SIGCOMM 1990

VirtualClock: A New Traffic Control Algorithm for Packet Switching Networks

Lixia Zhang
XEROX Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304
lixia@parc.xerox.com

Abstract

A challenging research issue in high speed networking is how to control the transmission rate of statistical data flows. This paper describes a new algorithm, VirtualClock, for data traffic control in high-speed networks. VirtualClock maintains the statistical multiplexing flexibility of packet switching while ensuring each data flow its reserved average throughput rate at the same time. The algorithm has been tested through simulation.

1 Introduction

High-speed networking introduces challenging issues in data traffic control. One is the large product of round-trip-time and channel bandwidth which makes it possible that at any given time a large number of packets can be stored in the "pipe". Another issue is the stringent performance requirements raised by new applications, such as real time voice and video. These applications often require a bounded transmission delay but possibly with a relaxed demand on error recovery. These new features makes it difficult for the conventional window-based flow control mechanisms, which have served well for reliable data transfer applications in low speed network environments, to meet the new challenge. Therefore, rate-based traffic control algorithms for packet-switching networks have become a focus of research in recent years.

A difficult issue in rate-based traffic control is 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 flexibility of statistical multiplexing. This paper introduces a new design, called *VirtualClock*, as a traffic control algorithm for high-speed packet switching networks. VirtualClock controls the average transmission rate of data flows, enforces each user's resource usage according to its reservation, builds firewalls among flows, and supports multi-priority transmissions. The algorithm has been tested extensively through simulation.

In this paper we discuss the design of the VirtualClock algorithm, its fundamental properties, and present some of the simulation results. We then compare the VirtualClock algorithm with a few others that have been proposed for network traffic control, the schedule-based approach [6], fair-queueing [1, 3], and Leaky-Bucket [8, 9].

2 VirtualClock Algorithm

2.1 Design Goals

VirtualClock was designed as part of a new network architecture, the Flow Network [10]. A Flow Network provides users with guaranteed performance by requiring explicit resource reservation and by employing rate-based traffic control. It models a user's data transmission demand as a *flow*, the switch processing power and channel bandwidths as distributed resources, and employs data rate control mechanisms to regulate the usage of the resources to meet the demand. As part of the network control mechanisms, VirtualClock was designed to provide the following functionalities:

1. To support diverse throughput requirements from various applications by enforcing the resource usage according to each flow's average throughput reservation.

2. To monitor average data flows and provide measurement input to other network control functions.

3. To provide firewalls among individual data flows.[1]
4. To preserve the full flexibility of statistical multiplexing of packet switching.

2.2 VirtualClock: First Outline

The idea of VirtualClock was inspired by the Time Division Multiplexing (TDM) system. A TDM system completely eliminates interference among users because individual user channels can transmit only during specific time slots. The capacity is wasted, however, when a slot is given to a flow that has no data to send at that moment; also the channel bandwidths allocated to each user are pre-fixed rather than dynamically adjustable.

We would like to achieve the firewalls of a TDM system as well as to preserve the flexibility of statistical multiplexing of packet-switching. A TDM system is driven by a real time clock; a statistical multiplexing system may use a *virtual clock* concept in a similar way. To make a statistical data flow resemble a TDM channel, we may imagine that arriving packets from the flow were having a constant rate in a virtual time space, so that each packet arrival would indicate that one slot time period had passed.

Following this thought, we assign each data flow a *VirtualClock* which ticks at every packet arrival from that flow; the tick step is equal to the mean inter-packet gap (assuming a constant packet size for the moment). In this way the VirtualClock reading tells the expected arrival time of the packet. If a flow sends packets according to its specified average rate, its VirtualClock reading should be in the vicinity of the real time. To imitate the transmission ordering of a TDM system, each switch may stamp packets by the flows' VirtualClock values and use the stamps to order transmissions, as if the VirtualClock stamp were the real time slot number in the TDM system.

We sketch an implementation outline below. For each switch,

1. When received $flow_i$'s set-up request, compute the value $Vtick_i = 1/AR_i$, where AR_i is the average transmission rate indicated in the request.
2. Upon the arrival of the first packet from $flow_i$, $VirtualClock_i \leftarrow$ real time.
3. Upon receiving each packet from $flow_i$, advance $VirtualClock_i$ by $Vtick_i$, then stamp the packet with the value of $VirtualClock_i$.
4. Transmit packets by the order of increasing VirtualClock stamp values.
5. When the switch runs out of buffer space, drop the last packet from the queue.

