Proceedings of the 2001 Winter Simulation Conference
B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, eds.

DISCRETE EVENT FLUID MODELING OF TCP

David M. Nicol

Department of Computer Science
Dartmouth College
Hanover, NH 03755 U.S.A.

ABSTRACT

The bulk of Internet traffic is carried using variants of the TCP protocol. A realistic simulation-based performance study of any distributed application run over the Internet (e.g. reliable multicast) must therefore account for the impact that TCP background traffic has upon application behavior. Because TCP flows are shaped by other TCP flows, it is difficult to model TCP and its impact on other traffic other than by explicitly simulating it. This adds a significant computational burden to the simulation. This paper describes how we use fluid-based models of TCP to reduce the computational workload of simulating background TCP traffic. In particular we describe how a number of significant aspects of TCP can be described within a fluid formulation, how fluid models give rise to specific challenges that must be addressed by modeler and simulation kernel, and how we have addressed these in the DaSSF simulator.

1 INTRODUCTION

It is impossible to under-estimate the importance of TCP to the Internet, or the effect it has on the behavior of Internet traffic. Numerous studies have been done to tune and optimize TCP traffic. One cannot realistically consider how an application distributed over the Internet will behave without accounting for the effects that TCP has on it, both indirectly as its traffic interacts with TCP shaped traffic in the network, and (if the application uses TCP itself) directly.

TCP is a moderately complex algorithm. Under TCP, the injection of an application's traffic is dynamically governed as a function of how much buffer space its intended receiver has available, the estimated round-trip delay of a packet from sender to receiver and back, and effects of congestion in the network as reflected in time-outs and lost packets.

The most straightforward way to model TCP is to do segment level simulations of individual TCP flows (a segment is TCP's "packet" which holds both TCP header information and data, the maximum size of which is typically between 1500 and 500 bytes,) because TCP's behavior is described in terms of its action on segments. However, this is a computationally costly solution if we are interested in the behavior of an application, and need only to capture the effects of TCP on its traffic. For instance, if we are interested in the behavior of a multicast session that involves one hundred multicast entities, and expect that each link involved will have 100 independent flows (only one of which will be the flow of interest), then on the order of ten thousand separate TCP flows have to be simulated. In this case, to a first approximation, only one percent of the simulation effort directly involves the multicast!

This paper develops a fluid-based simulation model of TCP. The intuition is that we describe traffic flow in terms of *rate functions*. At any point in time, at any place in the network, the traffic flow of a given TCP sesssion is described in terms of the rate at which its traffic flows there, in bytes-per-unit-time. The rate functions we consider are piece-wise constant in time. Computational work occurs in the simulation only at instants when a flow's rate changes. The degree to which this represents a savings over a segment-oriented approach depends, in part, on the number of "equivalent" segment events that one rate change represents—estimated as the rate times the length of simulation time the flow rate remains constant, divided by segment size (in bytes). Obviously, the potential for computational savings is greatest when rates remain constant for significant periods of time. However there are hidden ramifications of fluid modeling that can significantly detract from this potential. Nor should it be forgotten that the computational grain of an event in a fluid-model may be different from that of a segment-oriented model. Nevertheless, intuition tells us that potential exists; we are working to exploit and assess it.

The notion of using fluid to model communication traffic has been explored primarily in mathematical contexts. Its potential for performance gains in a simulator was recognized in (Kesidis, Singh, Cheung, and Kwok 1996) and have

been analyzed further in (Liu, Figueiredo, Guo, Kurose, and Towsley 2001). In (Nicol, Goldsby, and Johnson 1999) we show that under a wide range of conditions the mean packet latency estimated from a fluid model through a network of fluid-oriented switches is very close to that of an equivalent packet-oriented simulator, as is the mean fraction of successfully delivered packets.

Recently attention has turned to modeling TCP using a fluid simulation. Past work ((Yan and Gong 1999), (Bonald 1998) and (Kumaran and Mitra 1998)) does so through differential equations, whose solution describes average transient flow behavior than than specific sample paths. Because complex interactions between flows within a network will give rise for unpredictable boundary conditions for such methods, we consider here an approach that is less mathematically and more closely models individual sample paths. Our model includes slow start, congestion avoidance, timeouts, lost data, and the fast-retransmit mechanism. We accomplish this by developing a modeling framework in which both discrete events (like loss of a segment) and continuous activity (the uninterrupted flow of traffic for some period of time) can co-exist, with the continuous and discrete parts of the model explicitly affecting each other. This level of detail and interaction is generally not possible to capture in a mathematical model.

2 MODELING ASSUMPTIONS

The most important modeling assumption is that traffic can be thought of as a fluid. A simulator built around this premise is driven by events that describe changes in the average transmission rate of a traffic stream. Consider a small system comprised of some hosts, each connected to the same switch. A stream runs from a source host through the switch and to a destination host. The source initiates the stream with a message describing how often the source injects bytes into the stream. The message is scheduled to arrive at the switch, after a latency delay across the channel. This is the same latency delay as a segment would see. The switch model processes the message to compute the effects of a rate change in one of its input streams, may impose an additional latency delay internally, and then sends a message to the destination host, again after a latency period. Thus we see that a simulator of a fluid model exchanges messages about rate changes in very much the same way as would segments be exchanged, imposing the same latencies on their transmission. A stream can be thought of as a pipeline whose length is the sum of the latencies between elements, holding at various places in the pipeline messages (about rates) that are "moving" through it. It is important to remember though that the rate of movement through the pipeline is not the flow rate of the traffic being modeled—the former reflects latency while the latter reflects bandwidth. Later we will say much more about these messages and their content.

