6-5-2003

Discrete-Event Fluid Modeling of Background TCP Traffic

David M. Nicol
Dartmouth College

Guanhua Yan
Dartmouth College

Dartmouth College Department of Computer Science
Technical Report TR2003-454

Discrete Event Fluid Modeling of Background TCP Traffic*

David M. Nicol
Guanhua Yan
Department of Computer Science
Dartmouth College
Hanover, NH 03755

June 5, 2003

Abstract

TCP is the most widely used transport layer protocol used in the internet today. A TCP session adapts the demands it places on the network to observations of bandwidth availability on the network. Because TCP is adaptive, any model of its behavior that aspires to be accurate must be influenced by other network traffic. This point is especially important in the context of using simulation to evaluate some new network algorithm of interest (e.g. reliable mulitcast) in an environment where the background traffic affects—and is affected by—its behavior. We need to generate background traffic efficiently in a way that captures the salient features of TCP, while the reference and background traffic representations interact with each other. This paper describes a fluid model of TCP and a switching model that has flows represented by fluids interacting with packet-oriented flows. We describe conditions under which a fluid model produces exactly the same behavior as a packet-oriented model, and we quantify the performance advantages of the approach both analytically and empirically. We observe that very significant speedups may be attained while keeping high accuracy.

1 Introduction

It is impossible to overestimate the importance of the TCP protocol in shaping internet traffic. Simulation based evaluation of new internet applications or protocols must interact with "background traffic" that has TCP characteristics; ideally the application traffic both affects, and is affected by the background traffic. Consequently it is necessary to have a simulation model of background traffic that is efficiently executed, captures the salient features of TCP, and interacts with specific flows of interest even if those flows are packet oriented.

*A preliminary version of this work appears in the Proceedings of the 2001 Winter Simulation Conference, with the title *Discrete Event Fluid Modeling of TCP*.

Mathematical descriptions of traffic offer some hope for efficient generation of background traffic. The intuition is that abstraction aggregates behavior in a way that smooths over unessential details, allowing one to express the behavior using less computational effort. Analysis of fluid models in Markovian contexts was pioneered by Mitra, a recent application of which is found in [7]. Typically the focus in this type of work is on the behavior of a network component, such as a buffer. Another interesting approach to accelerating stochastic network simulations is to use importance sampling, e.g., [14, 6]; however, this is unlikely to serve our goals of efficiently generating representative background traffic as its focus is on fast estimation of statistics (e.g., probability of packet loss).

Direct efforts to reduce the computational cost of simulating a network include simulating "packet trains" rather than individual packets [1], simulating fluid models using time-stepping [15], and simulating fluid models using discrete events [5, 12]. The work reported in this paper uses essentially the traffic model employed in these last two papers, where a traffic flow is described by a piece-wise constant rate function.

This paper focuses on TCP and its simulation using a fluid model. Formulations of TCP that use differential equations are inherently "fluid-based" in that these express behavior in terms of rate functions. Sophisticated models have typically been used to analytically evaluate how TCP behaves. An oft-cited paper [13] shows how TCP throughput is related to the packet loss probability; models using stochastic differential equations (SDE) are used in [11, 2] to describe TCP behavior as a function of stochastic loss event models. The SDE approach treats the network as a generator of loss events and assumes some stochastic structure for the packet loss event process. Solutions to these equations are expectations (although these may be *time-dependent* expectations) with respect to the stochastic loss event process. There does not appear to be a direct coupling between the behavior of TCP modeled by an SDE, and its influence on the abstracted loss event process. Solutions of SDEs typically require the use of sophisticated numerical algorithms, although these can be found in standard mathematical packages.

An effort with a goal similar to ours is reported in [16]. There a packet-oriented simulation model is integrated with a fluid model (based on the SDE described in [11]), principally to show how to make packet and fluid models interoperate. The fluid model is used to estimate the total queueing delay of a packet as it traverses a region modeled by fluid approach. While packet flow information is used to provide the input to the fluid model for a time-step, there is no other direct interaction between packet and fluid within the fluid network, nor between packet streams that simultaneously cross the fluid network. In the experiments reported, the granularity of time-step for the fluid network solution needs to be at least one second for the hybrid approach to work as fast as an ordinary packet approach. Speedups of approximately 5 are observed for time-steps larger than 5 seconds.

The Time-stepped Hybrid Simulation (TSHS) approach[4] discretizes time into equal size intervals. All packet arrivals within a time-step are "chunked" together within the interval, no arrival time information is saved. "Chunk" versions of routers and TCP are developed. Speedups approaching 3 are observed (as compared to a packet-oriented **ns** solution) for sufficiently large time-steps.

Significant speedups over packet-oriented simulation have been achieved using an SDE approach, with a mathematical formulation that groups flows into "classes", where every flow in a class takes exactly the same route through the network[10]. The key to speedup in this approach is the class-based aggregation and the ability to compactly represent an entire class with a simple equation.

We are interested in a different corner of the modeling space. The model we propose and study needs no explicitly stochastic components; it simulates a particular sample path of a TCP session. There are a number of notable facets to the approach we describe:

- our model is closed-loop—it affects and is affected by other flows;
- we introduce a smoothing technique that provably defeats the well-known "ripple effect" of event explosions associated with fluid models;
- we are able to analyze the reduction in workload offered by the method (over a purely packet-based approach)

as a function of (i) the length of a TCP transfer, (ii) the rate at which an application offers data to be transfered, (iii) the round-trip-time, and (iv) the initial value of **ssthresh**.

- we intermingle packet-based representations of flows with fluid-based simulations of other flows;
- we implement a seamless mixture of fluid and packet representations in the same network simulation package. This model formulation is capable of working with dynamic routing and other realistic artifacts of network simulation and analysis.

We described some elements of our approach in a preliminary report [3].

One unique aspect of our work is that we prove mathematically that under certain conditions the method has *exactly* the same behavior as a packet-oriented simulation. We emphasize though that our interest is in developing lightweight but dynamic description of background traffic that behaves like TCP. Rather than ask (like prior work) how the interior elements of a network behave within the mixed model, we ask whether a reference packet flow behaves the same way when mixing with other packet flows as it does when mixing with fluid representation of those flows. We find the correspondence to be very good, and find that the simulation requires significantly less computation and memory than does a fully packet-oriented simulation. The degree of speedup depends very much on traffic characteristics.

This paper is organized as follows. Section 2 describes how the dynamics of a flow may be represented in a discrete event framework, then Section 3 shows how that framework is applied to model TCP. Section 5 formally shows that in the absence of loss our techniques are exact; section 6 develops mathematical expressions for the reduction in events one may achieve using our formulation. Section 7 reports on a set of experiments that consider the accuracy and speedup of the technique, and section 8 provides the conclusions.

2 A Discrete Event Fluid Model

We model a given traffic flow using a piece-wise constant rate function. In this view, at any physical point in the network, at any point in simulation time, the flow's behavior is described by a constant rate, e.g. in bits per second. That rate may change; when it does, it remains constant for some additional (and potentially arbitrary) period of time before changing again. This formulation is ideal for discrete-event simulation, where events describe rate changes. The advantage of such an approach (as opposed to a time-stepped approach, such as is used in the solution of fluid models based on differential equations) is that computation is performed only when, and where, it is needed to advance the model state. Our choice of model emphasizes computational speed, and simplicity.

Throughout this paper we will denote a function f with generic argument z, that is $f(z)$ refers to the function rather than a specific function value. $f(s)$, $f(t)$ and so on will denote specific values at specific points in simulation time. A number of quantities of interest in our model are based on piece-wise constant rate functions. Typically, function $f(z)$ (e.g. *cwnd(z)*) is defined implicitly through changes in some function $\lambda_f(z) = (d/dz)f(z)$, and $\lambda_f(z)$ is a piece-wise constant function of z. This means that at any time t

$$f(t) = \int_0^t \lambda_f(s)\, ds.$$

In particular, if an event occurs at time a where point $f(a)$ is known or computed, and another event occurs at time b with $\lambda_f(s) = c$ for all $s \in [a, b]$, then $f(b)$ is trivially computed as $f(b) = f(a) + c \times (b - a)$. The TCP model is based on byte indices within a flow; in this context the units of $\lambda_f(s)$ are bytes per unit time, and $f(b)$ is the byte index of the flow, at the point of observation, at time b.

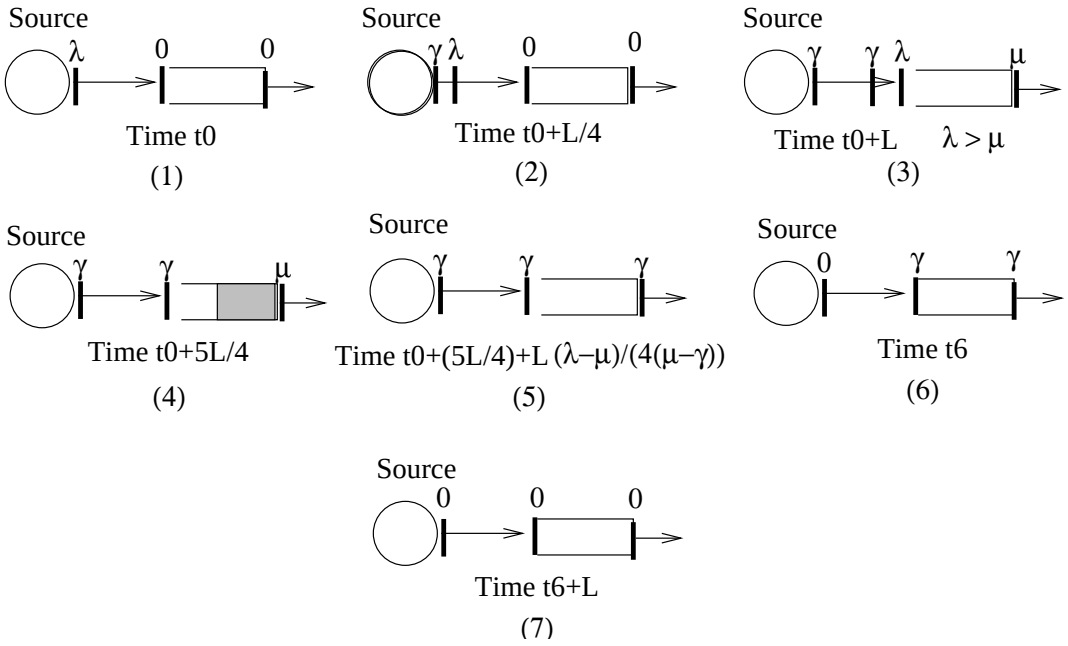

Figure 1: Example of Fluid Model

2.1 Example of Fluid Model

A simple example illustrates many important points about our modeling approach. Consider the sequence of steps shown in Figure 1. The model is of a traffic source which is connected to a buffered server (like a network interface card), with a latency L between them. Flow rates are shown at the output of the traffic source, the input to the server's buffer, and the output of the server. Event (1) of the sequence (at time $t0$) reflects a state where the source turns on, emitting traffic at rate λ. At $t0$ there is no flow entering or leaving the server. Event (2) illustrates the source changing the traffic rate from λ to γ only $L/4$ units of time later. This occurs before any of the λ-rate flow reaches the server—we illustrate this by positioning a flow descriptor on the connection between source and server. In the simulation this is implemented simply with an event on the event list marking the leading edge of the flow reaching the server. Event (3) depicts the state after processing that event. The source continues to emit traffic at rate γ; there is an input rate change scheduled for the server (depicted again as a flow descriptor on an arc), it is as-yet-unseen; the arrival rate λ exceeds the service rate μ, so that the output rate of the server is μ and we therefore expect traffic to build up in the buffer. As part of this event's processing we also schedule a "buffer full" event to occur in $B/(\lambda - \mu)$ units of time, where B is the capacity of the buffer. However, before this much time elapses, the input rate change from λ to γ arrives at the server; Event (4) illustrates the state of the system after processing this event. In the $L/4$ units of time since the server was first presented with a flow, it received $L\lambda/4$ units of traffic and pushed out $L\mu/4$ units of traffic. The size of the buffered backlog is thus $L(\lambda - \mu)/4$. With the arrival of input rate change $\gamma < \mu$, we see that in the absence of further rate changes, the buffer will decrease in size at rate $\mu - \gamma$, becoming empty after $L(\lambda - \mu)/(4(\mu - \gamma))$ units of time. At this point the server will have to change its output rate, so an event is scheduled to cause the server to do just that. Event (5) reflects the state of the system after executing that event. The system remains in this state until the source turns off the flow at time $t6$, illustrated in Event (6). Another L units of time pass before the input change reaches the server; Event (7) illustrates the system state after the server processes this rate change.

The important points of this example with respect to fluid modeling in general are

- *Latency can be modeled just as in packet oriented simulation.* Latency is just the length of time it takes a bit to

cross a communication channel, and so we can impose latency on a flow descriptor just as we would a packet.

- *Events are triggered by input rate changes, or timer firings.* When the input rate to a component changes, it may be necessary to change an output rate (e.g. Events (3) and (7)). It may also be necessary to analyze the state of the component, project its future behavior in time, and schedule a timer to trigger an event when the projected system state encounters some boundary, e.g. when the processing associated with Event (4) schedules Event (5).

- *The computational efficiency over packet oriented simulation depends entirely on the rates and the length of time between their changes.*

The example given takes the system through a certain span of time in 7 events. The number of events required to do this in a packet-oriented version of this model depends entirely on the number of packets transmitted between each of the 7 epochs described in the example. That might be small (if λ and γ are small), it might be large. It depends on how often the model reshapes the flows. In this paper we focus on how TCP reshapes such flows.

2.2 Additional Mechanisms

Our implementation of fluid TCP uses some additional mechanisms. One of these is motivated by the fact that TCP flows carry headers, which affect bandwidth consumption. The TCP logic is based on data bytes, while the network model considers total traffic. Furthermore, the TCP acknowledgement flow can be contained entirely in headers. We handle this by expressing flows at the network level in terms of the total bytes, and have the flow descriptor carry a "logical-to-physical" byte ratio (ρ). This allows us to express some logical flow (e.g. data bytes, or acknowledged bytes) in terms of a physical flow that implements it. The product of a flow's physical rate and its ρ gives the flow's logical flow rate. In the case of TCP, the ρ for a data-bearing flow is the ratio of the number of bytes in a data segment to the sum of header size and data segment size.

We will also have cause to embed discrete bits of information in a flow, and have that flow carry it along. We call these additions *corks*. If a cork is inserted into a flow at a point and time when byte position b is passing the point of insertion, then that cork appears downstream in the flow whenever and wherever byte b of that flow appears. We might use a cork to carry identity information (e.g., source or destination address), flag that a flow has terminated, or declare a new data segment size. We will use corks to mark where in a flow bytes begin to be lost. Corks are always attached to flow descriptors; if addition of the cork does not change any other of the flow's characteristics, a bit is set to flag this condition.

