 4 5 6 7 8 9 10 11 13 15 17
Flow-2's VirtualClock (AR = 0.5 pkt/sec)
2 4 6 8 10 12 14
Switch's packet forwarding queue (2 pkts/sec)
1 2 3 4 5 6 7 8 9 10

Fig. 1. Real time, VirtualClock, and packet-processing order.

packet switch performs the following two basic functions:

Data forwarding: The switch has one packet-waiting queue in front of each outgoing link; it serves packets in the following way:

(1) Upon receiving the first data packet from $flow_i$, $VirtualClock_i \leftarrow auxVC_i \leftarrow$ *real time*.

(2) Upon receiving each packet from $flow_i$,

(a) $auxVC_i \leftarrow \max(\text{real time}, auxVC_i)$;

(b) $VirtualClock_i \leftarrow (VirtualClock_i + Vtick_i)$, and $auxVC_i \leftarrow (auxVC_i + Vtick_i)$;

If all packets have a constant size, $Vtick_i = 1/AR_i(packet/sec)$. If packets vary in size, the value of $Vtick_i$ should be computed from individual packet sizes.

(c) Stamp the packet with the $auxVC_i$ value.

(3) Insert the packet into its outgoing queue. Packets are queued and served in the order of increasing stamp values.

Flow monitoring: The switch computes a control variable, $AIR_i = AR_i \times AI_i$ at $flow_i$ set-up. Upon receiving every set of AIR_i data packets from $flow_i$, the switch checks the flow in the following way:

— If $(VirtualClock_i - \text{real time}) > \text{T}$, where T is a control threshold, a warning message should be sent to the flow source. Depending how the flow source reacts, further control actions may be necessary. (See Zhang [25] for more details.)
The value of the control threshold T in the above will be discussed in Section 3.3.

— If $(VirtualClock_i < \text{real time})$, $VirtualClock_i \leftarrow$ real time.

At this time, the switch also synchronizes VirtualClock and auxVC if doing so does not cause packets from the same flow from being served out of order: $auxVC_i \leftarrow VirtualClock_i$ if either the outgoing link for $flow_i$ is idle or the packet being transmitted has a stamp value greater than $auxVC_i$.

Queueing packets in the order of their stamp values makes packets from different flows maximally interleaved as in a round-robin service system. We use an example to explain this effect: if a burst of packets from $flow_i$ arrives at a switch shortly after a burst from $flow_j$ and if the two flows have the same average rate, then the packets of the two flows will be one-to-one interleaved in the waiting queue; if $flow_i$ is twice as fast as $flow_j$, then the packets will be two-to-one interleaved in the queue.

The buffer pool at the switch is completely shared. When the pool is exhausted, the VirtualClock algorithm drops the last packet—the one with the largest stamp value—from the (longest) queue (if the switch has more than one packet queue).

Although the basic concept of VirtualClock is simple, not all of the individual steps in the above description are obvious. In particular, the reader may wonder why it is necessary to introduce the auxiliary variable, auxVC, instead of simply stamping packets with the VirtualClock value; why the switch checks each flow after receiving every set of AIR packets instead of after every AI period; or how a bursty data source should regulate its packet transmissions so that it will not be throttled by network monitoring. In the remainder of this section we first discuss some distinct features and functionalities of VirtualClock; the first two questions will be answered along the way. We then explain in detail the last issue raised above.

Before proceeding, a brief explanation about the relevant values of VirtualClock and auxVC is in order. VirtualClock and auxVC will contain the same value most of the time (when packets from a flow arrive at the expected time or earlier); auxVC may have a larger value temporarily (when a burst of packets arrives very late in an average interval) until being synchronized with VirtualClock again. Therefore, in the following discussions we can reasonably assume that packets are stamped with VirtualClock values in order to make the algorithm easily understood. The need for auxVC will be explained in Section 3.2.3.

3.2 Basic Functionalities

3.2.1 *Providing Firewall Protection Among Flows.* Serving packets in the order of VirtualClock values assures that each flow will receive the resources that it has reserved. Therefore, in a network controlled by VirtualClocks, although an aggressive flow can consume idle resources, it cannot disturb network service to other flows. Through resource reservation, the network assures that no congestion will occur if every flow transmits according to its reserved throughput. In cases where one or more flows violates its reservation, flows that remain within their specified throughput rate will not be affected, while the most offending flows will receive the worst service (because their VirtualClocks advance too far beyond real time, their packets will be placed at the end of the service queues or even be discarded). The VirtualClock algorithm thus prevents interference among flows.[3]

[3] However, if there exist a number of ill-behaved flows, they might interfere with each other See analysis by Weinrib [23] for more details.

3.2.2 *Providing Priority Service.* Priority service can be easily acommodated by the VirtualClock algorithm. A network can provide priority services to a flow simply by letting each switch replace "real time" by "real time − P" in the previous VirtualClock algorithm description, where P is a chosen value representing the priority. In general, P's value should be chosen large enough to separate priority flows from nonpriority flows in the service queue.[4] Use of a priority value, however, will not allow priority flows to take unfair advantage of others. If a prioritized flow runs faster than the claimed throughput, its VirtualClock will eventually run ahead of the real time; hence, its packets will lose priority in service.