If packets have variable sizes, the value of Vtick in the above can be chosen proportionally to the size of each packet.

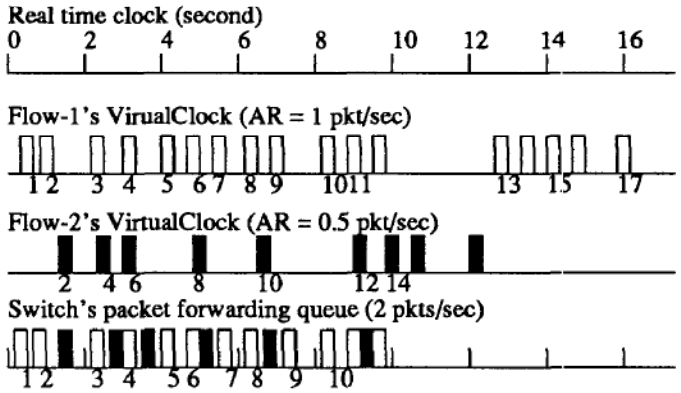

Figure 1: Real time, Virtual Clock, and packet processing order.

One major difference between a VirtualClock controlled packet switching network and a TDM system is that the VirtualClock algorithm merely *orders* packet transmission without changing the statistical sharing nature of packet switching – the network forwards all packets as long as resources are available. Another major difference is that the packet network can support arbitrary throughput rates of individual flows. The network reservation control determines how much share of the resources each flow may take *on average*; the VirtualClock algorithm determines, if more than one packet is waiting, which packet should go next based on the flows' reserved transmission rates.

2.2.1 VirtualClock as a Data Flow Monitor

From another viewpoint, VirtualClock plays the role of a "flow meter" driven by packet arrivals. Because it is advanced according to the flow's specified average transmission rate, the difference between the VirtualClock and the real time clock will show how closely a running flow is following its claimed rate. Therefore we

[1] As has been frequently observed in operational networks, users may sometimes misbehave, i.e. a user may not follow the network control protocol but rather transmit data at a high rate. Even if we assume no users have malicious intention, such misbehavior can still be caused by software or hardware failures, by protocol implementation errors, or even by protocol design errors [7, 5]. It is the responsibility of the network control to prevent misbehaving users from interrupting normal service to others.

can monitor each flow by comparing its VirtualClock with the real time clock periodically in order to provide feedback to flow sources if their actual throughput ever depart significantly from the reserved rate.

How often should a flow's VirtualClock be checked? A reasonable *average interval* (AI) is needed over which we can check a flow's average transmission rate while tolerating burstiness and randomness in packet flows as much as possible. In [10] we argued that this AI value is application-dependent and should be provided in each flow's reservation request. With the AI value given, a flow can then be monitored by checking its meter, VirtualClock, after every AI time period. Such a measure, however, may react too slowly, especially when the value of AI is large. A derivative detector will be able to catch misbehaving flows more quickly — that means checking the amplitude of changes.

Therefore we let the switch check each flow, $flow_i$, after receiving every $AIR_i (= AR_i \times AI_i)$ packets from it. By counting the number of packets, traffic impulses can be quickly detected. Had we used a specific time interval for measurement, we would have faced the dilemma of picking a period that is either too small to keep control stable, or too big to detect overload promptly.

The VirtualClock algorithm monitors data flows in the following way:

- At $flow_i$ setup, the switch computes the value $AIR_i = AR_i \times AI_i$.
- Upon receiving each set of AIR_i packets,
  - ($VirtualClock_i >$ real time) indicates that the flow has been sending faster than the specified rate. If $VirtualClock_i$ is ahead by more than a threshold value, proper control actions should be taken.
  - If ($VirtualClock_i <$ real time), then $VirtualClock_i \leftarrow$ real time.

We see that VirtualClock is driven either by incoming packets or by the real time, whichever runs faster. No credit can be saved over an average interval, even when the flow runs more slowly than the specified rate. From a resource allocation viewpoint, unused resources are gone; if a flow were allowed to accumulate credits, it could increase its priority by idling for a while and then transmitting in bursts, which would cause packets from other flows to experience a sudden increase in queueing delay, or even switch buffer overflow.