We use another mechanism to report the existence and quantity of data loss in a flow. We associate a "delivered fraction attribute" (τ) with a flow; it will be carried by a cork. A flow's τ value indicates what fraction of the physical flow at point of origin is actually passing in the present flow; the τ value at point of origin is 1.0. If the network model introduces loss into a flow, it modifies the physical flow rate and decreases the flow's τ value by multiplying the existing τ by the fraction of real flow observed at that point which continues to be delivered. It attaches to the flow descriptor a cork that gives the new flow τ value. So for example if a flow is reduced by 20% at the first router it encounters, a τ value of 0.8 is associated with it, and if an additional 10% loss is introduced by a subsequent router, then τ is changed to $0.8 \times 0.9 = 0.72$ to reflect the fact that the flow being delivered is 72% of the flow that was transmitted. An element of the network can therefore always infer what originally transmitted byte index corresponds to a flow passing it, at any instant in time. If the flow descriptor has changed at times $s_1, s_2, \ldots, s_n$ with rate λ_i and delivered flow attribute τ_i at time s_i, the byte index corresponding to the flow passing it at time $t > s_n$ is

$$\sum_{i=0}^{n} (s_{i+1} - s_i)\lambda_i/\tau_i$$

where for notational convenience we define $\lambda_0 = 0$ and $s_{n+1} = t$, and assume that $\tau_i = 1$ whenever $\lambda_i = 0$. If we allow for the possibility that one flow descriptor may contain a different ρ than another, the *logical* byte index of the flow passing the point of observation is

$$\sum_{i=0}^{n}(s_{i+1} - s_i)\lambda_i\rho_i/\tau_i,$$

where ρ_i is the logical-to-physical value associated with the i^{th} flow descriptor.

Taking all of these mechanisms into consideration, a flow in our formulation has the following attributes :

- physical byte rate (in bytes per unit simulation time),
- ratio of logical to physical bytes (ρ),
- delivered fraction ratio (τ),

A flow description may contain these, along with a list of corks (possibly empty).

3 Fluid Modeling of TCP

We view a TCP session in terms of a sending agent and a receiving agent, both in protocol stacks at their respective hosts. TCP is fully duplex by specification, and both functions can be simultaneously functioning. For our purposes it suffices to describe the sender and receiver roles separately, understanding that they can be merged into a single entity. Throughout our discussion we assume that the TCP sender always sends packets of the maximum segment size (MSS in TCP parlance). We also assume that the TCP receiver always has sufficient memory to buffer any transmissions from the TCP sender, and so do not consider the receiver window size as a constraint in this model. Unless specifically indicated, all rates described here are logical (data byte) rates, not physical network rates.

TCP views a transfer in terms of a stream of bytes, indexed from 0, from a sender to a receiver. The receiver agent provides the data to the protocol layer above it, in byte sequence order, and without loss. To support this functionality, TCP puts byte indexing information in headers of the packets it sends, and requires acknowledgements be sent for received packets (so that the sender can discard the packet, once it knows that the packet need not be retransmitted). The TCP protocol imposes flow control rules, in order to avoid sending packets faster than the network can accept and move them. These rules are expressed in terms of a few key state variables, listed below.

Variable	Meaning
LBS	Last Byte Sent
LBA	Last Byte Acked
$cwnd$	Congestion window size
$ssthresh$	Mode transition threshold

A sending agent stores in LBS the index of the last byte in the last packet it sent. It stores in LBA the index of the last byte whose receipt has been acknowledged by the TCP receiving agent. The most basic TCP rule is that "the next" packet may not be sent if the number of unacknowledged bytes exceeds threshold $cwnd$, e.g., if $LBS - LBA > cwnd$. Threshold $cwnd$ is not static. TCP rules govern how $cwnd$ changes in response to received acknowledgements, and indications of packet loss. In *slow start* mode TCP increases $cwnd$ by a packet length every time a packet is acknowledged. The effect is that TCP sends out packets in rounds, with the number of packets doubling in each round, the next round being triggered by the receipt of acknowledgments from the previous round. Eventually $cwnd$ reaches threshold $ssthresh$, or TCP detects a packet loss, in which case it enters congestion avoidance mode. In this mode TCP increases $cwnd$ more slowly. When TCP sets $cwnd$ to allow K unacknowledged packets to be sent, it requires K

Variable	Description
$LBS(t)$	value of LBS at simulation time t
$LBA(t)$	value of LBA at simulation time t
$cwnd(t)$	value of $cwnd$ at simulation time t
$\lambda_{app}(t)$	maximum data rate from Application at time t
$\lambda_{ack}(t)$	acked byte rate from Application at time t
$\lambda_{send}(t)$	data rate sent from TCP at time t
$\lambda_{bw}(t)$	maximum bandwidth (in data bytes) available to TCP sender
$\lambda_{cwnd}(t)$	rate at which $cwnd$ is changing at time t

Table 1: Functions used in fluid TCP sender model

packets to be acknowledged before it increases $cwnd$, to allow $K + 1$ unacknowledged packets. Detection of packet loss in congestion avoidance mode causes $ssthresh$ to be halved, and causes a transition back into slow start mode.

In our fluid formulation the key TCP variables become functions of simulation time, as given in Table 1. All of the λ functions are piece-wise constant functions of simulation time t. Note that $(d/dz)LBS(z) = \lambda_{send}(z)$ and $(d/dz)LBA(z) = \lambda_{ack}(z)$, which implies that $LBS(z)$ and $LBA(z)$ are piece-wise linear functions of z. $\lambda_{cwnd}(z)$ will be a function of the congestion mode. The big picture is that the TCP sending agent computes flow descriptors defining $\lambda_{send}(z)$ and pushes these into the network; the specifics of this description are a function of input rate functions $\lambda_{app}(z)$ and $\lambda_{ack}(z)$, and available bandwidth function $\lambda_{bw}(z)$. Of course, $\lambda_{ack}(z)$ reflects the behavior of $\lambda_{send}(z)$ in the past, and the influence of the network on both the sending stream and the acknowledgement stream.

Figure 2 illustrates our view. It contains representations for the application, the TCP sender, the Network Interface, and devices we introduce called Input Delay Elements (IDE). We will say more about IDE's later; they shift rate changes in simulation time to account for the time needed to accumulate a full packet of data. As we will see, they are needed to keep a fluid representation of a flow in temporal synchronization with an equivalent packet representation of the flow.

The application maintains a rate $\lambda_{app}(z)$ of the maximum rate it can provide data to the TCP sender, independent of how fast the TCP sender actually takes it. There is a feedback loop reporting rate $\lambda_{send}(z)$ back to the application, telling it how fast its data is actually being accepted. The Network Interface environment can in principle dynamically alter bandwidth available to the session through changes in function $\lambda_{bw}(z)$, e.g. if multiple sessions share bandwidth out of a single network interface card.

A change in $\lambda_{app}(z)$, $\lambda_{ack}(z)$, or $\lambda_{bw}(z)$ at time t may cause a change in $\lambda_{send}(z)$ at time t. A change in this output can also be triggered by the firing of an internally scheduled timer at t. From this description it is apparent that the sender agent can in principle be described as a finite state machine, and we will shortly do exactly that. The model is better understood though if we first work through details of how TCP behavior may be captured using fluid description.

3.1 Some Simplifications

The methodology we will develop is capable of modeling fairly sophisticated behavior of TCP such as fast retransmit, timeouts, and retransmission of lost data. However, attempts to faithfully represent TCP behavior with respect to data retransmission significantly complicate the model. Our goal for this model is to simply capture how network latency and packet loss affects the rate at which a TCP sender transmits. The model will respond to *reports* of data loss from the network, and move between slow start and congestion avoidance modes appropriately. It will retransmit lost data. However, a TCP sender will not *detect* data loss either by analysis of acknowledgements, nor expiration of time-out

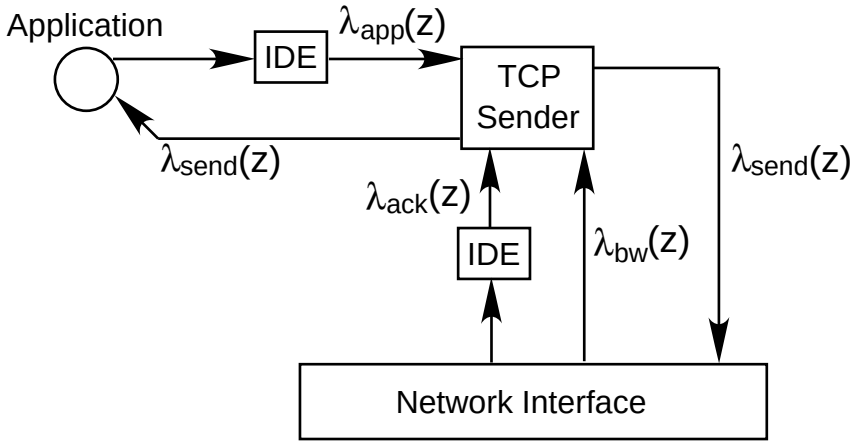

Figure 2: View of TCP Fluid Model

timers.

3.2 Modeling sender window dynamics

Management of the sender window size is the soul of TCP, and is the centerpiece of our model. We essentially fluidize the discrete description of TCP's rules. In our model $LBA(z)$, $LBS(z)$ and $cwnd(z)$ are all piece-wise linear functions of time. $LBA(z)$'s behavior is directly defined by $\lambda_{ack}(z)$'s as follows:

$$LBA(t) = \int_0^t \lambda_{ack}(s)\, ds.$$

$LBS(t)$ is similarly defined. The fluid characterization of *cwnd* depends on the congestion mode. When TCP is in slow-start mode, the receipt of one packet's acknowledgement increases $cwnd$ by one packet's worth of bytes. However in the fluid model, acknowledgement bytes arrive in a stream; the logical continuous extension is that every bit acknowledged increases $cwnd$ by one bit. The *rate* at which $cwnd(t)$ increases is precisely the rate at which acknowledgements arrive. Thus

$$\lambda_{cwnd}(z) = \frac{d\, cwnd(z)}{dz} = \lambda_{ack}(z).$$

In congestion avoidance mode, real TCP requires K packets to be acknowledged before increasing $cwnd$ from $K \times MSS$ to $(K+1) \times MSS$, where MSS is the maximum segment size, e.g. the packet length. We can emulate that behavior exactly in the fluidized context. Suppose that on entering congestion avoidance mode at time s, $cwnd(s) = K * MSS$. If we assume that $\lambda_{ack}(z)$ remains constant, we can initialize a variable $w_{ack} = K \times MSS$ to store the amount of acknowledgement fluid needed for $cwnd$ to change, and schedule an event to increase $cwnd(t)$ to $(K+1) * MSS$ at time $s + \lambda_{ack}(s) \times w_{ack}$. If $\lambda_{ack}(z)$ changes at time t before this event occurs, part of processing the acknowledgement rate change will be to subtract $(t-s)\lambda_{ack}(s)$ from w_{ack}, and reschedule the $cwnd$ change event to occur after $w_{ack} \times \lambda_{ack}(t)$ units of time. Since $cwnd$ changes discontinuously, in congestion avoidance mode we define $\lambda_{cwnd}(t) = 0$.

3.3 Timers

The example we worked through in Figure 1 illustrates how a fluid model's state sets off on some trajectory (e.g. towards the buffer becoming full, or towards the buffer becoming empty), with a timer scheduled to execute an event when the fluid state encounters a boundary that triggers a change in the fluid model's behavior. The same is true modeling the TCP sender. There are critical transition points where the fluid state of the sender intersects a boundary, which forces a change in the output $\lambda_{send}(z)$. Understanding those boundaries and the means of intersecting them is the key to understanding our formulation of fluid TCP.

The central constraint of TCP is that $LBS(s) - LBA(s) \leq cwnd(s)$ at all times s (except for states brought on by data loss). When $LBS(s) - LBA(s) < cwnd(s)$, the sender is free to transmit data as fast as possible. When $LBS(s) - LBA(s) = cwnd(s)$ the sender is constrained to send no faster than $cwnd(s)$ grows. When $LBS(s) - LBA(s) > cwnd(s)$ the sender is constrained from sending at all. Thus we see that if $LBS(s) - LBA(s) \neq cwnd(s)$, it may be necessary to have a timer running, to fire (and execute an associated event) precisely at the time t that $LBS(t) - LBA(t) = cwnd(t)$, provided of course that the sender's fluid state ($LBS(t) - LBA(t)$) is moving towards intersecting $cwnd(t)$. We will call this timer *ConstrainedTimer*. In particular, *ConstrainedTimer* must be running in states where at time s the inequality $LBS(s) - LBA(s) < cwnd(s)$ holds, and the window size is moving to intersect $cwnd(s)$, i.e., $\lambda_{send}(s) > \lambda_{ack}(s) + \lambda_{cwnd}(s)$. Likewise, *ConstrainedTimer* must be running in states where at time s the inequality $LBS(s) - LBA(s) > cwnd(s)$ holds, and $\lambda_{ack}(s) > 0$ (note that in such states $\lambda_{send}(s)$ must always be zero.)

Scheduling the firing time for *ConstrainedTimer* is straightforward. The fluid state $LBS(t) - LBA(t)$ has a linear trajectory in t, with slope $\lambda_{send}(s) - \lambda_{ack}(s)$. The fluid variable $cwnd(t)$ has a linear trajectory in t, with slope $\lambda_{cwnd}(s)$. As illustrated in Figure 3, we are interested in finding their point of intersection. That intersection occurs at time $s + d$, where

$$s + d = s + \frac{cwnd(s) - LBS(s) + LBA(s)}{\lambda_{send}(s) - \lambda_{ack}(s) - \lambda_{cwnd}(s)}. \tag{1}$$

In the case illustrated, the result of the timer firing is to adjust $\lambda_{send}(z)$ so that for $t \geq s + d$, $\lambda_{send}(t) - \lambda_{ack}(t) = \lambda_{cwnd}(t)$.

Other internal factors give rise to additional timers. In slow-start mode, if at time s we have $cwnd(s) > 0$, there must be a timer *ModeTransition* running, scheduled to fire at the instant t when $cwnd(t) = ssthresh$. That timer fires at time

$$s + d = s + \frac{ssthresh - cwnd(s)}{\lambda_{cwnd}(s)}. \tag{2}$$

In congestion avoidance mode we use a timer *IncreaseCWND* to fire when enough acknowledgements have arrived to cause an increase in $cwnd(t)$. If we denote the value of variable w_{ack} at time s by $w_{ack}(s)$, then provided that $\lambda_{ack}(s) > 0$, $cwnd(s)$ logically increases by MSS at time

$$s + d = s + \frac{w_{ack}(s)}{\lambda_{ack}(s)}. \tag{3}$$

While a straightforward implementation will make sure that *IncreaseCWND* is running for all times s such that $\lambda_{ack}(s) > 0$, there is an important optimization we can employ. If an increase in $cwnd$ at time t cannot change $\lambda_{send}(t)$ from what it would have been without the increase, then there is actually no point in having *IncreaseCWND* running, so long as we can always compute what $cwnd$ ought to be, when needed. Now if $LBS(t) - LBA(t) < cwnd(t)$, an increase in $cwnd$ at time t cannot affect $\lambda_{send}(t)$; thus in congestion avoidance mode, *IncreaseCWND* is running only at times t where $LBS(t) - LBA(t) \geq cwnd(t)$.