The VirtualClock algorithm uses this priority service function to multiplex flows that require guaranteed performance with datagram traffic that expects a best-effort service. Packets of datagram traffic are assigned a priority value of $-\infty$, i.e., they are stamped with a value of ∞. Therefore, datagram packets are always waiting at the end of the service queue whenever a queue exists. As a result, they can utilize only resources left over after flows with guaranteed performance requirements have been served. This allows random datagrams to be transmitted without making reservations and allows the network resources to be fully utilized by performance-insensitive traffic.

3.2.3 *VirtualClock as a Data Flow Monitor.* From another viewpoint, VirtualClock plays the role of a "flow meter" driven by packet arrivals. Thus it can be conveniently used for flow measurement. Because a VirtualClock is advanced according to the flow's specified average transmission rate, the difference between the VirtualClock and the real time indicates how closely a running flow is following its specified transmission rate. We monitor each flow by periodically comparing its VirtualClock with the real-time clock, providing feedback to the flow source whenever its actual throughput departs significantly from the reserved rate.

Given an average interval value in the flow set-up request, one way to monitor a flow is to check its meter, VirtualClock, after every AI time period. Such a measure, however, may react to traffic changes too slowly when the value of AI, which is specified by the user, is large. A derivative detector, which checks the amplitude of changes, will be able to catch misbehaving flows more quickly. Therefore we let the switch check each flow, $flow_i$, after receiving every $AIR_i (= AR_i \times AI_i)$ packets from it. This is equivalent to checking the flow after every AI seconds when the flow is transmitting at the specified average rate, but checking much sooner if the flow is sending faster. By counting the number of packets, traffic impulses can be quickly detected.

[4] Using time stamps for priority purposes does have a side-effect: packets of low priority can have their priority increased with time. We argue that, if the channel maintains a proper utilization, P can be set to a value longer than the resource contention period. If we define channel state from idle to next idle as an epoch, P should be much longer than the average epoch length. Therefore low priority load can be effectively hidden from high-priority flows. Only in the presence of misbehaving users may a channel be in busy state for long, in which case the misbehaving users will be detected, as described in Section 3.1, and proper control actions will be taken.

Had we used a specific time interval for measurement, we would have faced the dilemma of picking a period either too small for stable control or too large to detect overload promptly.

3.2.4 *Flow Monitoring: Burstiness Tolerance Versus No Credit Saving.* If a flow's VirtualClock is running behind real time (because the flow has been transmitting at lower than the specified rate), the difference between the two may be considered some sort of "credit" that the flow has built up. Because packet traffic is bursty, it may seem reasonable to allow such credit savings for later use. From a resource allocation viewpoint, however, unused resources are gone. If a flow were allowed to save up an arbitrary amount of credit, it could remain idle during most time of its session and then send all the data at the last second. Such behavior would violate the specified average packet rate over each AI period and may cause temporary congestion in the network. However, it would not be detected by the VirtualClock-monitoring methods, since the flow would have saved adequate credit beforehand. There is, therefore, a conflict between tolerating bursty transmissions and controlling network congestion. The parameter of *average interval* is chosen precisely to compromise the two. Data transmissions of a flow can vary within each average interval, but no credit may be saved from one average interval to the next. Each switch enforces this policy by resetting the VirtualClock to the real time at checking point if the former ever lags behind.

What should the network do with credit saved within an average interval? Simulation has revealed that, if such intra-AI credit savings are permitted, when a burst of packets arrives from a flow that has been idle for a while (within the current AI period), the burst can still cause sudden queueing increases to others. This is because VirtualClock is designed to tolerate flow variations within an average interval, which is primarily chosen by individual users according to their applications' need. Thus $VirtualClock_i$ can fall far behind the real time without not being checked until AIR_i packets have been received.

We conclude that, if VirtualClock is used for both service ordering and flow monitoring, the two purposes have a conflict between tolerating statistical variations within each AI time period and not allowing credit saving even within an AI period. Therefore a second control variable, named *auxiliary VirtualClock* (auxVC), is needed in order to take the arrival time of packets into account. When a burst of packets arrives very late in an average interval, although the VirtualClock value may be behind real time at that moment, use of auxVC will ensure the first packet to bear a stamp of the current time and each of the subsequent packets to bear a stamp value with an increment of Vtick to the previous one. These stamp values will then cause this burst of packets to be interleaved with packets arrived from other flows, if there are any, in the waiting queue.

By replacing VirtualClock by auxVC in the packet stamping, no longer can a flow increase the priority of its packets by saving credits, even within an average interval. VirtualClock retains its role as the flow meter that measures the progress of a statistical packet flow; its value may fall behind the real-time clock between checking points in order to tolerate packet burstiness within each average interval.

Another design issue we have not discussed so far is how to choose a proper threshold value, T, such that whenever ($VirtualClock_i$ – real time) > T, the switch can assume with confidence that $flow_i$ has indeed been transmitting too fast and that control actions are necessary. Intuitively, one might expect that the variations in a flow's data generation would cancel each other out over time and that the VirtualClock reading would fluctuate about real time within finite limits. If so, a moderate value of T would work well. We conducted a number of simulation tests to experiment with various threshold values. The test results show that, contrary to intuition, the difference between the VirtualClock and real time may exceed any fixed threshold, triggering false control actions even when the flows have specified the average throughput accurately. Below we give a simple analytical explanation and a proposed solution to this problem.