2.2.2 Providing Priority Service

Priority service can also be easily accommodated by the VirtualClock algorithm, i.e. by decreasing a flow's VirtualClock by a certain amount, P, at the start of a flow.

$$VirtualClock \leftarrow (real\ time - P)$$

where P represents the priority. The value of P should be big enough to separate priority flows far apart from the rest in the transmission queue.[2] This priority, however, does not allow the former to take unfair advantage of other flows. If a prioritized flow runs faster than the claimed throughput rate, its VirtualClock will eventually run ahead of the real time and hence its packets will lose priority in transmission.

2.2.3 Building Firewalls among Flows

Ordering packet transmission by VirtualClock stamps assures that, although an aggressive flow may take up idle resources, it cannot disturb the network service to other flows. The resource reservation control ensures that no congestion will occur if every flow transmits according to its reserved average throughput rate. In cases where one or more flows violates the reservation to cause congestion, flows that follow their specified throughput rate will not be affected, while the most offending flows will receive the worst service because their VirtualClocks will run too fast, hence their packets will be put at the end of the service queues, or even get dropped. The VirtualClock algorithm prevents interference among flows.

2.3 Further Revisions of VirtualClock Algorithm

The VirtualClock design, as described in the above, was tested through simulation and a few revisions were made to correct the problems discovered in the simulation, as explained below.

---

[2] Using time-stamp for priority purposes has a side-effect: low priority objects can have their priority increased with time. We argue that, if the channel keeps a proper utilization, P can be set to a value longer than the resource contention period, therefore low priority load can be effectively hidden from high priority flows. If we define channel state from idle to next idle as an epoch, P should be much longer than the average epoch length. 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 the previous section, and proper control actions will be taken.

2.3.1 No Credit Saving vs Flow Variation Tolerance

The first problem revealed by simulation tests is that, if a burst of packets arrives from a flow, $flow_i$, that has been idle for a while (within an AI period), the burst can still cause sudden queueing increases to others. This is because the algorithm is designed to tolerate flow variations within each average interval (which is primarily chosen by individual applications), and $VirtualClock_i$ has not been advanced since the last checking point, and will not be until AIR_i packets have been received.

To resolve this conflict, we assign each flow an auxiliary VirtualClock (auxVC), and revise step (3) of the packet stamping rule described in Section 2.2 in the following way:

- Upon receiving each packet from $flow_i$,
  1. auxVC $\leftarrow$ max(real time, auxVC);
  2. $VirtualClock_i \leftarrow (VirtualClock_i + Vtick_i)$, and auxVC $\leftarrow$ (auxVC + Vtick);
  3. stamp the packet with auxVC.

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

2.3.2 Need for A User-Behavior Envelope

As proposed in Section 2.2.1, the switch monitors a statistical flow by comparing its VirtualClock with the real time after receiving each set of AIR packets. An unresolved issue 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 control actions deem necessary.

A number of simulation runs were conducted to test various threshold values. The results show that, even when a flow generates packets by a Poisson process and uses a reasonably large average interval (e.g. AR = 5 packets/second, AI = 10 seconds), the difference between the VirtualClock and the real time may still exceed any fixed threshold and trigger false control actions. Intuitively, one might think that the variations in a flow's data generation over each average interval should cancel each other out, and hence the VirtualClock reading would stay within some finite vicinity of the real time. Close observations of simulation runs show that, contrary to intuition, these variations grow unbounded. Below we give a simple analysis of the observed phenomenon.

First let us 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 goes on.

Let us cut packet arrivals from a Poisson source into equal time intervals, $T_1, T_2, \ldots, T_i \ldots$, and let P_i represent the number of packets arrived during T_i, we have

$$D_i = P_i - AIR \quad (1)$$

$$Sum_n = \sum_{i=1}^{n} D_i \quad (2)$$

$$= (VirtualClock - RealTime)/Vtick \quad (3)$$

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

$$Mean(Sum_n) = Mean(D_i) \times n = 0 \quad (4)$$

$$Var(Sum_n) = Var(D_i) \times n \quad (5)$$

Sum_n represents a *random walk* process, and the value of $|Sum_n|$ is unbounded as $n \rightarrow \infty$. Equation (5) indicates that, probabilistically, the value of Sum_n, i.e. the difference between VirtualClock and the real time clock, may vary above any fixed threshold after the flow has run long enough.