If we employ this optimization, we can reconstruct $cwnd$ as needed. We remember the total volume of acknowledgements received by the instant when the sender last entered congestion avoidance mode, and remember the value of $cwnd$ at that instant, say, C. We know that once congestion avoidance mode is entered, exactly C bytes must be

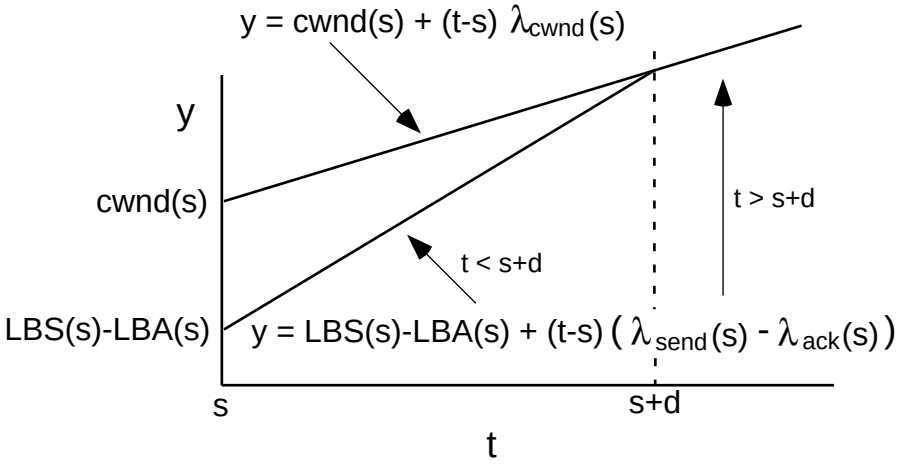

Figure 3: Point where $LBS(t) - LBA(t) = cwnd(t)$ found by intersection of two linear functions

acknowledged before $cwnd$ is increased by MSS, after which $C + MSS$ more bytes must be acknowledged before $cwnd$ is increased to $C + 2 \times MSS$, and so on. If Λ bytes have been acknowledged since the transition into congestion avoidance mode, we can compute the number of times n that $cwnd$ is increased. For we know that

$$\Lambda = \left(\sum_{k=0}^{n-1} (C + k \times MSS) \right) + \alpha,$$

where n is as large as possible and still have $\alpha \geq 0$. We can determine n by treating it as a real number x, expand the sum above, and solve for x in

$$\Lambda = x \times C + MSS \times \frac{x(x-1)}{2}.$$

This is a quadratic equation in x; we take n' as the integer fraction of its positive root then reconstruct

$$cwnd = C \frac{n'(n'-1)}{2}.$$

3.4 Defining $\lambda_{send}(z)$

Next we turn to consideration of $\lambda_{send}(z)$. The basic observation is that $\lambda_{send}(t)$ should always be "as large as possible" at time t. The sending rate is always constrained by both the maximum rate data that can be drawn out of the application, and the maximum rate at which data can be injected into the Network Interface portion of the model. Thus $\lambda_{send}(t) \leq \min\{\lambda_{app}(t), \lambda_{bw}(t)\}$ at all times t.

At times t where $LBS(t) - LBA(t) < cwnd(t)$, there are no further constraints on $\lambda_{send}(t)$. When $LBS(t) - LBA(t) > cwnd(t)$ the sender is prohibited entirely from sending. At times t where $LBS(t) - LBA(t) = cwnd(t)$ continuously on the right (i.e. there exists $\epsilon > 0$ such that $LBS(x) - LBA(x) = cwnd(x)$ for all $x \in [t, t + \epsilon)$), then $LBS(x) = LBA(x) + cwnd(x)$ continously on the right. The right hand side of this equality changes at rate

$\lambda_{ack}(x) + \lambda_{cwnd}(x)$, hence the left hand side does as well. We are led then to define

$$\lambda_{send}(s) = \begin{cases} \min\{\lambda_{bw}(s), \lambda_{app}(s)\} & \text{if } LBS(s) - LBA(s) < cwnd(s) \\ \min\{\lambda_{bw}(s), \lambda_{app}(s), \lambda_{ack}(s) + \lambda_{cwnd}(s)\} & \text{if } LBS(s) - LBA(s) = cwnd(s) \\ 0 & \text{if } LBS(s) - LBA(s) > cwnd(s) \end{cases} . \quad (4)$$

3.5 Data Loss

We will require that the network model insert a cork reporting a change in a flow's τ value, whenever and wherever that τ value changes. That cork will always eventually make it back to the flow's originating sender. Because corks are carried along in a flow without losing byte position, the recipient of a cork can calculate that byte position from knowlege of a previous byte position and measurement of flow and time that has past since that position.

When a TCP sender receives a cork reporting $\tau < 1.0$ on a flow whose τ value had been 1.0, we arrange that the cork be repositioned at the *beginning* of the segment in which the loss occurred. In Figure 2, the IDE between the Network Interface and the TCP Sender implements this repositioning. Recognizing a new loss, the sender modifies $cwnd$ and $ssthresh$ in accordance with TCP rules for data loss. We assume that under these transformations $cwnd$ and $ssthresh$ are both integral multiples of MSS. This will become important when we analyze the accuracy of the methods. The sender suspends further output until all transmitted bytes have been acknowledged; employing a timer *LossCleared* to fire when this condition is achieved. If the loss is first detected at time t and if $\lambda_{ack}(t) > 0$, then *LossCleared* is scheduled to fire at time $t + (LBS(t) - LBA(t))/\lambda_{ack}(t)$. If $\lambda_{ack}(z)$ changes before the timer fires, *LossCleared* is rescheduled appropriately.

When *LossCleared* fires *ssthresh* is set equal to one-half of *cwnd*, *cwnd* is set to reflect MSS bytes, and the mode is set to slow-start. While not modeled on TCP behavior per se, it is a simple and fast technique for clearing an error condition.

We do not model timeouts. The flow carries a loss indication back to the source as quickly as it would be carried by an implementation that uses fast retransmit. The presumption is that an implementation which supports fast retransmission rarely identifies loss by timeouts.

3.6 State Space

We can define a state vector for a TCP sender such that given the state, it is possible to determine which timers are scheduled, when they are scheduled to fire, and what value $\lambda_{send}(t)$ has in that state.

One component of the state vector describes the relationship of $LBS(z) - LBA(z)$ to $cwnd(z)$. We define state component S_w to be 'u' (unconstrained), 'c' (constrained), or 'e' (exceeded) depending on whether $LBS(t) - LBA(t)$ is less than, equal to, or greater than $cwnd(t)$ at time t, respectively. We define another component, named S_m (m for mode) to have value 's' when in slow-start mode, 'a' when in congestion avoidence mode, and 's' when suspended waiting for a loss condition to clear. Still another component reflects whether the send window size is growing, shrinking, or neither with respect to the congestion window. This component is named S_c (c for change), which is defined to be '+', '-', or '0' depending on the sign of the difference $\lambda_{send}(s) - \lambda_{ack}(s) - \lambda_{cwnd}(s)$, with '0' reflecting equality. A final component is named S_a (ack indicator) which has value 1 if $\lambda_{ack}(s) > 0$ at time s, and 0 otherwise. We thus describe the state of a TCP sender with a four-component vector (S_w, S_m, S_c, S_a). There are 54 unique state vector values.

A few simple invariants dictate which timers are running, in which states. *ConstrainedTimer* is running in every state where $S_w = u$ and $S_c = +$, or $S_w = e$ and $S_c = -$. This is simply a formal statement that the send window size isn't identically the congestion window size, and that the difference between the send window and congestion window sizes is changing to move the send window size towards the congestion window size. *ModeTransition* is running in

every state where $S_m = s$ and $S_a = 1$, which says that in slow-start mode $cwnd(z)$ is increasing towards *ssthresh*. *LossCleared* is running in every state where $S_m = s$ and $I_a = 1$. Finally, *IncreaseCWND* is running in every state where $S_m = c$, $I_a = 1$, and $S_w = c$. This means that in congestion avoidance mode the sender window size is constrained from growing by $cwnd$, but that acknowledgements are actively accumulating to grow w_{ack}, and hence that $cwnd(z)$ will eventually change and that when it does, $\lambda_{send}(z)$ may change.

The output and state transitions of the TCP sender finite-state machine are completely determined by these invariants and Equation 4.

3.7 Input Delay Element

In a packet-oriented simulation, a packet is not considered to have "arrived" until it is completely received. So for example, a 1K packet sent across a 1ms latency 10Mb link requires 1ms for the first bit to be received, then 0.1ms for the rest of the packet to show up. Consider a fluid flow that passes through a router which is otherwise idle. In previous models of fluid flow, *the instant* the input rate changes at the router, the flow's output rate changes as well. If the flow had been completely idle up until that instant, then this is equivalent to having the first bit of the first packet be immediately routed through. To achieve an exact correspondence between fluid and model packets we must model the packet arrival delay, within the fluid model; we do this with a logical construct we call an *input delay element*, or IDE.

An IDE is configured to delay a rate change by the time needed to receive a given byte volume V, in the packet model. If the bandwidth of the channel into the IDE is λ_{bw}, that delay is $d = V/\lambda_{bw}$. If $\lambda_{in}^{ide}(z)$ is input rate function to the IDE, we define the output function of the IDE by

$$\lambda_{out}^{ide}(s) = \lambda_{in}^{ide}(s - d).$$

Thus an input delay element defines a temporal shift in the input flow description. When λ_{bw} and V are fixed and the IDE has no other function, the simplest implementation of the IDE is to add d to the latency imposed on a rate change. Otherwise it is straightforward to implement an IDE using a queue of received flow descriptors, and a timer whose firing releases the one at the front of the queue.

Our model uses IDEs at the input of every component that receives flow descriptors from some sort of channel. In Figure 2 the IDE between the Application and TCP Sender assumes a memory channel bandwidth, and delays by the time needed to deliver a data segment. IDEs sit in front of a router or switch, and are configured to delay by the transmission time of a full IP packet. The IDE between the Network Interface and TCP Receiver likewise delays by the channel transmission time of a full IP packet, but the IDE between Network Interface and TCP Sender delays by the transmission time of an IP header only. This one is specialized, to align loss data corks with the beginning of the header.

3.8 Modeling TCP receiver behavior

The only thing a TCP receiver does in our model is to acknowledge data bytes. There are however some subleties to our approach. One is that we have the receiver acknowledge all *intended* data bytes, not just delivered ones. As we have noted earlier, a received data byte rate is transformed into an intended data byte rate by dividing through by the flow's τ value. A continuous acknowledgement of intended bytes actually reflects one aspect of TCP behavior, in as much as it acknowledges every segment received, whether in order or not. In actual TCP the sender can infer from acknowledgement information when there are gaps in the received flow, and hence that a received acknowledgement is in response to a segment that follows loss. In our model acknowledging all intended flow accomplishes the same thing.

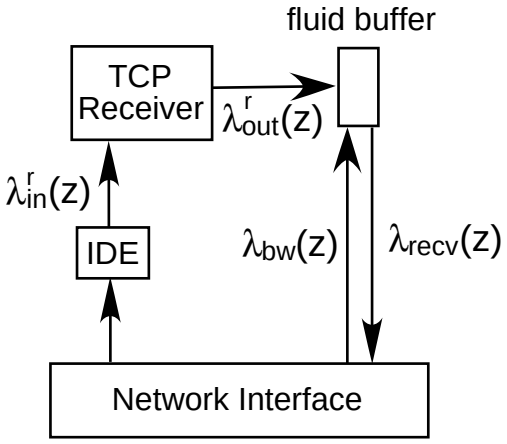

Figure 4: Fluid TCP Receiver Organization

Our discussion of the TCP receiver assumes that the raw incoming byte flow is transformed by the network interface into a data byte flow, by scaling the arrival rate by the flow's ρ value (logical-to-physical ratio). A receiver essentially acknowledges bytes, rather than segments. Since an actual TCP receiver does not acknowledge a segment until that segment is completely received, we likewise require that the receiver begin to acknowledge a segment only after all of the intended bytes for that segment have been detected. An IDE that delays the inflow by an intended data segment transmission time gives us precisely what we need. We denote the time-shifted rate out of this IDE by $\lambda^r_{in}(z)$.

Assuming an asymmetric TCP session, the acknowledgement flow requires less bandwidth than the received flow. The receiver needs to create a flow whose logical rate carries the received (intended) data rate. If the TCP header (which carries the acknowledgement) has α bytes and a TCP data segment has δ bytes, then the receiver's *offered output rate* at time s is

$$\lambda^r_{out}(s) = \lambda^r_{in}(s) \times \frac{\alpha}{\delta}.$$

This expression scales down the intended data byte flow rate to reflect that fewer bytes are needed to acknowledge a segment than to send one. A flow descriptor received from the IDE simply has its rate changed in accordance with this transformation; this resulting flow has $\tau = 1$ and $\rho = \delta/\alpha$.

A cork on the incoming flow is normally passed back, attached to the same descriptor (with rates now transformed, as above). However, a cork reporting data loss is first marked as having been seen by the receiver. This marking allows the TCP sender to distinguish between corks reporting data losses on the send path (to which it responds) from the data losses on the acknowledgement path (which it ignores).

The final step is to transform the receiver's offered output flow into an actual flow. The transformation is needed to deal with the possibility that $\lambda^r_{out}(z)$ is larger than the bandwidth available to the receiver from the network interface. To regulate the output flow rate we direct the offered output flow into a fluid buffer like the one described in Figure 1. The buffer has service rate $\mu_{bw}(z)$, a function regulated by the network interface to reflect the maximum bandwidth available to the receiver. The buffer has no limits on its capacity. The normal state will be for the buffer to have no backlog, and $\lambda^r_{out}(z) < \mu_{bw}(z)$. The final flow rate function from the fluid buffer is $\lambda^r_{ack}(z)$.

Because the TCP receiver is so tightly coupled with its IDE and fluid buffer, an implementation can easily merge them all. In normal circumstances the fluid buffer is empty and a flow descriptor released at time s translates immediately into a new flow descriptor on the acknowledgement flow. The code implementing the firing of the IDE timer can transform the flow rates and deal with the fluid buffer.

Figure 4 depicts this organization of a TCP receiver.

3.9 Ack Processing

We now discuss how a TCP sender processes the returning ack flow. The raw acknowledgement stream delivered by the network to the network interface (see Figure 2) reflects network effects on the flow emitted from the receiver with rate function $\lambda_{ack}^{rcv}(z)$; the transformed stream's rate function is $\lambda_{ack}^{in}(z)$. Just as an IDE delays the input stream to the TCP receiver by an intended packet arrival time, the acknowledgement stream to the TCP sender is delayed by an intended *header's* arrival time. The delayed header stream rate is multiplied by ρ/τ to produce $\lambda_{ack}(z)$, a flow whose units are acknowledged intended bytes per unit time.

The network interface consumes corks that report loss on the acknowledgement flow, but passes along to the TCP sender the corks marked as originating on the sender-to-receiver flow. The TCP sender can then detect when a flow it transmitted begins to lose fluid, and as described earlier, suspend further transmission of that flow until all of its outstanding bytes are acknowledged.

4 Fluid Routers

Our prime motivation for developing a fluid version of TCP is to allow efficient representation of background TCP traffic on specific packet oriented flows. Different flow representations interact in routers, and we turn next to issues of modeling flow interactions in a router.