3.3 User-Behavior Envelope: Balancing Tolerable Burstiness with Monitorability

To make the analysis simple, first let us assume that packet arrivals within a flow follow a Poisson process. Let us also use reasonably large average intervals (e.g., AR = 5 packets/second, AI = 10 seconds) and assume that the VirtualClock is advanced only by packet arrivals. We are interested in how the difference between a flow's VirtualClock and the real-time clock may grow as time passes.

Let us partition packet arrivals from a Poisson source into equal time intervals of AI seconds, $T_1, T_2, \ldots, T_i \ldots$, and let P_i represent the number of packets arrived during T_i, D_i the difference between the number of actual arrivals and the expected value, and Sum_n the accumulation of D_i's over n time intervals. Thus,

$$D_i = P_i - AIR, \tag{1}$$

$$Sum_n = \sum_{i=1}^{n} D_i = \sum_{i=1}^{n} P_i - \sum_{i=1}^{n} AIR, \tag{2}$$

$$= (VirtualClock - \text{real time}) / Vtick. \tag{3}$$

The P_i's are independent, identically distributed (IID) random variables, so are the D_i's. Thus, Sum_n is a sum of n IID variables, and we have

$$Mean(Sum_n) = Mean(D_i) \times n = 0 \tag{4}$$

$$Var(Sum_n) = Var(D_i) \times n \overset{n \to \infty}{\to} \infty. \tag{5}$$

Sum_n represents a *random walk* process. Equation (5) indicates that, probabilistically, the value of Sum_n, i.e., the difference between a flow's VirtualClock and the real-time clock, may vary above any fixed threshold after the flow has run for long enough. This phenomenon has indeed been observed in simulations.

When VirtualClock is advanced either by packet arrivals or by the real time (at each periodic checking point), Sum_n in (2) becomes

$$Sum'_n = \sum_{i=1}^{n} D_i, \qquad D_i > 0. \tag{6}$$

Intuitively, the variance of Sum'_n should grow with n at least at the same rate as Sum_n.

It is also worth pointing out that the value of $Var(D_i)$ is application dependent; so is $Var(Sum'_n)$. If applications are allowed to transmit data with no constraint, the burstier the generation process, the bigger the D_i values will be. This fact adds to the difficulty in distinguishing whether a VirtualClock that is running ahead of the real time indicates a misbehaving flow or whether it is merely caused by large variations in data generation.

Facing this variance accumulation problem in flow measurement, we propose a *user-behavior envelope* (UBE) control as a solution: a flow source should constrain itself from sending more than AIR packets during an average interval. Under this UBE constraint, D_i (and hence Sum'_n) in (6) will become zero at the first switch hop from the source host. If packets do not clump together significantly while traversing through the network, as is the case in a VirtualClock controlled network,[5] the value of Sum'_n will remain small at subsequent switches en route. Thus, it is safe to choose the threshold value T to be the value of $AIR \times Vtick$. (Also see Zhang [25] for a complete description of the solution.)

3.3.1 *Justification for UBE Control.* We conclude from the above analysis that, without the UBE control at sources, it will be difficult for the network to correctly measure the average throughput of statistical data flows. Whenever control feedback is needed, UBE (or other similar measures) is necessary for the network to provide correct feedback information and to reduce false alarms.

We assume that the applications that generate flows, which can be either real-time applications or data retrieval processes that fetch data from storage, are able to adjust the generation rate in some way according to the UBE constraint. Either the data generation rate is controllable (as when the data sources are in support of bulk data transfer or are produced from variable-rate video/audio encoders), or some of the packets are marked droppable and can be safely discarded [16]; or the data in the excessive packets (i.e., those that would have been sent if there were no UBE control) can be encoded in subsequent packets.

Packet switching offers unbounded flexibility to users. A well-defined UBE is therefore necessary to counter-balance this flexibility. The widely

[5] Because packets are served in VirtualClock stamp order, packets from different flows are interleaved as much as possible The interleaving makes it very unlikely that packets from individual flows will clump into bursts. In our simulation tests, after the flow sources restrict the transmission by the UBE, the VirtualClock value only fluctuates over a small interval around real time.

employed window flow control algorithms have long enforced such constraints. A window flow controlled data connection is prohibited from having more than a certain amount of outstanding data in the network at any time. Our proposed UBE control somewhat resembles the window flow control mechanism in constraining data transmission, except that now the transmission "window" is opened up continuously by time rather than by acknowledgment returns. It is superior to window flow control in that, instead of waiting for acknowledgments (whose arrival times incur random delays) before further transmission, applications can schedule ahead in data generation. The overhead of processing and transmitting the acknowledgments is also eliminated.

Requiring self-constraints on users is a necessary cost, which ought to be recognized explicitly. Significant work remains to be done on how to design application protocols that can adjust themselves to such flow constraints.

3.3.2 *Resource Over reservation.* The above discussion leads to a related question: if the partial sum of a random data source can depart significantly from the average at a given moment, there will be flows that generate traffic at well above the specified average rate, as well as flows that transmit at well below the average rate. Further, for each flow, there may be periods of heavy data generation as well as periods of relatively low activity. Restricting a flow's transmission to fall within a fixed envelope means cutting off the high peaks. The overall transmission rate, therefore, may average lower than the specified value, and the resources may be over reserved.

Simulation tests have indeed manifested such resource over-reservation (see the simulation results in Section 4.2.1). When a statistical data source restricts its transmission according to the UBE, its actual throughput is lower than the expected average. Choosing an adequate average interval can make this difference negligible. One cannot, however, totally eliminate it by a finite-average interval as long as flow sources are statistical processes.