When VirtualClock is advanced either by packet arrivals or by the real time, Sum_n in Equation (2) becomes

$$Sum_n = \sum_{i=1}^{n} D_i, \quad D_i > 0 \quad (6)$$

Intuitively, the variance of Sum_n in Equation (6) should grow with n not more slowly than linearly. Also note that the value of $Var(D_i)$ is application-dependent, so is $Var(Sum_n)$. This fact adds to the difficulty in distinguishing whether a VirtualClock which is running ahead of the real time indicates a misbehaving flow or whether it is merely due to large variations in data generations.

Facing this variance accumulation problem in flow measurement, we proposed a *user-behavior envelope (UBE)* as a solution: a flow source must constrain itself from sending more than AIR packets during each average interval. After flow sources restrict the transmission

within the above envelope, simulation tests show that the VirtualClock value is stabilized, varying around the real time. The value of AI is chosen to be the threshold, T. (Also see [10] for a complete description of the solution.)

Justification

We assume that flows' data generators, which can be either real-time applications or data retrieval processes fetching storage, are able to adjust the generation rate in certain ways according to the UBE constraint. Either the data rate can be adjusted without causing application performance degradation, 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. Appendix A shows a simple implementation of this user-behavior envelope.

Although the user-behavior envelope was introduced as a solution to our specific problem in using VirtualClock for flow measurement, we believe that it is a mandatory part of rate-based flow control systems in general. Packet switching offers unbounded flexibility to users, a well defined constraint is therefore necessary to counter balance this flexibility. The widely employed window flow control mechanisms provide a good example of such constraints — users restrict themselves from having more than a certain amount of outstanding data in the network at any time. Requesting self-constraints on users is a necessary cost, which ought to be recognized explicitly. Much work needs to be done on how to design application protocols that can adjust themselves to the constraints.

Resource Overbooking

The above discussion may have triggered a related question in the reader's mind: 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 data much above the specified average, as well as flows much below the average. And for each flow, there will be periods of heavy data generation and periods of relatively low activity. Restricting a flow's transmission by 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 overbooked.

Simulation tests indeed manifested such resource overbooking. When a flow with a statistical data source restricts its transmission according to the proposed envelope, its actual throughput is lower than the expected average. Enlarging the average interval can reduce this difference to a negligible value. One cannot, however, totally eliminate it by any finite average interval.

It is also possible that a user, predicting a high variation in its data generation process, may purposely specify an average rate higher than the estimated mean in order to minimize the cut-off by the user-behavior envelope constraint, even if such overbooking may be associated with a cost.[3] Besides a reduced constraint on its data transmission, a flow that overbooks resources may also receive a better delay performance, because its VirtualClock will be advanced by a smaller step at each packet arrival. One flow's overbooking will not have any negative effect on the performance of others, because the VirtualClock algorithm assures everyone the amount of its own reserved throughput.

2.4 VirtualClock: the Final Version

Below is a description of the final VirtualClock algorithm: at each packet switch,

- Upon receiving the first packet from $flow_i$, $VirtualClock_i \leftarrow$ real time.
- Upon receiving each packet from $flow_i$,
  1. auxVC $\leftarrow$ max(real time, auxVC);
  2. $VirtualClock_i \leftarrow (VirtualClock_i + Vtick_i)$, and auxVC $\leftarrow$ (auxVC + Vtick);
     If all packets have a constant size, $Vtick_i = 1/AR(packet/sec)$. If packets have variable size, the value of $Vtick_i$ should be computed from individual packet sizes.
  3. stamp the packet with the auxVC value.
- Transmit packets by the order of increasing stamp values.
- When the switch runs out of buffer space, drop the last packet from the queue.
- Upon receiving each set of $AIR_i (= AR_i \times AI_i)$ packets from $flow_i$,
  - if ($VirtualClock_i$ − real time) > threshold, control actions should be taken.
  - If ($VirtualClock_i <$ real time), $VirtualClock_i \leftarrow$ real time.