4.1 Reduction of Ripple Effect

The interaction of fluid flows within a router has already received considerable attention [5, 12, 8, 9]. The classical approach uses multi-input-flow fluid buffers at each output port, and models provisioning of FCFS service. When there is no fluid backlog and the aggregate arrival rate (over all flows) to the buffer is less than the output service rate, then each flow's output rate is identical to its input rate. Things are more complex when there is backlog. This is illustrated by Figure 5. In the first diagram we see the input rate (12) for a single flow exceeds the buffer's output rate (10), and so a backlog has accumulated (illustrated with the black volume at the head of the queue). The next diagram reflects a time after a positive arrival rate begins on a second flow. The backlog is a mixture of backlog built up before the second flow began, and after. In this case the first black volume has diminished in size, and a mixture of black and white volumes are accumulating behind it in the queue. According to FCFS modeling, the first backlog is served before any of the backlog of the second flow is served—even through the second flow has input rate 8, its output rate is 0. The third diagram shows the situation after the priority backlog is completely served; the two flows share the limited output bandwidth in proportion to their relative proportions in the backlog being served. There are two different proportions of flow mixtures shown; the first mixture must be entirely worked off before any of the volume that accumulated after the second flow's rate change (to 24) occurred.

A well-known "ripple effect" is associated with this formulation. The effect of only one input rate changing while the buffer has backlog is to ultimately change *every* output rate, because the proportions of fluid arrivals are all altered by the input rate change. Rather than reduce the number of events needed to model a flow, the ripple effect can multiply the number of events.

Our model formulation offers one way to mitigate the ripple effects. It is often the case that a non-zero latency is associated with a channel out of a router. As we observed in Figure 1, an output rate change does not affect the receiving component until that latency time has expired. When we use IDEs there there an additional period of insensitivity, to model the packet transmission. Within this aggregate period of insensitivity we "smooth" already emitted output rates, before they affect the recipient. We can do so in a way that does not affect the volume of fluid that is delivered. As before, let L be the latency; let L' be the latency plus the delay imposed by the receiver's IDE. We buffer rate change events between the time they are generated at the output, and delivered to the receiver. When a

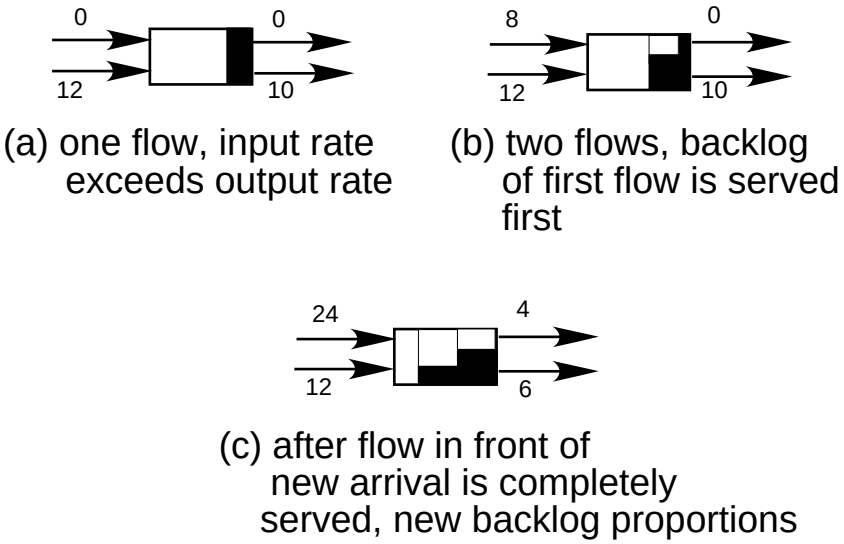

Figure 5: FCFS Modeling of Fluid Buffer

rate change event is produced at the output the buffer state is checked. If empty, a timer is set to fire after L' units of time, and the event is queued. If non-empty the rate change is simply queued. When the timer fires there will be some number of changes queued up, say, at times t_i with rates λ_i, for $i = 0, 1, \ldots, k$, with $t_0 \leq t_1 \leq \ldots t_k$. The timer fires at time t_0. We can compute the total volume of fluid to be delivered between times t_0 and t_k, change λ_0 to deliver that volume over the period $[t_0, t_k]$, and eliminate all but two of the rate change events. That is, define

$$\mathcal{V} = \sum_{i=0}^{k-1} \Big(\lambda_i (t_{i+1} - t_i) \Big),$$

and *redefine* $\lambda_0 = \mathcal{V}/(t_k - t_0)$. The new rate can be delivered to its recipient, and the buffer emptied of all change events except the last one, at t_k. The timer is scheduled to fire at time $t_k + L'$.

Using this technique we can guarantee that no more than 2 output rate change events ultimately occur per flow on an output channel in any period of L' units of simulation time, where L' is the sum of latency and unit transmission time.

Theorem 1 *Consider any flow out of a router, along a channel with insensitivity L', and suppose smoothing is applied as described. Then in any period of L' units of simulation time, at most 2 output rate changes are delivered.*

Proof : Suppose not. Let times t_0, t_1, and t_2 be times at which output rate changes are delivered to the recipient, $t_0 \leq t_1 \leq t_2$, and $t_2 - t_0 < L'$. Observe that the smoothing operation does not alter the times at which rate changes occur. Therefore the change at time t_1 corresponds to a rate change from the router into the channel. At time t_0 therefore, when that rate change is delivered, the algorithm observes the changes at times t_1 and t_2, and by its very definition removes the one at t_1. This is a contradiction to the assumption that a rate change is delivered at time t_1. □

A direct result of this theorem is that we can bound the total number of fluid rate changes processed by the simulation. Assuming that IDEs are deployed by all receivers (or that all channels between fluid components have non-zero latency), for any period of simulation time T we can put an upper bound on the number of rate change

events that are delivered, across every link in the model. This smoothing then assures that an uncontrolled exponential explosion of rippling events cannot occur.

Corollary 1 *For any fluid model, let L' be the average link insensitivity (latency plus packet transmission time), let N be the number of bidirectional links, and let T be the simulation duration time. Then the number of rate change events received by network components is no greater than $4NL'T$.*

4.2 Mixing Packet and Fluid Flows

Our approach to mixing packets and fluid flows has two basic components. The first is to base the decision of whether to drop a packet—and a queueing delay if it is not dropped—on the state of a joint packet-fluid router. Conceptually the only difference between what happens to a packet in the mixed model and a pure-packet model is that the state of the router's output buffer (which determines whether a packet is dropped and the queueing delay) reflects a fluid formulation, not a packet formulation. The second component is to additionally represent the packet flow as a fluid, and have it influence the fluid flows in the way any fluid flow affects another—through competition for service and storage in a fluid buffer. In this way the packet stream contributes to the overall state of the mixed model router.

When considering how to represent a packet stream as a fluid, our twin goals are to try to be efficient, and to effect the transformation in such a way that the total volume of fluidized packets that is presented to the fluid queue is in exact correspondence with the number of packets that arrive at the queue. We accomplish both goals using a methodology of *observing* packet arrivals during one measurement interval, and *reporting* an arrival rate in the next, based on the observations. If the measurement interval is large, one rate change might reflect many packet arrivals (although the flip side is that if the measurement interval is too large, the packets output rate is not responsive to fast changes).

First, for the purposes of interacting with fluid flows, we can aggregate all packet flows that share an output fluid buffer, and represented them using a fluid flow. The transformation is based on observed packet arrivals over a measurement interval m, defined as follows. If a packet with B bytes arrives and at least m time has elapsed since the last packet arrival, the packet flow rate λ_p is set equal to B/m, and a timer is scheduled to fire in m units of time. While the timer is running, any new packet arrival is noted; suppose that when the timer fires that A new arrivals were seen, with an aggregate arrival of B_A bytes. When the timer fires, if $B_A > 0$ then λ_p is set to B_A/m and the timer is rescheduled to fire again in m units of time. If instead there were no additional arrivals by the instant the timer fires, then λ_p is set to 0 and the timer is not rescheduled. The changes in packet flow rate are treated by the fluid queue just as would any input flow change of a regular fluid flow.

By construction, the volume of fluid representing packets presented to the fluid buffer corresponds exactly to the number of bytes in packets that arrive to the queue. If m is much less than the packet inter-arrival time, then every packet will tend to cause two rate changes—one to turn the packet flow on, and another to turn it off. Increasing m has the advantage of reducing the number of rate change events, but the disadvantage of delaying the impact of new arrivals on the queue. We correspondingly modify m dynamically with every timer firing. Using an exponentially decayed measured packet arrival rate, m is set to span the time needed for an estimated τ packets, where τ is a relatively small parameter (and m is strictly bounded from above to ensure responsiveness).

4.2.1 Queueing Delay Model

Our model for queueing delay assumes that packets are served in FCFS order. It maintains a variable t_p that records the instant when the last known packet to enter the buffer will (or has) fully departed. If a new packet arrives at time $t < t_p$, then some other packet is in queue ahead of the new arrival, else the queue is empty of packets. In the former case we compute the delay as t_p plus the time needed to process all fluid arrivals since the arrival of the last packet; in the latter case we compute the delay as the time needed to work through the existing fluid backlog (excluding the

fluidized representation of packets). In both cases t_p is reassigned to be the delay plus the transmission time of the packet.

It may happen that the storage requirements of the new packet arrival exceed the modeled residual buffer capacity at the instant of arrival. However, we don't automatically drop the packet. The mechanics of modeling loss are considered next.

4.2.2 Packet Loss Model

In the classical formulation of a fluid buffer, when the buffer is full there is never an instant when the buffer temporarily has enough residual capacity to accept a bulk arrival like a packet. Mixing packet and fluid arrivals at a congested buffer so as to accurately capture packet loss behavior is a delicate operation. The heuristics we've developed are just that—heuristics based on a lot of experimentation, whose principle value is that they seem to work.

Let t_{last} reflect the time of arrival of the last known packet, let δ_{last} be the level of the fluid buffer at that time, and let b be the buffer size. When a new packet of size A arrives at time t, we project what the level of the buffer would be at t if the buffer had no capacity limit, and the fluid arrivals had no loss. This value, say Z, is just δ_{last} plus $(t - t_{last})$ times the buffer level growth rate (assuming no fluid loss). Then

- If a transient measure of relative traffic (to be defined more precisely, momentarily) indicates that a fraction θ or more of the traffic is fluid, then we compare Z to b.
  - If $Z > b$, we choose randomly with probability $1 - \theta$ to drop the packet.
  - If $Z \leq b$, the packet is not dropped.
- If θ or more of the traffic is packet-oriented then
  - the packet is dropped when $Z > b$.
  - when $Z \leq b$ and $Z + A > b$, we drop the packet with probability $1 - (\theta - Z)/A$.

The transient measure of arrival rates to the the buffer is based on a relatively short measurement interval. The volume of packet arrivals over this interval is maintained, as is the volume of fluid arrivals. Rates are constructed by dividing the aggregate received bytes by the length of the measurement interval. These rates are used to construct the relative fraction of fluid arrivals in the preceding measurement interval. θ, used above, is a parameter that seems to work satisfactorily when set to value 0.8.

5 Accuracy

We are now in a position to analyze the accuracy of the proposed approach. Our model formulation is designed to keep the TCP portion of the network model from introducing error related to latency.

Suppose that the first bit of a segment leaves a TCP sender at the same instant in both the discrete and fluid formulations. If the latency imposed upon that bit is the same in both models, then the first bit in the fluid model will reach the input delay element in the TCP receiver at the same instant as the first bit of the segment in the discrete model. Following this, the time needed to fully transmit the segment across the last link—the segment length divided by the bandwidth— elapses before the segment is fully received in the discrete model. At this instant the entire segment is presented to the TCP receiver in the discrete formulation, and the first bit of the segement is presented in the fluid formulation. Notice that precisely the same argument applies for the transmission of an acknowledgement-bearing header from TCP receiver to TCP sender. We formalize this observation with the statement of a lemma.

Lemma 1 *If the network latency for a bit is the same in both discrete and fluid models of TCP, and if the first bit of a segment (alt., acknowledgement) departs a TCP agent at the same instance in both discrete and fluid models, then the segment (alt., acknowledgement) is recognized by the receiving agent in the discrete model (if it is not lost) at the same instant that the first bit of the segment (alt., acknowledgement) is presented to the receiving agent in the fluid model.*

We note in passing that the network latency for a bit will be the same in both formulations if no packet (or fluid) encounters congestion in any fluid queue it visits. The exact correspondence between fluid and packet models we now work to establish is limited to this special case. In light of the fact that interesting network simulations invariably have congestion and packet loss, the value of formally establishing the correspondence when it can be shown is in demonstrating that the fundamental basis for the approach is sound. Real-life phenomena like congestion will force some of the measures we compute to be approximations.

The discussion to follow will at times refer to fluid behavior "just before" a given time t, at a time we denote by $t-$. Formally, we can always order all discrete events in a simulation by timestamp, $s_0, s_1, s_2, \ldots$. For any time t, we let $i(t)$ be the largest index such that $s_{i(t)} < t$, and define $t-$ (somewhat arbitrarily) as $t- = (t + s_{i(t)})/2$. The key idea is to identify a time just before t that has no other system events between it and t.

Next we introduce two definitions describing desirable behavior of the fluid model.

Definition 1 *Let $\lambda_A(t)$ describe the application data offered rate function, that is, what the rate function would be if TCP never altered the application data through the feedback control. We say the fluidized application model is* synchronized *with the discrete model if, for every t such that $\int_0^t \lambda_A(s)\ ds/MSS$ is integral and $\int_0^{t-} \lambda_A(s)\ ds/MSS$ is not, then the discrete model offers a full segment to the TCP agent at time t. Furthermore, for every time t, if $\lambda_A(t) = 0$, then $\int_0^t \lambda_A(s)\ ds/MSS$ is integral.*

Effectively, the application offered load function is synchronized with the discrete model if the first bit of a fluidized segment hits the TCP sender at the same instant that the discrete model releases the discrete version of that packet, and if transition of the application offered load rate to zero implies that fluid equivalent to an integral number of segments has been offered.

A second definition describes a continuity property of the fluidized network model.

Definition 2 *A fluidized network model is said to* preserve flow movement *if for every flow f, switch S, and time t, if the flow rate for f out of S at t is zero, then the flow rate for f into S is also zero, and no fluid for that flow is buffered.*

Flow movement preservation is just a formal way of saying that once a switch begins to allocate output bandwidth to a flow, it can only withhold all bandwidth to that flow after its input rate has become zero, and all buffered fluid associated with that flow has been sent.

The definition of application flow synchrony is a statement about how a modeler represents the application offered load. However, we need a stronger statement of synchrony, because the offered load rate function is not the same as the accepted load rate function—the TCP sender is able, through feedback, to alter the rate at which the application flow is taken from the source. We need for *that* flow to be synchronized in the same sense as the offered load. To see that it is, we need first the following result.

Theorem 2 *Consider a fluid simulation where all application flows are synchronized with the discrete model, the network model preserves flow movement, and network bandwidth is always available to a TCP sender. For every TCP sending agent at every time t, if $\lambda_{send}(t) = 0$ and $\lambda_{ack}(t) = 0$, then $LBS(t)/MSS$ is integral.*

Proof: We can describe a TCP sending agent's output behavior in terms of rounds, during each of which $\lambda_{send}(t) > 0$ and $\lambda_{ack}(t) = 0$. A round ends at t if $\lambda_{ack}(t) = 0$ and $LBS(t) - LBA(t) = cwnd(t)$. In the absence of data loss, the rules governing $cwnd$ evolution clearly indicate that an integral number of segments are sent each round, if $cwnd$ starts off being an integral number of segments. Data loss is detected at segment boundaries; $cwnd$ and $ssthresh$

are modified upon detecting loss, but remain integer multiples of MSS. Thus, at the end of every round—even in the presence of data loss—an integral number of segments have been sent. The theorem's statement formalizes this observation. □.