It is also conceivable that a user, predicting a high variation in its data generation process that cannot fit into a reasonable average interval, may purposely specify an average rate higher than the estimated mean in order to minimize the cut-off by the UBE constraint, even if such over-reservation may be associated with a cost.[6] Besides a reduced constraint on its data transmission, a flow that over reserves resources may also receive a better delay performance, because its VirtualClock will advance by a smaller step at each packet arrival. This is an interesting area for further exploration.

3.4 Summary

Although a VirtualClock-controlled packet-switched network mimics the operation of a TDM system, there exist fundamental differences between the two. One is that the VirtualClock algorithm merely *orders* packet transmission; it does not change the statistical sharing nature of packet switching—the network forwards all packets as quickly as possible and as long as resources

[6] However, the case where malicious users over reserve resources to deny service to others must be prevented by proper charging or authentication mechanisms.

are available. Another major difference is that a VirtualClock-controlled network can support arbitrary throughput rates for individual flows. The network reservation control algorithm determines what fraction of the resources each flow may consume *on average*; the VirtualClock algorithm determines, if more than one packet is waiting in a switch, which one should go first based on the reserved transmission rates of the flows through that switch.

Summing up, the *VirtualClock* algorithm should be able to ensure the following functionalities:

— Every flow receives guaranteed service as measured by its reserved throughput rate.

— Flows running faster than the reserved throughput rate will be detected by their fast-running *VirtualClock*. Depending on resource availability, they may be punished by longer queueing delays, or even packet losses, while other flows will not be disturbed.

— Multiple-level priority services can easily be provided.

— Packets from different flows are maximally interleaved, which is an important contributor to good network performance [4].

Extensive simulations have been conducted to verify the above conclusion. We discuss the simulation results in the next section.

4. SIMULATION RESULTS

In this section, we first describe the network model used in our simulations and then present some of the results demonstrating that VirtualClock provides a fair service, supports diverse throughput rates, and builds firewalls between individual flows. Some interesting results showing the impact of VirtualClock on packet-queueing delays will also be discussed briefly. It is not possible, however, to include all the simulation results in a single paper; in particular, testing results of priority flows have been omitted. Interested readers are referred to Zhang [25] for a more complete presentation of the simulation results.

4.1 Simulation Model

4.1.1 *Network Topology.* A simple network topology model is used in the simulations (see Figure 2). It is constructed of four switches in a row. Each link is a duplex communication channel (below we use the words *link* and *channel* interchangeably). All the switches and links are assumed to provide error-free transmission. The link from each host to its attached switch has a bandwidth of 10 Mbps and a propagation delay of 1 msec. The three switch-to-switch links each has a bandwidth of 400 Kbps and a propagation delay of 5 msec. All the four switches have a moderate buffer pool size of 100 packets. The switches are assumed to have adequate capacity to process incoming packets from all attached links at their full speed.

Although the network bandwidths used in the simulation model are relatively low, we expect to be able to extend the results presented below directly

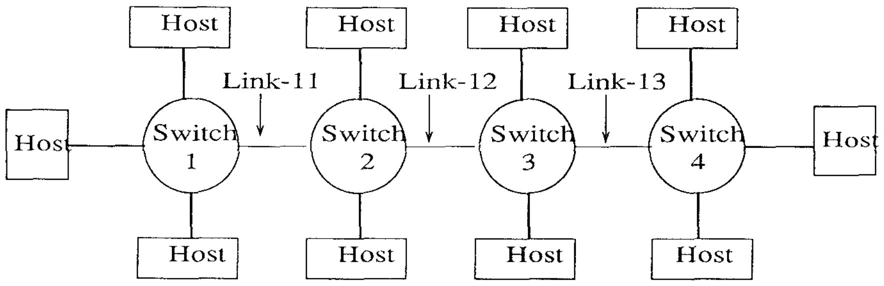

Fig. 2. The simulation topology.

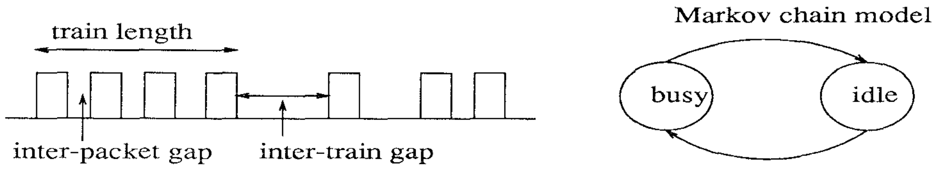

Fig. 3. Packet train model.

to higher-speed environments. We hold this belief because it is the channel utilization that determines the queueing distribution. If we scale up both the transmission rates of flows and the network bandwidths by a factor of 1000, for example, the channel utilization will remain unchanged and so will the queue length distribution. The queueing delay, however, will be decreased by a factor of 1000.

4.1.2 *Data Generator Model.* Data generation is an application-dependent random process. Because packet-switched networks must serve various types of current and potential applications, a universally accurate data generation model does not exist. In previous network performance studies, a commonly used data source model is an infinite data source where there is always data ready to be sent [11]. This model, although it is simple and stresses the network performance under heavy load, lacks the randomness that is a typical feature of packet traffic. Another commonly used data generation model is the Poisson-arrival model. There is a widely held suspicion, however, that use of the Poisson model may not result in a realistic performance estimate. We chose to use a train model proposed by Jain and Routhier [10] for data generation in all of the simulations discussed below, except where the data generation models are explicitly specified.