In handling priority flows, *real time* in the above should be replaced by $(real\ time - P)$.

The above *VirtualClock* algorithm can ensure the following functionalities:

[3] However, the case where malicious users overbook resources to deny services to others must be prevented by proper charging or authentication mechanisms.

- Every flow receives a fair service measured by its claimed transmission parameters.[4]
- Flows running faster than the claimed throughput rate will be detected by their fast running *VirtualClock*. 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, and flows with priority are prevented from taking unfair advantage of others.
- Packets from different flows are maximally interleaved, which is an important measure in keeping good network performance [2].

Extensive simulation tests have been conducted to verify the above conclusions [10]. Due to the space limit, however, only part of the simulation results will be presented in the next section.

3 Simulation Results

In this section, we first discuss the network model used in simulation tests, and then present the results showing that VirtualClock provides a fair service, supports diverse throughput rates, and builds firewalls between flows. Some interesting results showing the impact of VirtualClock on packet queueing delays will also be discussed briefly.

3.1 Simulation Model

3.1.1 Network Topology

A simple network topology model is used in the simulations (see Figure 2). It has 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 links from hosts to the attached switch have a bandwidth of 10 Mbps, and a propagation delay of 1 msec. The three switch-to-switch links have the same bandwidth of 400 Kbps and propagation

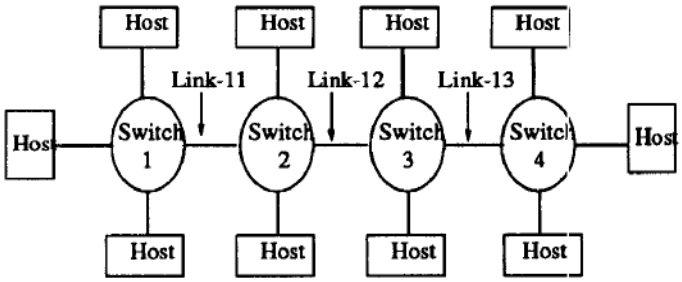

Figure 2: The simulation topology.

delay of 5 msec.[5] 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.

Although the network bandwidths used in the simulation model is relatively low, one should be able to extend the results presented below directly to a higher speed environment. It is the channel utilization that determines the queueing distribution. If we scale up both flows' transmission rates and the network bandwidths up by a factor of 1000, for example, the channel utilization will remain unchanged, so will the queue length distribution. The queueing delay, however, will be decreased by a factor of 1000.

3.1.2 Data Generator Model

Data generation is an application-dependent random process. Because the packet switching network is to serve multiple current and potential applications, a universally accurate data generation model does not exist. Most previous network performance studies have used the Poisson arrival model for data generation. There exist various speculations, however, that use of the Poisson model may not result in a realistic performance estimate. In [4], Jain and Routhier presented a *packet train* model based on their traffic measurement. We chose to use this train model for our data generation in the simulation tests.

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, inter-train gap, and inter-packet gap (see Figure 3). Packet trains fit into a Markov chain model of two states; it is one step forward from the Poisson arrival model. Many ap-

[4] The definition of fairness is a difficult subject. We consider it as a policy issue above the network control layer. A control algorithm should be able to support whatever fairness definition is given. This research assumes that the service parameters in each flow request have been checked by the fairness policy.

[5] The propagation of the speed of light in fiber is about 200,000 km/sec. A coast-to-coast span of the continental USA is 4000 km, with a propagation delay of 20 msec.

plications can be coarsely modeled by a Markov chain (probably with more states).

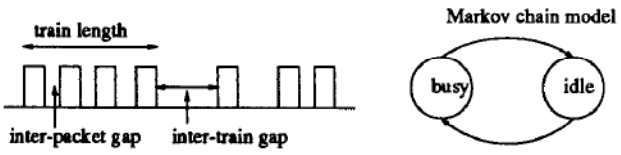

Figure 3: Packet train model.

In simulation tests, the train length is modeled as a geometrically distributed random variable. The inter-train gap is modeled as an exponentially distributed random variable. The inter-packet gap is set to $1/(2 \times$ *Average Rate*), i.e. the burstiness degree is 2. All data packets are assumed a constant size of 250 bytes.