We are prepared now to state the main result.

Theorem 3 *Suppose that every fluidized application flow is synchronized with the discrete model, and suppose that the fluidized network model preserves flow continuity. Then under the conditions of Lemma 1 and when there are no lost segments, the first bit of every segment and departs the application at the same time in both discrete and fluid models, departs a sending TCP agent at the same time in both discrete and fluid models, and is acknowledged at the same time in both discrete and fluid models.*

Proof: The departure times of segments (or acknowledgements) from application, TCP sender, or TCP receiver in the model can be ordered by timestamp. We prove the result by induction on this ordering. For the base case we look to the first segment, which must be a departure from an application. By synchrony of the application flow, the first bit of the application fluid reaches the fluid TCP agent precisely at the same instant as the full segment reaches the discrete TCP agent, proving the base case. For the induction hypothesis assume there exists $n > 1$ such that the claim is true for all $n-1$ consecutively ordered segments and acknowledgements, and consider the n^{th} such. There are three cases to analyze, depending on the departure point of the n^{th} segment.

Departure from TCP receiver: For an acknowledgement to depart at time t, it is necessary that the segment being acknowledged be first recognized by the receiver at time t. By the induction hypothesis the first bit of that segment departed the TCP sender at the same time in both discrete and fluid models; by Lemma 1 the first bit of that segment is recognized by the TCP receiver at the same instant in both models, hence the acknowledgement for that segment departs at the same time in both models.

Departure from Application: The segment departs the application at time t, for one of three reasons. (1) If $LBS(t) - LBA(t) < cwnd(t)$ and the last bit of the last segment sent from this agent left at time $s < t$, then the application is the bottleneck, and the segment is being sent because a new segment is finally available at time t. Since the departure of the previous segment from the application was synchronized in fluid and discrete models, by the synchrony property of the application fluid model, the next segment will as well (because during the epoch from s to t the output flow behavior is governed by the offered load rate). (2) If however $LBS(t) - LBA(t) < cwnd(t)$ and the last bit of the last discrete segment departed precisely at time t, the bandwidth provided to the sender is the bottleneck. This implies that the last bit of the previous fluidized segment leaves at the same instant as the last bit of the previous discrete packet, because the first bit left at the same time in both models (by the induction hypothesis), and the added delay until the last bit leaves in both models is the same—the segment size divided by the bandwidth. (3) If $LBS(t-) - LBA(t-) = cwnd(t-)$ and $\lambda_{ack}(t-) > 0$ then the TCP sender's outflow is constrained by the incoming acknowledgement rate. The rate at which the application data is accepted is identical to the rate at which the data is sent. Since the last segment sent out was synchronized in discrete and fluid models, and the application and output rate functions have been identical since, the fluid and discrete models must be synchronized at t as well. (4) If $LBS(t-) - LBA(t-) = cwnd(t-)$ and $\lambda_{ack}(t-) = 0$, then the release of the n^{th} segment in the discrete model is triggered by the arrival of an acknowledgement at t. Since $\lambda_{send}(t-) = 0$, then $\lambda_{app}(t-) = 0$ as well. Let s be the greatest instant less than t such that $\lambda_{app}(s-) > 0$ and $\lambda_{app}(s) = 0$. The application is signaled at s to stop because the TCP sender can no longer send. This implies that $\lambda_{send}(s) = \lambda_{ack}(s) = 0$, which by Theorem 3 implies that $LBS(s)/MSS$ is integral—but $LBS(s)$ is precisely the amount of traffic that the application has passed to the TCP sender. Now as a result of the induction hypothesis and Lemma 1, the arrival at t of the acknowledgement in the discrete model corresponds to the arrival at t of the first bit of the fluidized acknowledgement in the fluid model. Thus at t $\lambda_{app}(t) > 0$, producing the first bit of segment n, as required.

Departure from TCP sender: This case is identical to the Departure from Application case, because a segment departs the application if and only if it instantaneously departs from the TCP sender.

Thus the n^{th} segment departs in the discrete model exactly at the same instant as the first bit of the fluidized version of that segment departs, completing the induction. □.

6 Event Reduction

We can quantify the degree to which a fluid formulation of TCP reduces the number of events needed to simulate a given transfer. Our approach is to count the number of events needed to simulate a flow of T segments using an ordinary packet-oriented model, versus the number of events needed to simulate T segments using the fluid flow formulation. Our analysis is based on parameters R—the round-trip time, assumed for this analysis to be constant, λ_{app}—the offered application data rate (also assumed to be constant, and less than λ_{bw}), and s—the value of *ssthresh*, in units of MSS-sized segments.

For this analysis we exclude the events needed to simulate the flow in the interior of the network. These contribute of course to the overall event counts, but will be considered separately.

In a packet-oriented simulation we count four events for every segment sent: the segment sent and received, and the acknowledgement sent and received, We thus take $4 \times T$ as the cost of sending the transfer, assuming no data loss, and ignoring session set-up and tear-down costs.

In our fluid flow model events occur to start and stop flows. As long as $cwnd$ is a constraint, then the sender will start a flow and let it run until the number of unacknowledged bytes is $cwnd$, then stop. A pulse of flow is sent (two events, started and stopped) and received two more events), and a pulse of acknowledgements are sent and received (four events), for a total of eight events per pulse.

It will be notationally cleanest to compute the packet and fluid event counts at the granularity of TCP rounds. Observe that the number of segments sent in a round increases so long as they are all send before the acknowledgement for the first of them comes in. There is a transition window size, which we call n_s, after which $cwnd$ ceases to be a constraint. n_s is the solution to $n \times MSS/\lambda_{app} = R$, that is, $n_s = R \times \lambda_{app}/MSS$.

First consider slow start mode. At the end of the k^{th} round ($k = 1, 2, \ldots$) exactly $2^k - 1$ segments will have been sent, requiring $4 \times 2^k - 1$ events in the packet oriented case, and $8 \times k$ events in the fluid case. (For simplicity we assume here that s is a power of two). In this mode the ratio of the number of events needed in a packet simulation to the number needed in a fluid simulation is

$$\begin{aligned} r(T) &= \frac{4 \times (2^{\log T} - 1)}{8 \log T} \\ &= \frac{T}{2 \log T}. \end{aligned}$$

The growth of $r(T)$ in this region is not quite linear in T. At the end of slow start mode exactly $2s - 1$ segments will have been sent.

Now consider congestion avoidance mode, and transfer lengths at the end of a complete round; these have the form of the form $T = 2s - 1 + (m-1)s + m(m-1)/2$, for the length at the end of the $(\log s + m + 1)^{st}$ round, $m = 1, 2, \ldots$. For m small enough that $s + m - 1 \leq n_s$ the pipe is not yet full. The packet-oriented simulator requires $4T = 2(m^2 + (2s-1)m + 2(s-1))$ events, the fluid simulator requires $8(\log s + 1 + m)$, for a ratio of

$$\frac{m^2 + (2s-1)m + 2(s-1)}{4(\log s + 1 + m)}.$$

The growth of this function is asymptotically linear in m, and m is asymptotically proportional to $\sqrt{T}$, so $r(T)$ grows in proportion to $\sqrt{T}$ in this regime. Once $\log s + m - 1 > n_s$ however, the packet simulator continues to accrue

		$R = 1ms$			$R = 100ms$		
ssthresh	Application rate (seg/s)	10	100	1000	10	100	1000
16	100,000	472	17861	178061	472	76156	
256	100,000	255	1655	16055	255	5821	
16	1000	21	201	2001	472	17861	178061
256	1000	21	201	2001	255	1655	16055

Table 2: Transfer lengths achieving event reductions factors of 10, 100, and 1000

computational cost at the rate of 4 events per segment, while the fluid simulator accrues none, at least until the flow terminates. If we denote by m_s the round number (in congestion avoidence mode) where the pipe fills, we see that it satisfies $m_s = n_s - \log s - 1$, at which point $T_s = (m_s^2 + (2s-1)m_s + 2(s-1))/2$ segments have been sent. For $T > T_s$ the ratio of packet events to fluid events is

$$r(T) = \frac{T}{4(\log s + 1 + m_s)},$$

which is clearly linear in T.

Figures 6 and 7 show plots of $r(T)$ where the y axis is the ratio of packet events to fluid events, and the x axis is the transfer length T. Each graph varies T, and provides curves for values of R that span three orders of magnitude. The graphs differ in their assumption of s (16 and 256), and λ_{app} (1000 and 100,000 segments per second).

The relative gain of the fluid model over the packet model is tied very much to how quickly the pipe can be filled. The relationship $n_s = R \times \lambda_{app}$ shows that small R or small λ_{app} can fill the pipe faster than large R or large λ_{app} by making the pipe smaller. Situations with large *ssthresh* lead to pipe fill faster than situations with small *ssthresh* because the growth of the congestion window is so much faster in slow start mode than in congestion avoidance mode. These simple facts are reflected in the graphs. Beyond this, the graphs quantify the intuition that performance gains are modest for short transfers, but are potentially very significant for long transfers.

Another useful summary of this data gives the transfer lengths (in segments) necessary to achieve a given a given acceleration. Table 2 does exactly this, listing for R values of $1ms$ and $100ms$ the transfer lengths needed to achieve event reductions of factor 10, 100, and 1000. To put the application segment rates into perspective, recall that a typical TCP segment size is about 1000 bytes. The 1000 segment/sec and 100000 segment/sec application rates correspond to $1Mbs$ and $100Mbs$ offered load, respectively. The smaller of these is quite reasonable, and we see from the table that in this regime very significant performance gains are possible with moderate sized transfers.

7 Experiments

We next empirically evaluate accuracy and speedup. Our methodology is to measure the goodput, round-trip time, and packet loss rate of a reference packet stream, comparing the results when the background traffic is purely packet oriented, and when it is fluid-based. We conduct these experiments on two topologies. The first is the classic "dumbell" topology used for many TCP studies. The second is a large scale network with realistic topology; this one contains the potential for the sort of multi-hop event explosion characteristic of prior fluid models. Nevertheless, we find that on all measures the technique is usually quite good, and provides significant speedup.

7.1 Dumbbell Topology

The dumbbell topology connects a set of servers with a set of clients. Every server has a direct connection to a common router, every client has a direct connection to a separate common router, and the two routers are connected

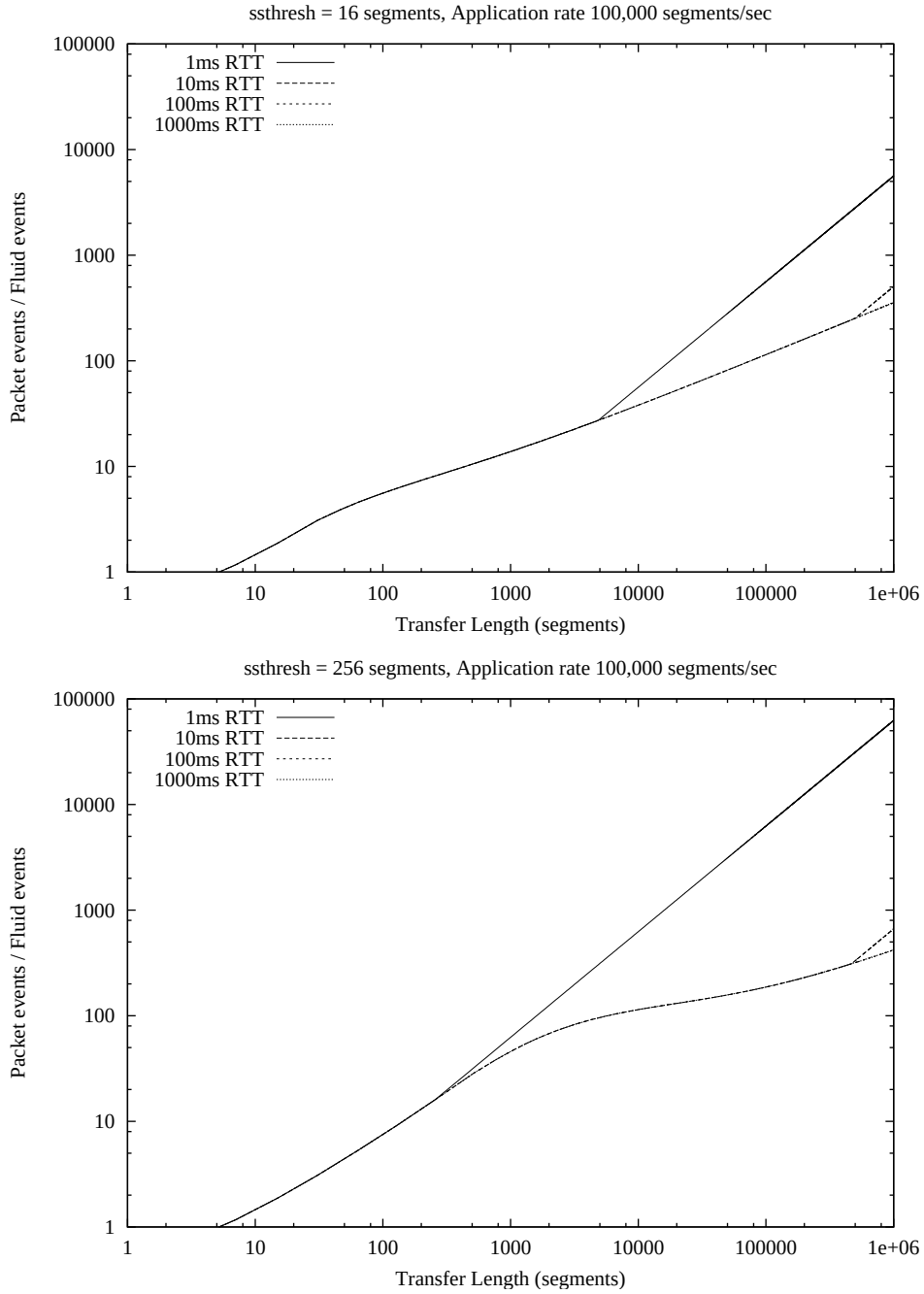

Figure 6: Event acceleration of fluid model over packet model, fast application data rate

through a bottleneck link. In each experiment we consider the goodput (number of delivered bytes per unit time), round-trip delay, and lost packet rate of a packet-oriented TCP stream that competes for resources with fluid-oriented TCP streams. Each experiment we conduct with fluid streams we also conduct after replacing each fluid stream with an equivalent packet stream. Comparison of corresponding experiments illuminates differences in the flow metrics.