Modeling each packet as a railroad car, a group of packets following one another closely is modeled as a *train.* The generation process of a packet train model can be described by three parameters: train length, intertrain gap, and interpacket gap (see Figure 3).

In simulation tests, we model the train length as a geometrically distributed random variable, the intertrain gap as an exponentially distributed random variable, and set the interpacket gap to a constant of 1/2 AR. With the above parameters, the packet trains fit into a two-state Markov chain model. Many applications can be coarsely modeled by a Markov chain (probably with more states) [21, 18]. For simplicity, we assume that all data packets have a constant size of 250 bytes. All the simulation tests discussed below also assume long-lasting data flows, since our focus is mainly on how well the VirtualClock algorithm performs, rather than on how well the network can handle dynamic data flows.

4.1.3 *Misbehaving Data Sources.* As a measure of robustness, a network control algorithm must be prepared to handle users that do not obey the control rules. We call them *misbehaving users.* This group does not include *malicious* users who attack purposely. We simulate misbehaving users as data sources that transmit much faster than the specified rate and that do not respond to network control messages.

4.2. Simulation Results

4.2.1 *Flows with the Same Throughput Requirements.* We first present the results from a simulation with the following traffic load: there are total 60 flows, each generating data in packet trains with a mean of 10 packets/sec (20 Kbps) and requesting an average throughput of 10 packets/sec. Flows 1 through 24 follow 1-hop paths, flows 25 through 48 2-hop paths, and flows 49 through 60 3-hop paths. (The hop count of a flow is the number of the switch-to-switch link(s) it crosses.) The sources and destinations of the 60 flows are more or less uniformly distributed. Later we will refer to this test as Test One.

The goal of this test was to demonstrate the VirtualClock algorithm's performance under heavy load. There were 18 flows on each of the inter-switch links in each direction, driving the link utilization above 85 percent. The test simulated a 10-minute run of the real system. The measurement statistics in both directions for Link-12 are given below as a sample of the network performance.[7] The link utilization is averaged over every 100-msec period. The queue length measures the number of packets in the queue, including the one being transmitted; "99-tile" means the 99th percentile of the queue length samples. The effective throughput is the number of packets delivered successfully from end to end. The total loss is the number of packet losses during the entire simulation run.

Switch Performance with Homogeneous Flows

Swit	Link	Utilization		Queue Length		
ID	ID	mean	dev	mean	dev	99-tile
2	12	0.86	0.11	2.64	1.8	10
3	12	0.86	0.11	2.61	1.7	9

Effective throughput: 584 packets/sec
Total loss: 0

[7] Due to the disk space limitations, it is impossible to log queueing data for all the links.

Dividing the 60 flows into three path-length groups, we computed the average throughput and the average queueing delay of each group below. Here the queueing delay is the waiting time each packet experienced in the queue(s), excluding its own transmission time.

	Throughput (packets/sec)	Queueing Delay (msec)
1-hop flow	9.59	7.76
2-hop flow	9.58	14.58
3-hop flow	9.62	22.37

Converting the packet-waiting time to the queue length (it takes 5 msec to transmit a 250-byte packet over a 400 Kbps link), we can see that the two measurements agree with each other (remembering that the queue length includes the packet being transmitted). Also notice that, due to the effect of the source UBE control, the actual average throughput is slightly lower (about 4 percent) than the expected value, as we have discussed earlier.

Summarizing the test results, we see that

— the network meets the flows' average throughput requirements,
— the average queueing delay is low,[8]
— the network load is stable and congestion free, and
— the network provides a fair service, independent of the path length of the flows.

4.2.2 *Supporting Diverse Flow Throughput.* Next we simulated flows with different throughput requirements, varying from 5 packets per second to 50 packets per second, as specified in the following table.

Diverse Throughput Rate of Flows

Throughput (packets/sec)	Flow ID
50	1, 18, 35
30	8, 25, 36
20	3, 12, 20, 29, 37
10	2, 4, 5, 6, 7, 9, 10, 11, 13, 14, 15, 19, 21, 22, 23, 24, 26, 27, 28, 30, 31, 32
5	16, 17, 33, 34

Among the total of 37 flows, flows 1 through 17 are 1-hop flows, 18 through 34 are 2-hop flows, and the rest are 3-hop flows. The test simulated a

[8] As a point of reference, an M/D/1 queue's average length under the same utilization would be around 4 packets or an average waiting time of 15 msec.

10-minute run of the real system and the results are presented in the same way as before.

Switch Performance with Diverse Throughput Flows

Swit ID	Link ID	Utilization mean	Utilization dev	Queue Length mean	Queue Length dev	Queue Length 99-tile
2	12	0.81	0.14	2.29	1.41	8
3	12	0.82	0.12	2.34	1.60	9

Effective throughput: 564 packets/sec
Total loss: 0

Again, we show the average throughput and queueing delay of the flows averaged by path length groups.