3.1.3 Misbehaving Data Sources

As a measure of robustness, a network control algorithm must be prepared to handle users who do not obey the control rules. We call them *misbehaving users*. This group does not include *malicious* users who attack purposely. The simulated model of misbehaving users is a data source that transmits faster than the specified rate and does not respond to network control.

3.2 Simulation Results

3.2.1 Flows with Same Throughput Requirement

We first present the results from a simulation run with the following traffic load: there are total 60 flows, each generates data by the packet-train model with a mean of 10 packets/sec (20 Kbps) and requests an average throughput of 10 packets/sec. Flows 1 through 24 have a path of 1-hop, flows 25 through 48 2-hop, and flows 49 through 60 3-hop. The hop count of a flow is the number of the switch-to-switch link(s) it crosses. The sources and destinations of the flows are more or less uniformly distributed. Later we will refer to this test as Test-One.

The goal of this test is to demonstrate the network performance under heavy load. There are 18 flows on each of the inter-switch links, driving the utilization above 85%. The test simulated a 10 minute run of the real system. The measurement statistics of both directions of Link-12 are given below as a sample of the network performance.[6] The link utilization is averaged over every 100 msec period. The queue length measures the number of packets in the queue, including the one under transmission; "99-t" 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 whole simulation run.

Measurement statistics with homogeneous flows

Swit	Link	Utilization		Queue Length		
ID	ID	mean	dev	mean	dev	99-t
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

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

Notice that the actual average throughput is slightly lower than the requested value (about 4%), due to the user-behavior envelope restriction we discussed earlier. If we convert the packet waiting time to the queue length (it takes 5 msec to transmit a 250-byte packet over a 400 Kbps link), we see that the two measurements agree with each other (remembering that the queue length counts the packet being transmitted as well).

Summarizing the test results, we see that:

- The network meets the flows' average throughput requirement.
- The average queueing delay is low.[7]
- The network load is stable and congestion free.
- The network provides a fair service, independent of flows' path lengths.

[6] Due to memory limitations, it is impossible to log queueing data for all the links.

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

3.2.2 Supporting Diverse Flow Throughput

We also simulated flows with different throughput requirements as given below.

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, $1 \sim 17$ are 1-hop flows, $18 \sim 34$ are 2-hop flows, and the rest 3-hop. The test simulated a 10 minute run of the real system and the results are presented in the same way as before.

Measurement statistics with diverse throughput flows

Swit ID	Link ID	Utilization mean	Utilization dev	Queue Length mean	Queue Length dev	Queue Length 99-t
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 by the path length groups.

Flow performance with diverse throughput rate

	Average Throughput (pkts/sec)				
Desired	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		

The above results show that the VirtualClock algorithm satisfies the users with their expected throughput; different path lengths show no effect. The different throughput rates of the flows do have a minor impact on the average queueing delay though; lower throughput flows seem to experience a higher queueing delay. This is because their VirtualClocks tick by bigger steps; 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.

3.2.3 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 user: 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.

Measurement statistics in the presence of misbehaving users

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

Effective throughput: 680 packets/sec
Total loss: 47106 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

Recall that when a switch runs out of buffer space, it drops the last packet from the longest queue; and that after a switch detects a misbehaving flow it will take proper control actions. In this test, the control actions are first to send a certain number of control messages to each misbehaving flow requesting it to slow down; if no change of the flow's behavior is observed, the switch deletes the flow. Because the misbehaving flows sent much too fast, their packets were put at the end of the service queues; and because they did not listen to the control, they were quickly deleted from the switches' flow tables. Further packets from these flows were treated as from unknown users and received the lowest priority in handling.[8]

The above results show that normal flows are well protected from the few misbehaving users, no one lost a single packet. Also note that, even though the misbehaving users have driven the link utilization up to 100%, 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 the switches deleted the misbehaving flows, making normal flows see a lower utilization.

[8] In the Flow Network design, packets from unknown users are served with a priority lower than any of the flows with resource reservation, allowing random datagrams to be sent with no reservation and the network resources to be fully utilized by performance-insensitive background traffic.