One set of experiments uses 10 background flows, another set uses 100 background flows. In both sets twenty percent of the flows push 5Mb per TCP session, and eighty percent push 0.5Mb per session. The "reference" packet flow is a 5.0Mb transfer. Each flow source is controlled by an on-off process where a flow is off for exponentially

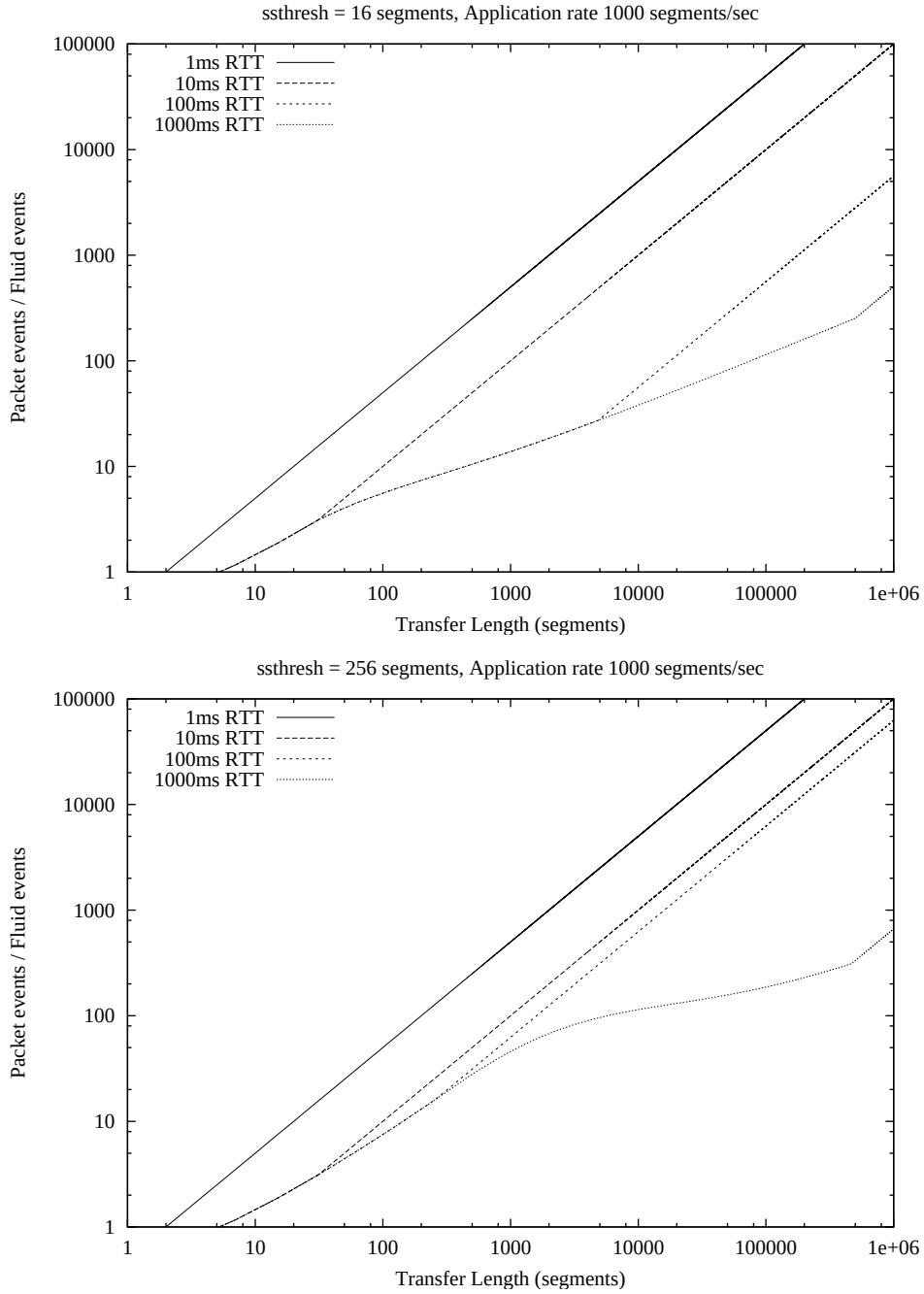

Figure 7: Event acceleration of fluid model over packet model, slow application data rate

distributed period of time, with mean 10 seconds, and is on for as long as it takes to set up a new TCP session, deliver and have acknowledged the 0.5Mb or 5.0Mb bulk of data it generates, and tear the session down.

Each connection between host and router has a bandwidth of 10Mbs, and has 1 ms latency. We vary the latency of the bottleneck link between 1 ms, and 10 ms; we vary its bandwidth between 1.5Mbs, 15Mbs, 150Mbs, and 1500Mbs. For every experiment we use the bandwidth-latency product to define the size of the buffer (in bytes) for access to the bottleneck link. There are sixteen combinations of all experimental variables, although the two 1-ms/1.5Mbs experiments are excluded because they do not allocate enough buffer space for even one packet.

7.1.1 Accuracy

Figure 8 describes the accuracy of packet loss rate and goodput for the experiments with a 1ms latency on the bottleneck link; Figure 9 does the same for the 10ms case. In these (and other graphs) the lines describe measurements associated with the reference packet stream in the hybrid simulation, while discrete points describe measurements of packet streams from the pure packet simulation. A line whose label contains "Nbf describes the measurements associated with N background flows. In the 1ms case the packet and hybrid measurements are in very close agreement when the bottleneck bandwidth is 150Mb and 1500Mb. The experiment with 100 background flows shows a wide spread of packet loss probabilities in the pure packet case, in a range that lies distinctly lower than in the fluid case. Interestingly, this large difference in loss rates has very little impact on comparative goodputs in the same experiment, because the system is utterly saturated, and no flow is getting much bandwidth. The 10ms case illustrated in Figure 9 is much the same. Recalling that the buffer size is the latency-bandwidth product, the 10ms case allows for a few packets to be buffered with the bottleneck bandwidth is 1.5Mb. However, this bandwidth is insufficient to support either 10 or 100 flows, yielding high loss rates and very low goodput. As before, excellent agreement is achieved on system configurations that yield better goodput.

Figure 10 shows the round-trip delay for all experiments, where we see that the model very accurately captures latency.

7.1.2 Speedup

Figure 11 illustrates how much faster the hybrid model executes than the pure packet model. We illustrate configurations with one packet and 10 fluid flows, and one packet with 100 fluid flows. The hybrid model actually takes longer when the bandwidth is 1.5Mb; this is not surprising, because the TCP window sizes are small, and the fluid model gains efficiency on large windows.

Speedup is affected by the reduction in the number of events needed by a fluid formulation, but also by the increase in the cost-per-event of the fluid formation. The latter consideration depends on the mixture of fluid events at hosts and fluid events at routers, as the costs are not identical. For this particular set of experiments we estimate that a fluid event costs $\delta = 2.5$ more than an ordinary packet event. Measurements also show that an average (averaging over the mixture of transfer lengths) fluid flow costs $\epsilon = 0.00555$ times the equivalent packet average. If the cost of a packet event is 1, and K packet events are needed per flow, then for these experiments with F background flows the speedup is approximately

$$\frac{(F+1)K}{\delta K(1+F\epsilon)} = \frac{F+1}{\delta(1+F\epsilon)}.$$

Using the specific values of $F \in \{10, 100\}$, $\delta = 2.5$, and $\epsilon = 0.00555$ this formula predicts speedups of 4.1 for 10 background flows, and 26 for 100 background flows. These predictions are validated by Figure 11 where values very close to these figures are observed in uncongested configurations. The speedup equation also illustrates inherent limits. As F grows the speedup is limited from above by $1/(\delta\epsilon)$ independent of F, in this case 72.

7.1.3 Interaction

We next illustrate that fluid and packet flows interact with each other in the expected fashion. We use a simplified bottleneck topology with one packet and one fluid flow. The buffer capacity at the router nearest the clients is 125K bytes, the capacity at the router nearest the servers is 12.5K. All links have 10Mbs bandwidth; the bottleneck link latency is 100ms and the latencies on all other lines is 10ms. A flow consists of a repeating on-off process that is off for a random period of time (exponentially distributed with mean 10), and when on carries 50Mb.

In each of two experiments one of the flows begins at time 0, and at time 900 the other flow starts. In such an experiment we consider the behavior of the congestion window of the flow that started at 0, and the queueing delay of

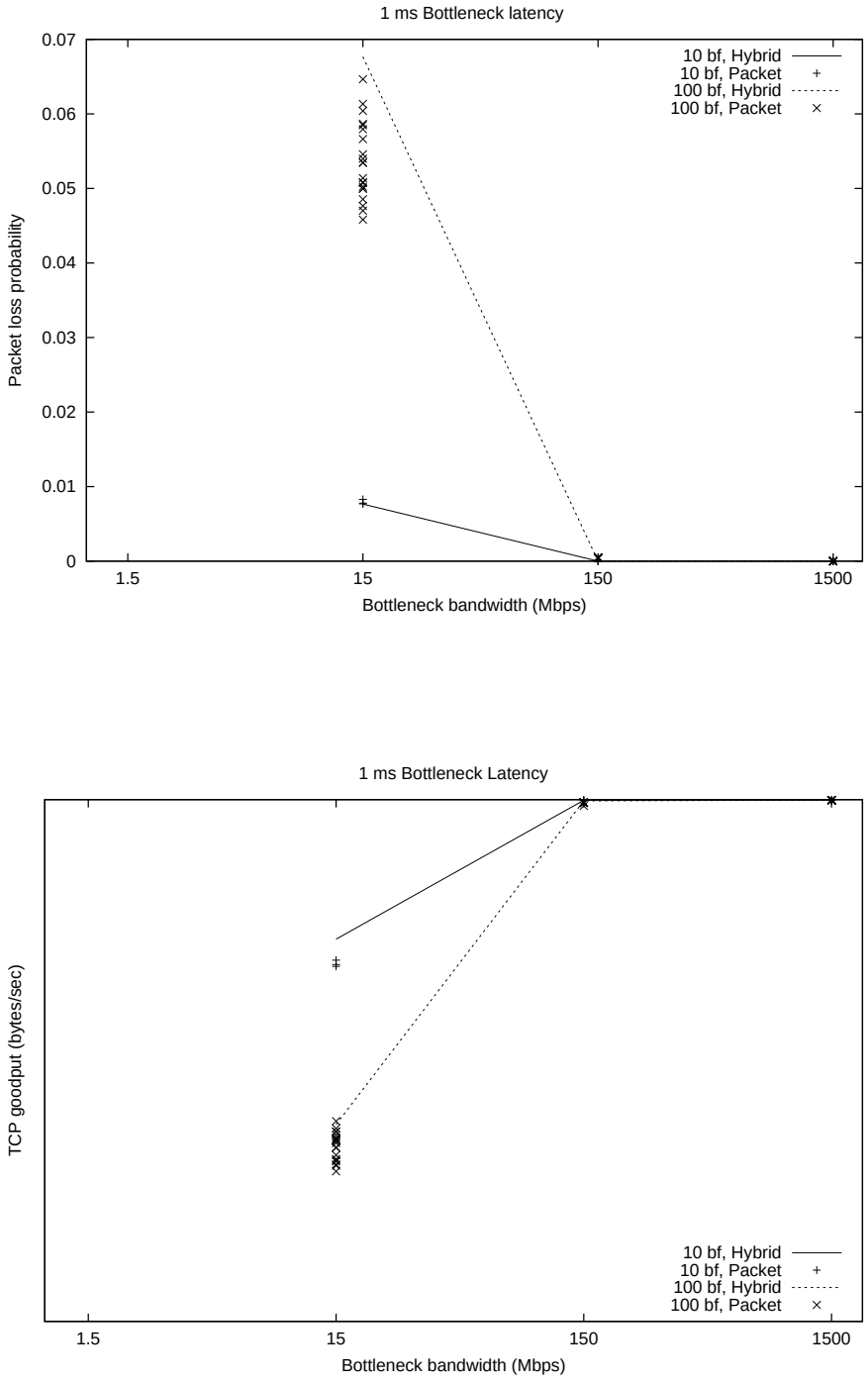

Figure 8: Accuracy of Hybrid Model on Bottleneck Topology, 1 ms Bottleneck Latency

packets associated with that flow. One experiment starts the packet flow and examines the impact that the later fluid flow has on it, the other experiment reverses the roles.

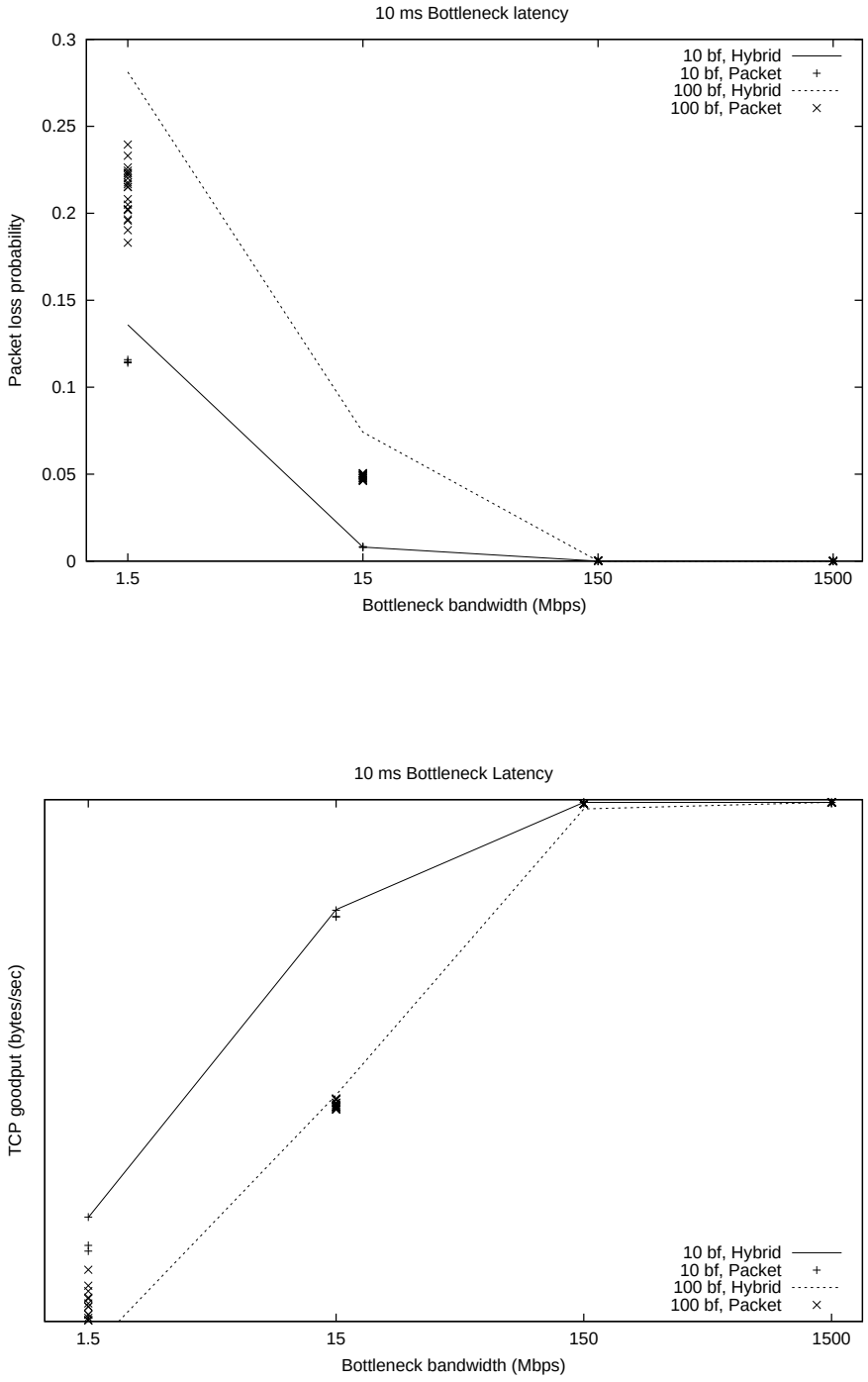

Figure 9: Loss and Goodput Accuracy of Hybrid Model on Bottleneck Topology, 10 ms Bottleneck Latency