Flow Performance with Diverse Throughput Rate

	Average Throughput (packets/sec)				
Expected	50	30	20	10	5
1-hop	48.2	29.0	19.3	9.6	4.7
2-hop	48.3	28.8	19.0	9.6	4.9
3-hop	47.8	29.0	19.4		
	Average Queueing Delay (msec)				
rate	50	30	20	10	5
1-hop	5.6	4.2	5.2	10.8	12.3
2-hop	8.5	8.3	7.8	17.6	21.3
3-hop	9.5	8.0	10.7		

These results demonstrate that the VirtualClock algorithm provides the users with their expected throughput; different path lengths show no effect on either the flows' throughputs or delays. The different throughput rates of the flows, however, do have a slight impact on the average queueing delay: flows with lower throughput seem to experience higher queueing delays. This is so because their VirtualClocks tick by bigger increments; one packet arrival may advance the VirtualClock so much that the next packet has to wait to let one or more packets from higher-speed flows, which arrived in a burst, pass by first.

4.2.3 *Building Firewalls between Flows.* Here the test condition is changed back to that of Test One, except that every 6th flow is now a misbehaving one: it sends at 5 times the specified rate and does not respond to network control. The test simulated a 5-minute run of the real system.

Switch Performance in the Presence of Misbehaving Users

Swit ID	Link ID	Utilization mean	Queue Length mean	Queue Length dev	Queue Length 99-tile
2	12	1.0	47.4	7.65	65
3	12	1.0			

Effective throughput: 680 packets/sec
Total loss: 47,106 packets (all from misbehaving users)

Performance of Normal Flows in the Presence of Misbehaving Users

	Throughput (packets/sec)	Queueing Delay (msec)
1-hop flow	9.59	8.33
2-hop flow	9.64	14.82
3-hop flow	9.65	16.36

The above results show that normal flows are well protected from the few misbehaving ones; no one loses a single packet. Also not that, even though the misbehaving users drive the link utilization to 100 percent, the queueing delay of the normal flows remains about the same as before. The 3-hop flows even receive a lower queueing delay than in Test One, because all packets from the few misbehaving flows are put at the end of the service queues, making normal flows see a lower utilization. The VirtualClock algorithm builds firewalls between flows both in terms of the throughput and of the queueing delay.

4.2.4 *Effects of VirtualClock on Queueing Delay.* The major role of VirtualClock is to build firewalls among flows in statistical multiplexing and to meter the average volume of statistical data flows. It should be made clear that VirtualClock does not contribute directly to queueing delay reduction. Rather, it helps indirectly by interleaving packets from different flows so that packets of each flow will not bunch together, by assuring individual flows their reserved throughput rates so that each flow will experience minimal queueing delay if it is transmitting at the reserved rate, and by feeding back the monitoring information to flow sources so that any potential overloading can be quickly corrected.

In addition, simulation results also reveal a useful side-effect of the VirtualClock algorithm: VirtualClock allocates the queueing delay of each flow according to the burstiness in the flow's data generation pattern, as we show next.

4.2.5 *Queueing Delay of Different Data Generation Patterns.* Statistical multiplexing absorbs randomness and burstiness in individual flows' data transmissions. Nevertheless, highly bursty data arrivals can still result in significant queueing delays even when the average load is held below the network capacity. Because of the strict service ordering enforced by VirtualClock, however, a higher burstiness in a flow's data generation results mostly in an increase of its own queueing delay.

This effect is demonstrated by the results of a simulation test with three different data generation patterns, constant rate, Poisson arrival, and packet train. The test condition is the same as that in Test One except that the last four flows are removed to lower the link utilization (the measured utilization in this test is 78 percent),[9] and the data generation patterns are changed. The data generation models of flows 1 through 48 repeat the patterns of two

[9] Experiments show that when the utilization is too high (say above 80 percent) the difference in queueing delays among different data source models gradually diminishes.

constant-rate sources, two Poisson arrival sources, and two packet train sources. Flows 49 through 56 repeat the pattern of one constant-rate source, one Poisson source, and two packet train sources. The average and deviation of the queueing delays of all the flows are summarized below, sorted by the path length groups.

Queueing Delay Statistics of Diverse Data Patterns

	Average Delay (msec)			Delay Deviation		
	1-hop	2-hop	3-hop	1-hop	2-hop	3-hop
Constant rate	1.76	4.50	5.57	1.58	3.40	4.09
Poisson arrival	4.55	8.37	11.40	6.44	9.74	11.15
Packet train	6.49	10.47	15.25	9.50	12.17	15.49

Among the three data generation models, packet train has the highest burstiness. Consequently, the flows generated by packet trains attained the highest queueing delays and delay variations. It seems fair to make highly bursty data sources bear the bulk of the consequence. It is also possible for an individual flow source to adjust its queueing delay and delay variance by adjusting its own data generation patterns.

In particular, notice that the flows of the Poisson data sources have both a lower average delay and a smaller delay deviation than the flows of packet trains. By using the packet train model instead of the Poisson model in simulation tests, we have stretched the performance of the VirtualClock algorithm, demonstrating its enhanced robustness for a wider range of data generation patterns.

5. RELATED WORK

Before concluding, we compare the VirtualClock algorithm with a few others that have been proposed for network traffic control, mainly fair queueing, a schedule-based approach, and the Leaky-Bucket algorithm.

5.1 Fair Queueing

Fair queueing is a simple control strategy that provides all users with an equal allocation of network resources. Similar to the round-robin-scheduling algorithm often used in operating systems, the basic idea of fair queueing is to transmit data from each user in turn. Hahne [8] and Demers et al. [3] have done extensive analysis and simulation work on the performance of fair-queueing algorithms.