3.2.4 Effects of VirtualClock on Queueing Delay

The major role of VirtualClock is to meter the average volume of a statistical data flow and to build firewalls among flows in statistical multiplexing. It should be made clear that VirtualClock does not contribute directly to queueing delay reduction. Rather, it helps indirectly through interleaving packets from different flows and assuring individual flows their reserved throughput rates.

Queueing Delay of Different Data Generation Patterns

Although statistical multiplexing absorbs certain randomness and burstiness in individual flows' data transmission, highly bursty data arrivals can still significantly increase queueing delay. Because of the strict service ordering enforced by VirtualClock, however, one would expect that a higher burstiness in a flow's data generation will be reflected back mostly to an increase in its own end-to-end delay.

This is indeed the case, as evidenced by the results of a simulation run with three different data generation models, constant rate, Poisson arrival, and packet-train. The test condition is the same as that in Test-One except that the last four flows were removed to lower the link utilization[9] (the measured utilization in this test is 78%), and that for flows 1 to 48, the data generation model is changed to two constant-rate, two Poisson arrival, and two packet-train in a repeated pattern. Flows 49 to 56 repeat the pattern of one constant-rate, one Poisson, and two packet train. 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)		
	1-hop	2-hop	3-hop
Constant	1.76	4.50	5.57
Poisson	4.55	8.37	11.40
Train	6.49	10.47	15.25
	Delay Deviation		
	1-hop	2-hop	3-hop
Constant	1.58	3.40	4.09
Poisson	6.44	9.74	11.15
Train	9.50	12.17	15.49

Packet-train has the highest burstiness among the three data generation models. Consequently, the flows with the train data generation received the highest queueing delay and delay variation. It seems fair to let bursty data sources bear the result of their own behavior. In particular, notice that the flows with the Poisson arrival model have both a lower average delay and a smaller delay deviation than the flows with the packet-train model. Using the packet train model instead of the Poisson model in testing stretches the performance of the VirtualClock algorithm, demonstrating its enhanced robustness for a wider range of data generation patterns.

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

4 Related Work

4.1 Schedule-Based Approach in Data Flow Control

In [6], Mukherji proposed a schedule-based approach to data traffic control. In his 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 using a slot whose designated user has no packet to send at the moment. Each user has an auxiliary flow control window which 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 channel slots to individual users, VirtualClock orders packet transmission sequences. Hence it achieves the same functionalities of the Mukherji algorithm but with more flexibility in handling different channel bandwidths and different user throughput demands.

4.2 Fair-Queueing

Fair-queueing is a simple 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 [3] and Demers et al. [1] have done extensive analysis work on fair-queueing's performance.

VirtualClock can be considered as performing a fair-queueing function, where the fairness is defined to be assuring each user the requested throughput rate. In fact various queueing policies can all be implemented by a simple computation on the VirtualClock value. In

particular, the fair-queueing algorithm by Demers et al [1] and the VirtualClock algorithm share a number of features. A major difference between the two is that VirtualClock is based on the resource reservation. Therefore instead of an equal share to all users, VirtualClock can allocate any specific amount of resources to each user.

4.2.1 Leaky-Bucket

Leaky-Bucket has been suggested as a flow control algorithm to be used at the network interface for high-speed networks [8, 9]. Various versions of the algorithm have been proposed. A simple model described in [9] works in the following way: the switch puts packets from each data flow into a corresponding bucket which has a fixed size; the bucket opens periodically to drop a packet out 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 (the bucket size) can be saved. Arriving packets are transmitted immediately if the flow has credits, otherwise they are either dropped or transmitted with a low priority.

Packet-switching networks should tolerate variations in packet arrivals while controlling flows' average transmission rate. The first version of Leaky-Bucket reduces statistical multiplexing because packets are transmitted at a constant rate rather than whenever the channel is available. The second version may result in highly bursty transmissions if all buckets in the switch is served in an FIFO order. The VirtualClock algorithm avoids those drawbacks by merely ordering packet transmissions 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* while the latter a *transmission control*. Leaky-Bucket enforces control at the switch entrance and determines whether an incoming packet should be accepted. Once packets are in, Leaky-Bucket has no further control over the order of transmission and thus it cannot discriminate packets according to different delay requirements. VirtualClock can also be used for admission control (e.g. when a flow's VirtualClock is running too fast, further packets from that flow can be rejected). Its main merit, however, shows at transmission control. VirtualClock determines the transmission order of packets from all users. The action is applied right at the multiplexing point, controlling exactly which packet go next.