Figure 12 shows the impacts the flows have on each other's congestion window sizes. In both cases the base flow's window grows in accordance with TCP windowing rules until the window grows so large (before the pipe fills) that

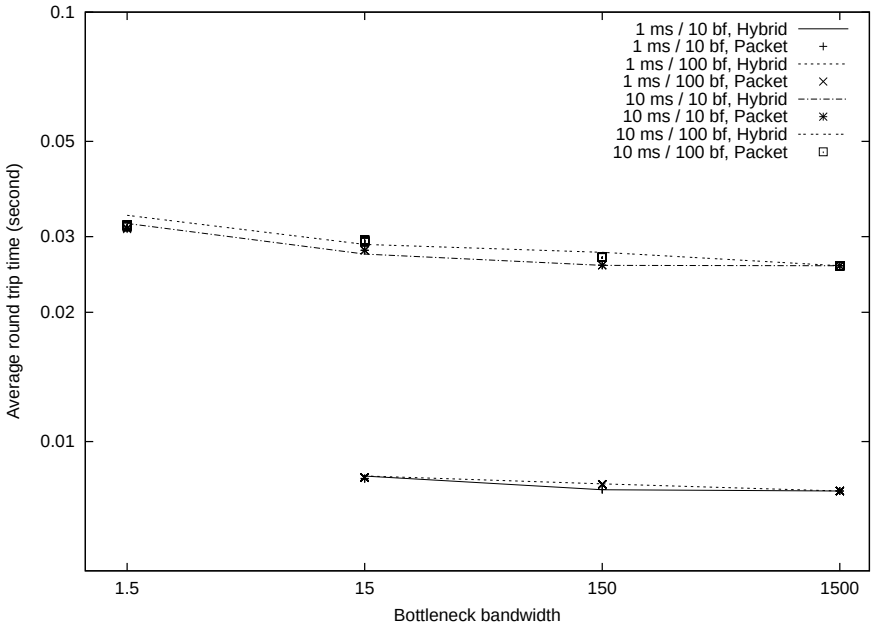

Figure 10: Latency Accuracy of Hybrid Model on Bottleneck Topology

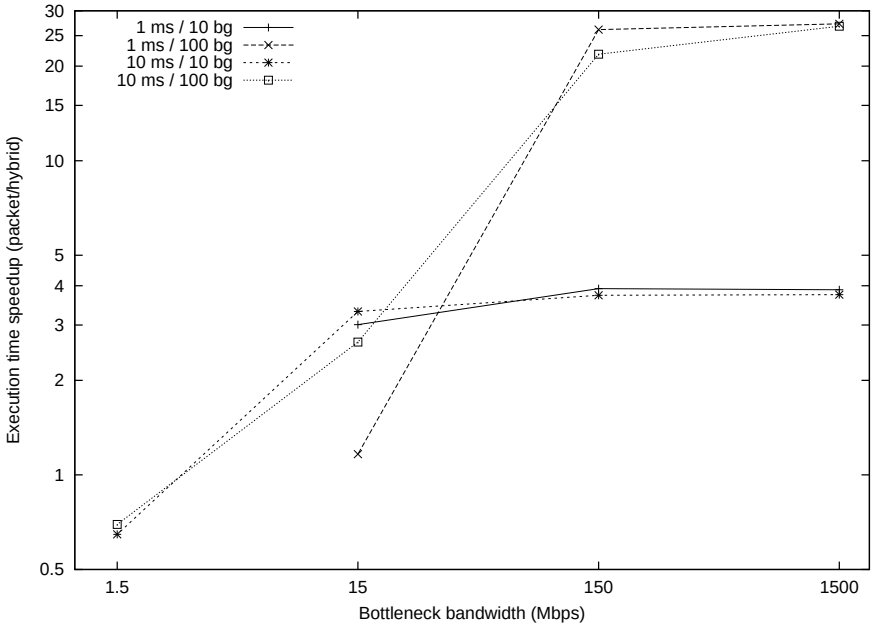

Figure 11: Speedup of Hybrid Model on Bottleneck Topology

buffer limitations cause a packet loss, triggering slow start. The "gaps" in the plots of the fluid flow's congestion window are artifacts of the samples being taken only at fluid events. One can see the doubling of the congestion window in slow-start. Up to time 900 the behavior is the same in both experiments. After time 900 the effect of the "other" flow is plainly evident, and the congestion window behaviors are similar, but not identical.

Figure 13 shows the impacts the flows have on each other's queueing delays at the router with smaller buffer capacity. Each tick in the experiment where the packet flow is the base marks the delay of a packet; each tick in the other experiment reflects a queueing measurement at the instant a fluid event is processed. The latter graph is sparser as a result. The two graphs are similar in exhibiting tall spikes of delay, and in being limited at 0.01 (which is due to the buffer limit). Here again we see that packet and fluid flows interact intuitively.

7.2 Campus Network

We also evaluated the hybrid technique on a large model called the "campus network". This topology was developed for baseline studies in the DARPA Network Modeling and Simulation program. The campus is the basic unit of this topology, shown in Figure 14. Globally the network consists of campuses in a ring with some regularly placed chords. A campus has 23 LANs that collectively hold 504 client hosts, 4 server hosts, and 18 routers. One router connects the whole campus to other campuses in the network. The network is comprised of 20 such campuses.

The traffic pattern in our experiments has every client host establishing a TCP connection with a server host on a different campus network, more or less concurrently. All traffic crosses between ASes, and all traffic converges on the subnetwork holding a campuses four servers. The intensity of fluid interactions radiating out from the servers give rise to conditions where the classical fluid formulations suffer event explosion. Our experiments vary the bandwidth of links between campus subnets. Lower bandwidths induce more loss, and hence less opportunities for fluid formulations to aggregate. Ten percent of the traffic flows are pure packet, ninety percent are fluid. All flows are 0.5Mb in length.

Figure 15 shows that the error loss and goodput metrics in the fluid model are accurate in this context as well, on the order of 10% or better. Figure 16 shows that the latency predictions are extremely accurate. Figure 17 reflects the higher cost of extreme interaction when the pipe between ASes is small, with a speedup tending towards 7. The limits on speedup here are both the higher cost of events because of the hybrid formulation and the effects of flows interacting on links of limited bandwidth.

A final experiment on the campus network clearly shows the potential for significant speedups using the hybrid model. The inter-campus links in this experiment are all 1Gbs. A different traffic mix causes 90% of the flows to have endpoints in the same AS; half the packet flows are UDP and half are TCP. Traffic intensity is 1000 packets per client per second, and traffic path lengths are 5 hops on average. There are twenty campuses in the network, which yields a native model activity intensity of approximately 170 million model state changes per simulation second, where a state change is effected when a packet leaves or arrives at a new destination. (This estimate includes the assumption of one ack packet per sent TCP packet). This model is run on a parallel computer where a single CPU can execute approximately 500K events per second. If a single state transition required one event, and if parallelism could be exploited perfectly without additional overhead, then a minimum of 340 CPUs would be needed to execute this model so that the simulation clock advances faster than a real-time clock.

We affect the amount of work needed to advance the model (and hence the execution requirements) by using our fluid representation for most of the flows. We can affect the execution time by varying the fraction of flows that are pure packet. Figure 18 expresses *slowdown*—the ratio of wallclock execution time to simulation duration for a run—as a function of the percentage of flows that are purely packet oriented. The simulation runs are executed in parallel, using 20 CPUs. We see that for packet/fluid mixtures where 1% or less of flows are pure packet, the simulation runs faster than real-time. Execution is nearly a linear function of the fraction of pure packet flows, which is a reflection on the scalability of the mixed fluid-packet routers; there is no evidence of the sort of event explosion that has plagued

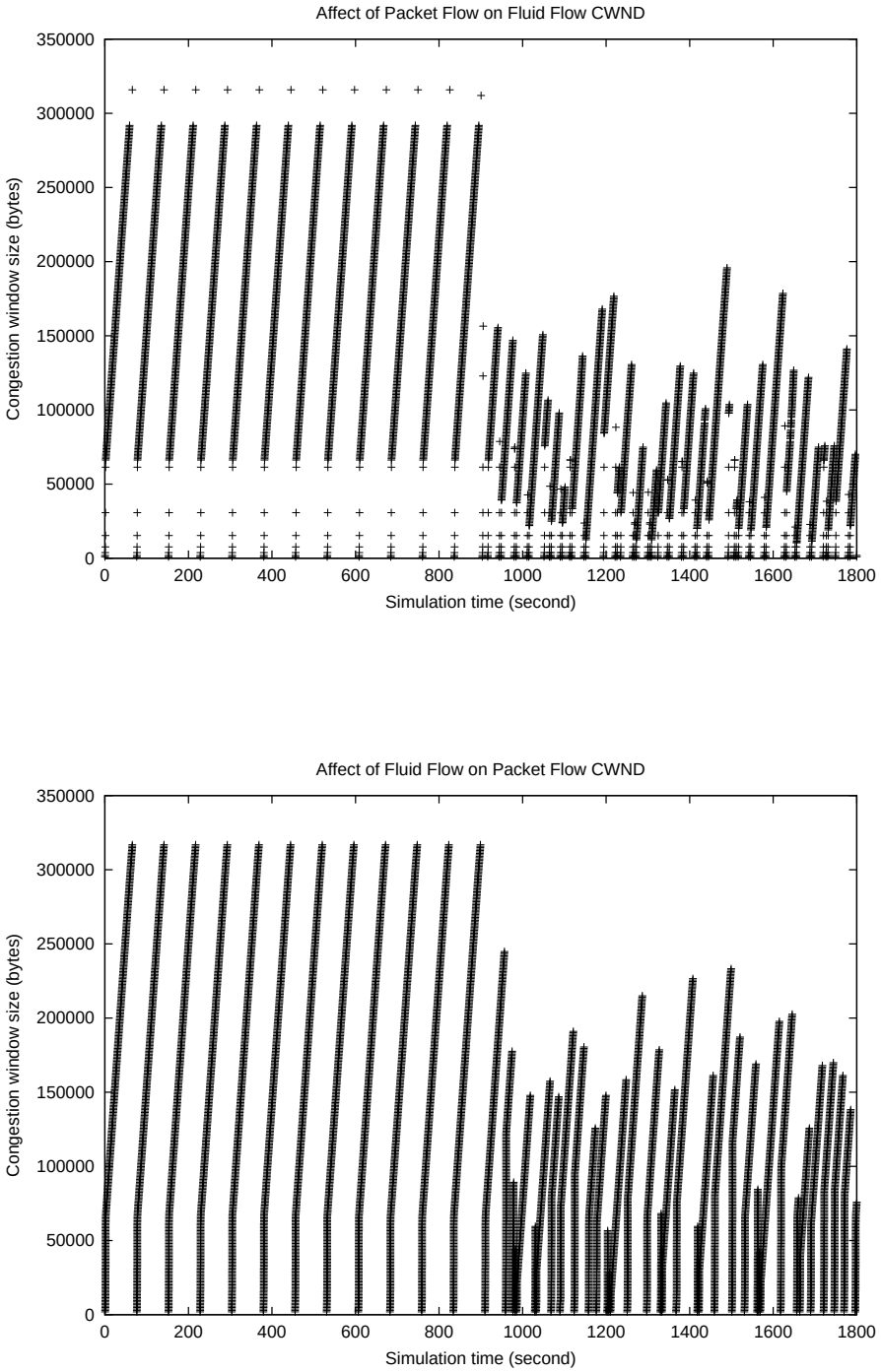

Figure 12: Interaction of Fluid and Packet Flows: Effects on Congestion Window

fluid simulations in the past.

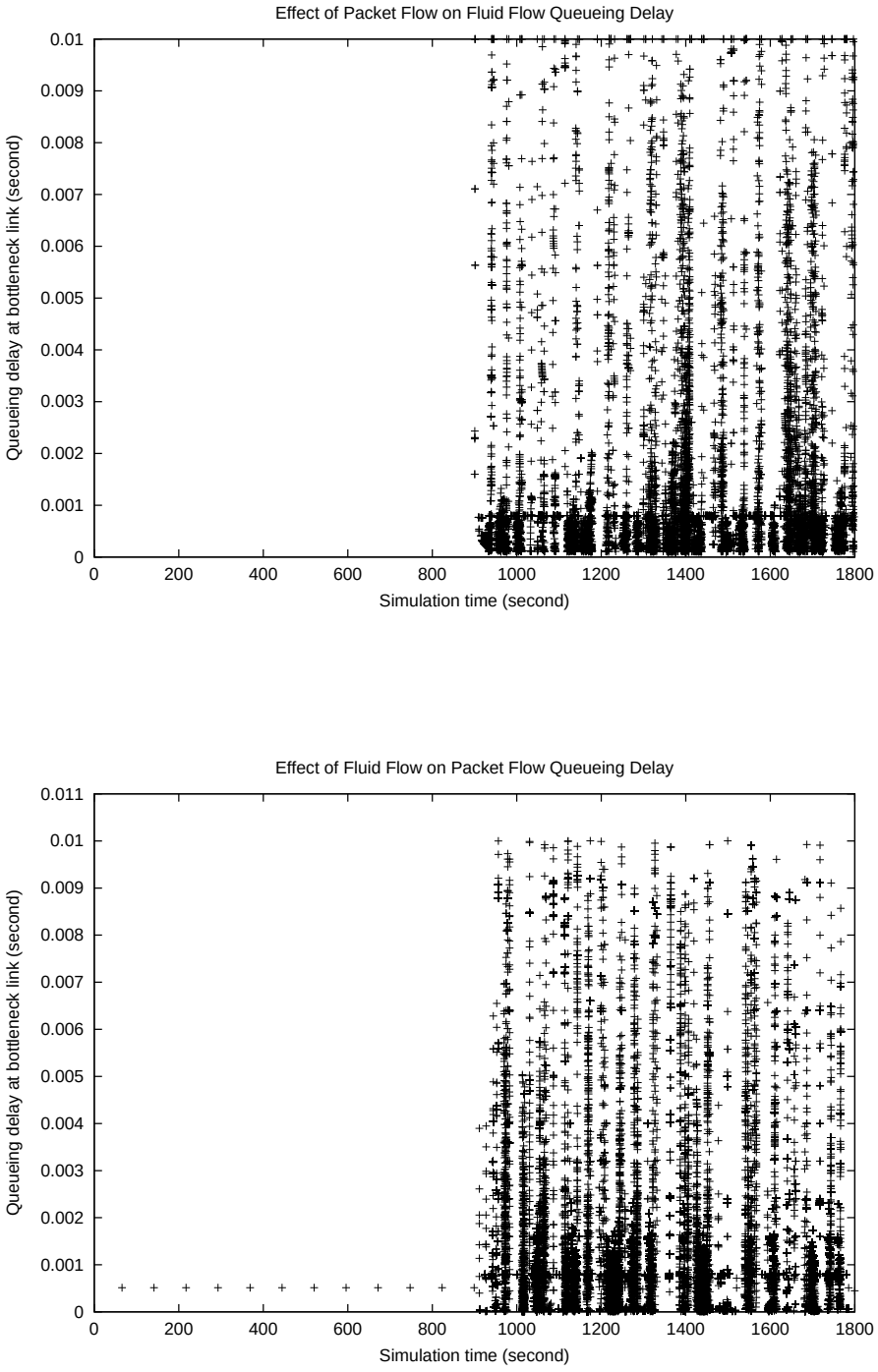

Figure 13: Interaction of Fluid and Packet Flows: Effects on Queuing Delay

8 Conclusions

This paper develops new methods for the generation of background TCP like traffic. Our technique models flows as fluids, but in contrast to most other fluid-based modeling approaches uses a discrete-event formulation. We show that

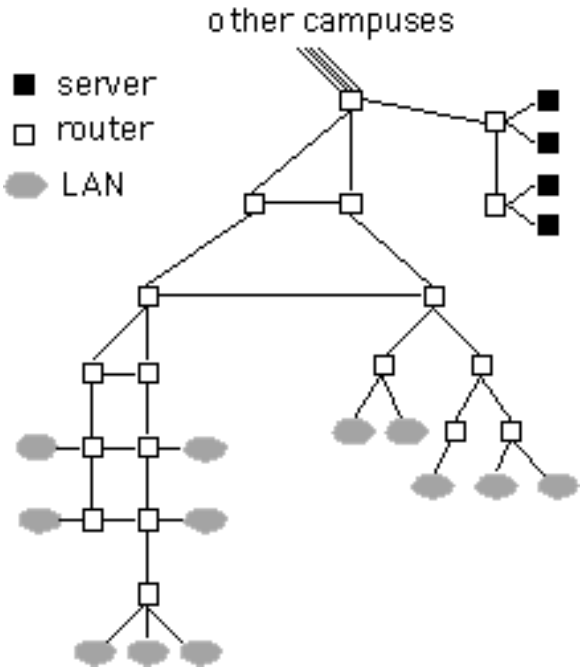

Figure 14: Topology of Campus

our formulation is rigorous exact (as compared to an equivalent packet formulation) over epochs when a flow suffers no loss. We quantify the reduction in the numbers of events used to model a fluid flow as a function of TCP parameters and effective round-trip time. We describe how fluid flows and packet flows can be handled together in a router, and study empirically the accuracy and speedup of our techniques on a classic topology and a large-scale topology. We find that accuracy is very good in operational regions where TCP typically operates, and see factors of 10 speedup on these networks.