VirtualClock can be considered as performing a fair-queueing function, where the fairness is defined to be assuring each user the requested throughput. In fact various queueing policies can all be implemented by a simple computation on the VirtualClock value. In particular, the VirtualClock algorithm shares a number of features with the fair-queueing algorithm proposed by Demers et al. [3]. A major difference between the two is that VirtualClock is based on the resource reservation. Therefore, instead of allocating resources equally among all present users, VirtualClock is able to allocate, and enforce the usage of, any specific amount of resources for each user and to monitor individual flows against their reservations for control feedback.

5.2 Schedule-Based Approach in Data Flow Control

Mukherji has proposed a schedule-based approach to data traffic control [14]. In this approach, channel bandwidths are divided into equal time frames; each frame has a fixed number of slots, which are assigned to individual users. A user can send a packet by using its own slot or by using a slot whose designated user has no packet to send at the moment. Each user has an auxiliary flow control window that limits the number of packets a user may send by using others' slots.

The VirtualClock algorithm takes a similar approach of reserving resources for individual users. However, instead of assigning specific channel slots to individual users, VirtualClock orders packet service sequences according to the reservation. Hence it achieves the same functionalities of the Mukherji algorithm, but with more flexibility in handling different channel bandwidths and different user throughput demands.

5.2.1 *Leaky-Bucket.* Leaky-Bucket has been suggested as a network access algorithm for high-speed networks [17, 20]. Various versions of the algorithm have been proposed. A simple model, described by Turner [20], works in the following way: each switch at the network entrance puts packets from each data flow into a corresponding bucket that has a fixed size. The bucket opens periodically to emit packets for transmission. When the bucket is full, incoming packets are discarded. In another version of Leaky-Bucket, credits for each flow are generated at a constant rate, and a certain number of credits (up to the bucket size) can be saved. Arriving packets are transmitted immediately if the corresponding flow has credits; otherwise the packets are either dropped or stamped with a droppable mark and are then transmitted (if possible) at a low priority.

Packet-switching networks should tolerate variations in packet arrivals while controlling flows' average transmission rates. The first version of Leaky-Bucket, as described above, reduces statistical multiplexing because packets are transmitted at a constant rate rather than whenever the channel is available. The second version may result in bursty transmissions if buckets in the switch are served in FIFO order. The VirtualClock algorithm avoids those drawbacks by merely ordering packet service without reducing statistical sharing; at the same time, it also makes packets from different flows maximally interleaved.

The major difference between Leaky-Bucket and VirtualClock, however, is that the former is an *admission control* algorithm while the latter is a *service order control* algorithm. Leaky-Bucket enforces control at the network entrance and determines whether an incoming packet should be accepted. Once packets are accepted, Leaky-Bucket has no further control over the order of service. Thus it may not make maximal use of the resources if the admission policy is too conservative. Neither can it discriminate among packets according to different delay requirements. VirtualClock can also be used for admission control (when a flow's VirtualClock is running too fast, further packets from that flow can be rejected). Its main merit, however, is in controlling service orders. VirtualClock determines the service order of packets from all users. The action is applied right at the multiplexing point, controlling exactly which packet may be served next and what share of the resources individual users can receive.

6. SUMMARY AND FUTURE RESEARCH

In this paper we have presented VirtualClock as a new traffic control algorithm for packet-switched networks. Its fundamental merit arises from its imitation of a TDM system in a statistically multiplexed packet network, achieving the desired properties of both approaches. A network controlled by VirtualClocks maintains the statistical multiplexing flexibility of packet switching while ensuring each flow its reserved average transmission rate.

All data traffic control algorithms place certain constraints on users. In the VirtualClock algorithm, such constraints are expressed as a *user-behavior envelope* (UBE), which states that each flow source must refrain from sending more than $(AR \times AI)$ units of data over each AI time period. Although the UBE is introduced as a solution to our specific problem in using VirtualClock for flow measurement, we believe that a data transmission constraint such as UBE should be a mandatory part of all rate-based traffic control systems in general. How to design application protocols that can adjust themselves to such flow constraints is an interesting and important area for further study.

We are currently developing a prototype implementation of the VirtualClock algorithm, using Sun Microsystem's Sparcstation as the packet switch box. Our experience to date indicates that the algorithm itself is simple and adds minimal overhead to per-packet-processing time. A variable overhead cost in our implementation is the time to insert packets into the ordered service queues, since we use a simple linear search at present. In our limited local test environment we have not been able to build up long packet queues at the switch and thus have not noticed any significant processing overhead. Nevertheless, we are currently looking into better queue insertion algorithms in order to maintain a low overhead even in the presence of substantially long packet-waiting queues.

We also plan further exploration of the performance of the VirtualClock algorithm under more bursty traffic. Some preliminary simulation tests indicate that, with the traffic burstiness degree increased from two to eight, the network can maintain the same queueing delay by a modest decrease in the channel utilization. However, more rigorous testing is needed to achieve more quantitative conclusions.

Design of, and experimentation with, the VirtualClock algorithm has also raised a number of other interesting issues for future study. First, it may be attractive to applications with stringent delay requirements to slightly over-reserve resources in exchange for an improved transmission delay. More work is needed to investigate a quantitative relation between the percentage of resource over-reservation and the corresponding delay reduction. Another interesting area to explore is to provide a bounded transmission delay and guaranteed delivery to emulate services provided by a circuit-switched network. Although the VirtualClock algorithm, as described in this paper, shows low queueing delay in simulation tests, it does not guarantee bounded transmission delays nor loss-free data transmission. The algorithm is being further developed to achieve these properties.