5 Summary and Future Research

In this paper we presented VirtualClock as a new traffic control algorithm for packet switching networks. Its fundamental merit comes from its imitation of a TDM system in a statistically shared packet network, hence it achieves the desired properties of both systems; it maintains the statistical multiplexing flexibility of packet switching while ensuring each flow its reserved average transmission rate at the same time.

All control algorithms imply certain constraints on users. In the VirtualClock algorithm, such constraints are expressed as a *user-behavior envelope*, which requires each flow source constrain itself from sending more than ($AR \times AI$) data over each average interval. Although the user-behavior envelope was introduced as a solution to our specific problem in using VirtualClock for flow measurement, we believe that it is a mandatory part of rate-based flow control in general. How to design application protocols that can adjust themselves to the constraints is an interesting area to be further explored.

We also plan to further explore the performance of the VirtualClock algorithm under highly bursty traffic. Some preliminary 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 any quantitative conclusions.

Design of, and experimentation with, the VirtualClock algorithm also brought up a number of other interesting issues for future study. First, it may be attractive to applications with stringent delay requirements to slightly over-book resources to trade for an improved transmission delay. More work is needed to investigate a quantitative relation between the percentage of resource overbooking and the delay reduction. Another interesting area is to provide a bounded transmission delay and guaranteed delivery to emulate services provided by a circuit-switching 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 data loss-free. 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 av-

erage interval (AI), to describe and control statistical data flows. AI sets a bound on sources' transmission variation and defines a checking point in network measurement. The possible range of the AI value is

$$\frac{1}{average\ rate} < AI < flow\ duration$$

Tuning the value to reach the left limit, a data flow would transmit at a constant rate and we end up with a TDM system. Tuning the value to the right limit, a data flow would be able to transmit in any arbitrary manner as in an uncontrolled datagram network. In other words, a data transmission network can move along a continuous spectrum, with a TDM system at one end and a datagram network at the other, by adjusting the AI value. AI provides us a tuning knob between the system constraints and the service flexibility.

Acknowledgment

This paper is based on my thesis work completed at 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 valuable comments.

A An Example Implementation of User-Behavior Envelope

In our simulation test, the flow sources implement the user-behavior envelope control by a moving-average algorithm:

- Each flow source keeps the transmission time of the last AIR packets in a circular ring, with a pointer, Ptr, to the oldest slot.
- When sending out a packet, P, the source checks whether the real time $\geq$ $(Ptr(time) + AI)$. If so, the next packet, P_{next}, can be sent after the minimum inter-packet gap; otherwise P_{next} must wait for at least an average inter-packet gap.
- P's transmission time is saved in the slot pointed by Ptr, and Ptr is moved to the next slot.

References

[1] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of Symposium on Communication Architectures and Protocols*. ACM SIGCOMM, September 1989.

[2] A. Desclous, *Contention Probabilities in Packet Switching Networks with Strung Input Processes*, Teletraffc Congress 1988

[3] Ellen L. Hahne. *Round Robin Scheduling for Fair Flow Control in Data Communication Networks*. PhD thesis, Massachusetts Institute of Technology, December 1986.

[4] R. Jain and S. Routhier. Packet Trains - Measurements and a New Model for Computer Network Traffic. *IEEE Journal on Selected Areas in Communications*, SAC-4(6):986–995, September 1986.

[5] 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.

[6] Utpal Mukherji. *A Schedule-Based Approach for Flow-Control in Data Communication Networks*. PhD thesis, Massachusetts Institute of Technology, February 1986.

[7] John Nagle. Congestion Control in TCP/IP Internetworks. *ACM Computer Communications Review*, 14(4), October 1984.

[8] E. P. Rathgeb. Comparison of Policing Mechanisms for ATM Networks. Submitted to IEEE INFOCOM'90, June 1990.

[9] Jonathan S. Turner. New Directions in Communications (or Which Way to the Information Age?). *IEEE Communications Magazine*, 24(10):8–15, October 1986.

[10] Lixia Zhang. *A New Architecture for Packet Switching Network Protocols*. PhD thesis, Massachusetts Institute of Technology, July 1989.