The key limiting factor to speedup using our technique are flow conditions where there is significant packet loss. Our aggregation exploits epochs of time when a TCP session sends a round of data at constant rate; packet loss interrupts that round and creates a new event in the fluid description. Another limiting factor is that we represent each background individually; memory restrictions limit the number of flows we may represent. Future work in discrete-event background modeling is best focused on higher level model abstraction that efficiently capture flow interruptions caused by loss, and that aggregate multiple TCP-like flows on a link while maintaining an acceptable level of accuracy.

Acknowledgements

This research was supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-98 08964, NSF Grant EIA-98-02068, and Dept. of Justice contract 2000-CX-K001.

References

[1] J. Ahn and P. Danzig. Packet network simulation : Speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking*, 4(5):743–757, October 1996.

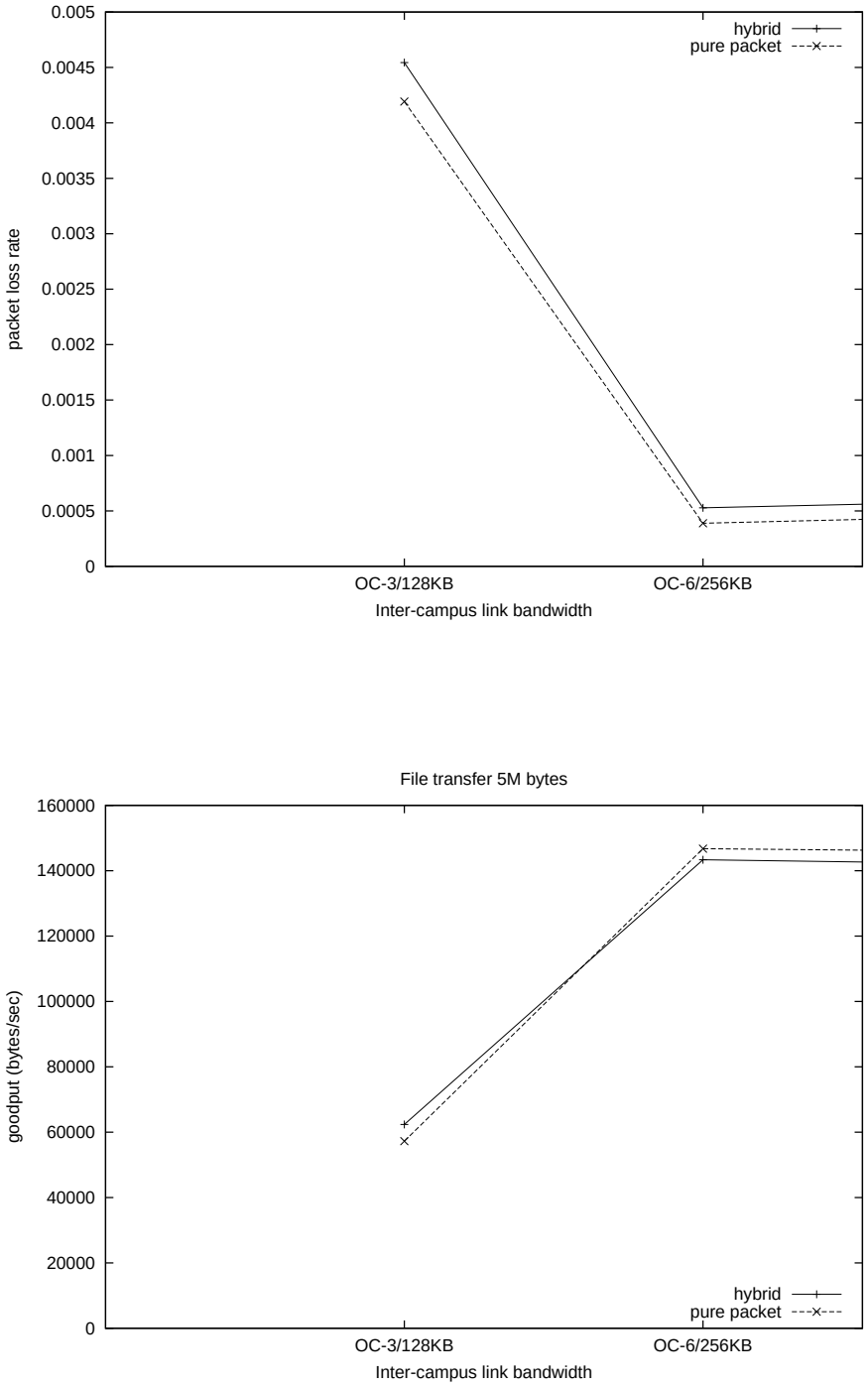

Figure 15: Loss and Goodput Accuracy of Hybrid Model on Campus Network Topology

[2] E. Altman, K. Avrachenkov, and C. Barakat. A stochastic model of tcp/ip with stationary random losses. In *Proceedings of SIGCOMM '00*, Stockholm, Sweden, September 2000.

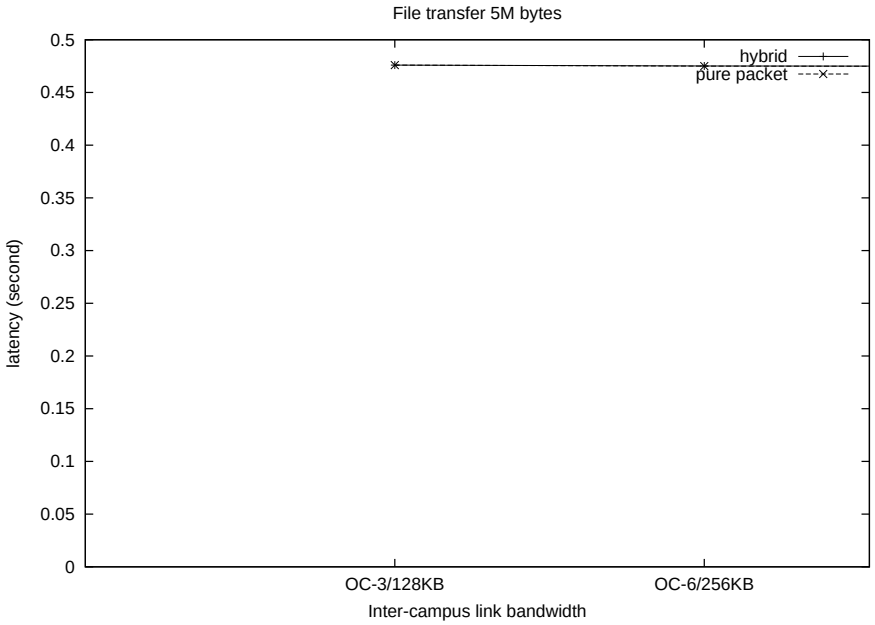

Figure 16: Latency Accuracy of Hybrid Model on Campus Network Topology

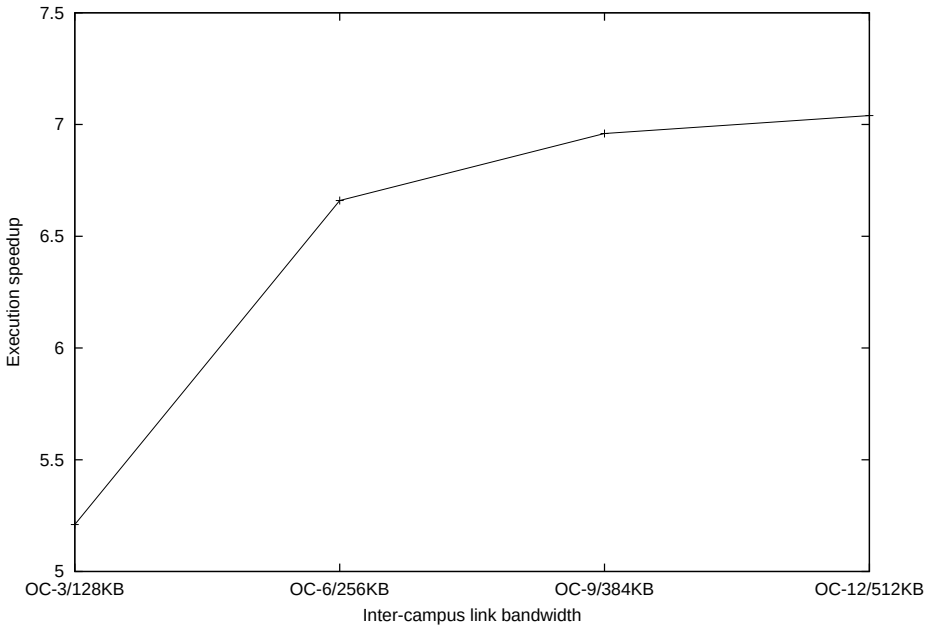

Figure 17: Speedup of Hybrid Model on Campus Network Topology

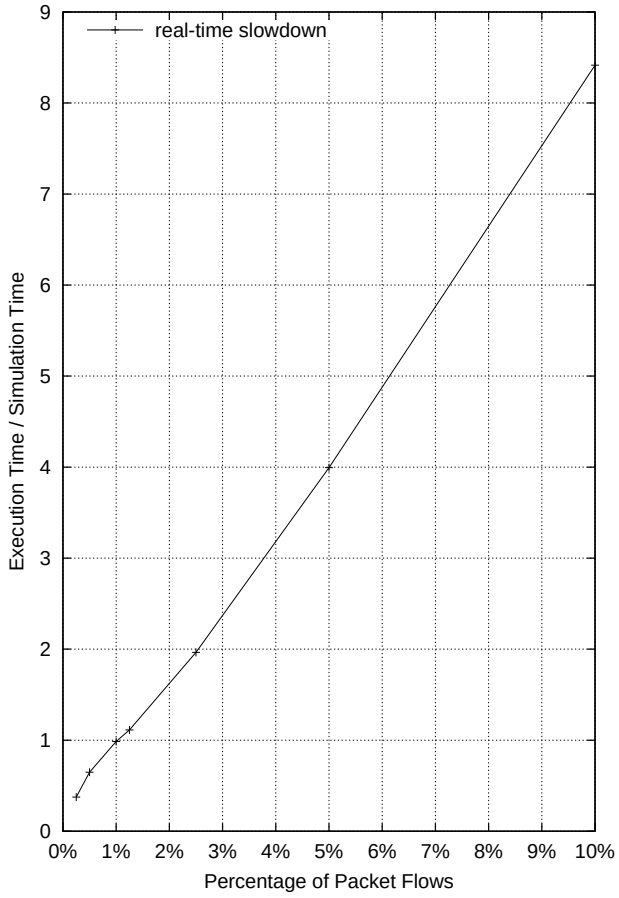

Figure 18: Real-time simulation of a campus network with 10,000 concurrent TCP flows. The native workload intensity is 170M state changes per simulated second; the simulation is run in parallel using 20 CPUs.

[3] D.Nicol. Discrete-event fluid modeling of tcp. In *Proceedings of the 2001 Winter Simulation Conference*, Arlington, VA, December 2001.

[4] Y. Guo, W. Gong, and D. Towsley. Time-stepped hybrid simulation (tshs) for large scale networks. In *Proceedings of IEEE INFOCOM 2000*, pages 441–450, Tel Aviv, Israel, March 2000.

[5] G. Kesidis, A. Singh, D. Cheung, and W. Kwok. Feasibility of fluid event-driven simulation for atm networks. In *IEEE Globecom 1996*, Nov. 1996.

[6] George Kesidis and Jean C. Walrand. Quick simulation of ATM buffers with on-off multiclass markov fluid sources. *Modeling and Computer Simulation*, 3(3):269–276, 1993.

[7] K. Kumaran and D. Mitra. Performance and fluid simulations of a novel shared buffer management system. *ACM Transactions on Modeling and Computer Simulation*, 11(1):43–75, January 2001.

[8] B. Liu, Y. Guo, J. Kurose, D. Towsley, and W. Gong. Fluid simulation of large scale networks: Issues and tradeoffs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume IV, pages 2136–2142, 1999.

[9] Benyuan Liu, Daniel R. Figueiredo, Yang Guo, James F. Kurose, and Donald F. Towsley. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *INFOCOM*, pages 1244–1253, 2001.

[10] Y. Liu, F. Lo Presti, V. Misra, D. Towsley, and Y. Gu. Fluid models and solutions for large-scale ip networks. In *Proceedings of the 2003 Sigmetrics Conference*, San Diego, CA, June 2003. To appear.

[11] V. Misra, W. Gong, and D. Towsley. Stochastic differential equation modeling and analysis of tcp-windowsize behavior. In *Proceedings of PERFORMANCE99*, Istanbul, Turkey, 1999.

[12] D. Nicol, M. Goldsby, and M. Johnson. Fluid-based simulation of communication networks using ssf, Oct. 1999.

[13] T. Ott, J. Kemperman, and M. Mathis. The stationary behavior of ideal tcp congestion avoidance. Technical report, Bellcore, 1996. ftp://ftp.bellcore.com/pub/tjo/TCPwindow.ps.

[14] A. Ridder. Fast simulation of markov fluid models. *Journal of Applied Probability*, 33(3):786–804, 1996.

[15] A. Yan and W. Gong. Fluid simulation for high speed networks. *IEEE Trans. on Information Theory*, June 1999.

[16] T. Yung, J. Martin, M. Takai, and R. Bagrodia. Integration of fluidbased analytical model with packet-level simulation for analysis of computer networks. In *Proceedings of SPIE*, 2001.