As a final remark, the concept of the *average interval* seems particularly interesting. The VirtualClock algorithm uses two parameters, average rate (AR) and average interval (AI), to describe and control statistical data flows. As a tuning knob between the system constraints and the user flexibility, AI sets the bound of permitted burstiness in the transmissions of data sources. By tuning the AI value, we can tune a data transmission network along a continuous spectrum, with a TDM system at one end and an uncontrolled datagram network at the other. Similar choices in using AI as a data flow parameter are also being explored by others [5, 12].

ACKNOWLEDGMENT

This paper is based on my thesis work completed at the Massachusetts Institute of Technology. I wish to thank my thesis supervisor, David D. Clark, for his guidance and input. I also wish to thank Scott Shenker and the anonymous reviewers for their helpful comments on earlier versions of this paper. Finally, I wish to thank Dan Swinehart for his effort and humor in improving the readability of this paper.

REFERENCES

1. CHERITON, D. VMTP: A transport protocol for the next generation of communication systems. In the *Proceedings of SIGCOMM'86* (Stowe, Vt., Aug. 1986).
2. CLARK, D., LAMBERT, M., AND ZHANG, L. A high throughput bulk data transfer protocol. In the *Proceedings of SIGCOMM'87* (Stowe, Vt., Aug. 1987).
3. DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and simulation of a fair queueing algorithm. In the *Proceedings of SIGCOMM'89* (Austin, Tx., Sept. 1989).
4. DESCLOUS, A. Contention probabilities in packet switching networks with strung input processes. In *Proceedings of the 12th Teletraffic Congress* (Turin, 1988), 51A.1.1-1.7.
5. FERRARI, D. Real-time communication in packet-switching wide-area networks. Tech. Rep. TR-89-022, International Computer Science Institute, Berkeley, Calif., May 1989.
6. FERRARI, D., AND VERMA D. A scheme for real-time channel establishment in wide-area networks. *IEEE J. Selected Areas Commun. 8*, 3 (Apr. 1990), 368-389.
7. FERRARI, D. Client requirements for real-time communication services. *IEEE Commun. Mag. 28*, 11 (Nov. 1990), 65-72.
8. HAHNE, E. L. Round robin scheduling for fair flow control in data communication networks. Ph.D. thesis, MIT, Dec. 1986.
9. GROVER, G., AND BHARATH-KUMAR, K. Windows in the sky—Flow control in SNA networks with satellite links. *IBM Syst. J. 22*, 4 (1983), 451-463.
10. JAIN, R., AND ROUTHIER, S. Packet trains—Measurements and a new model for computer network traffic. *IEEE J. Selected Areas Commun. SAC-4*, 6 (Sept. 1986), 986-995.
11. JAIN, R., RAMAKRISHNAN, K., AND CHIU, D. Congestion avoidance in computer networks with a connectionless network layer. In *Innovations in Internetworking*, Artech House, 1988.
12. KALMANEK, C., KANAKIA, H., AND KESHAV, S. Rate controlled servers for very high speed networks. In the *Proceedings of GLOBCOM'90* (San Diego, Calif., Dec. 1990), 12-20.

13. LAMBERT, M. An end-point adaptive rate control strategy for the NETBLT protocol. Unpublished research notes, MIT Lab for Computer Science, May 1988.
14. MUKHERJI, U. A schedule-based approach for flow-control in data communication networks. Ph.D. thesis, MIT, Feb. 1986.
15 NAGLE, J. Congestion control in TCP/IP internetworks. *ACM Comput. Commun. Rev 14*, 4 (Oct 1984).
16. PETR, D., DASILVA, L., AND FROST, V. Priority discarding of speech in integrated packet networks. *IEEE J. Selected Areas Commun. 7*, 5 (June 1989), 644-656.
17. RATHGEB, E. P Comparison of policing mechanisms for ATM networks. In *Proceedings of the 3rd RACE Workshop* (Paris, Oct. 1989). Paper 21.1
18. SEN, P., MAGLARIS, B., RIKLI, N., AND ANASTASSIOU, D Models for packet switching of variable-bit-rate video sources *IEEE J. Selected Areas Commun. 7*, 5 (June, 1989), 865-869.
19. TCP-IP mailing list. TCP-IP mailing list is a special-interest-group mailing list moderated by the Network Information Center (NIC) located at SRI. In TCP-IP mail discussion, there have been numerous observations of malfunctioning hosts in the ARPA Internet.
20. TURNER, J. New directions in communications (or Which way to the information age?). *IEEE Commun. Mag. 24*, 10 (Oct. 1986), 8-15.
21. TURNER, J. Personal communication, 1989.
22. TYMES, L. Routing and flow control in Tymnet. *IEEE Trans. Commun. COM-29*, 4 (April 1981), 392-398.
23. WEINRIB, A VirtualClock and Leaky-Bucket: Flow control protocols for high-speed networks. In the *Proceedings of International Workshop on Protocols for High-Speed Networks* (Nov 1990).
24. ZHANG, L. Some thoughts on the packet network architecture. *Comput. Commun. Rev. 17*, 1&2 (Jan./Apr. 1987), 3-17
25. ZHANG, L. A new architecture for packet switching network protocols. Ph.D. thesis, Dept. of Electrical Engineering and Computer Science, MIT, July 1989.

Received August 1990; revised February 1991; accepted March 1991