We view a TCP session in terms of two communicating *agents*. We assume that each agent communicates with an application that either provides it data, takes data from it, or both. We assume that the network route from one agent to another may involve multiple routers, but that this route does not change over the life of the session. The route need not be the same in both directions.

We describe an application's presentation of data to TCP with an *offered rate* function, and assume the application is inter-locked with TCP with respect to data buffers (e.g., it doesn't try to send data when TCP's buffers are full.) We assume that every data segment a given agent sends has the same size, although sizes may be heterogeneous among agents. This assumption allows both for an agent that sends data in bulk, an agent that merely sends TCP headers containing acknowledgment information in response to that data, and an agent that sometimes sends segments containing data, and sometimes send segments comprised merely of a header.

We do not explicitly model the three-way handshake that initiates a TCP session, although there is no fundamental reason we could not—we assume the behavior of interest is the long term transfer behavior and focus exclusively on that. Similarly, in the interests of simplification, we assume that an application consumes data received from TCP as soon as it is available (in order). In this spirit we assume that the receiver advertised window (space available to receive data, advertised by the potential recipient in the header of every segment it sends) is constant.

At any point along a route between a session's agents, we describe the traffic behavior of the session (in one direction) there with a "rate function" that is piece-wise constant in simulation time. The fluid model we explore is deterministic, in the following sense. If at a given physical location, between simulation time s and t ($s < t$), the flow is λ bytes per unit time, then we assume that between s and t, *precisely* $(t - s)\lambda$ bytes flow through that location. There is no variance in byte inter-arrival times while the rate is constant.

Our fluid model explicitly accounts for latencies across communication channels. If a flow is mapped across a channel whose latency is d, then normally the behavior of the flow at the output end of the channel is related to the behavior at the input by a simple translation in time: $\lambda^{(out)}(t) = \lambda^{(in)}(t - d)$.

We describe the instantaneous state of a flow, at a physical point, with four elements: *raw flow rate*, in bytes per unit simulation time, *delivered fraction* (which is unitless) specifying how much of the original transmission is being delivered at this point, the *data ratio* in data bytes represented per transmitted byte, and the *acknowledged data ratio* in data bytes acknowledged represented per transmitted byte.

A *rate change* describes a change in one or more elements of a flow description, specifies (possibility implicitly) a place along a flow's path where the change is effective, and specifies a time at which it first becomes effective. A model element (e.g. a router) can easily determine the volume of flow that has past it, or even the index of the byte currently flowing past it, given a history of rate changes. We also allow a rate change to carry with it an additional data element we call a "cork". The idea is that the cork is carried along positionally in the flow, containing information placed there by a network element, to be read by other network elements on the path as the cork flows by. We employ corks to signal a variety of things, such as the start of a retransmission, and whether a raw byte flow rate of zero was set at the source.

3 TCP PRIMER

We can only sketch important characteristics of TCP. for a full treatment see virtually any standard networking text.

TCP is a full duplex protocol. Traffic flows in both directions between agents, even if one of the directions contains only acknowledgments. TCP views *data* flow (excluding header bytes) in terms of a contiguous sequence of enumerated bytes. It refers to a segment by the index of its first data byte, the *sequence number* Among other things, the header contains the sequence number and the number of bytes in the data segment. The recipient can therefore compute the sequence number of the next segment it expects to receive. This very quantity is carried in the header's *acknowledgment number* field.

TCP is a sliding window flow control protocol. Each agent maintains variables `LBS` (last-byte-sent) and `LBA` (last-byte-acked). The former is the largest sequence number of any segment sent by the agent, the latter is the largest acknowledgment number seen by this agent in any segment from its partner. TCP dynamically computes the maximum value that the window (`LBS` − `LBA`) is allowed to have. The agent is permitted to send the next segment only when this difference is smaller than the permitted maximum.

An agent expects to receive segments in contiguous order. Under normal circumstances an agent receives the next segment it expects, and either explicitly sends a header (with no data segment) with increased acknowledgment number, or soon sends another data segment whose header contains that acknowledgment number. It is possible for the network to discard a segment or deliver segments out-of-order. When an agent receives a segment other than the one it expects, the *fast retransmit* scheme requires it to send an acknowledgment—but the acknowledgment number will be a duplicate of the last one sent. An agent that receives three duplicate acks in a row considers the segment so identified to be lost. An agent also detects lost segments by using a timer, and considering a segment to be lost if it detected to be unacknowledged after a certain (dynamic) timeout interval. However detected, an agent retransmits a segment it considers to have been lost.

We now return to the dynamics of the maximum allowed window size. Recalling that the window identifies the total number of sent-but-unacknowledged bytes, one rule is that the maximum allowed window size can never exceed the *advertised receive window*, called `adrw`, inserted into every segment's header. An agent should never send more data than the recipient has available. In this the window serves a flow control purpose, additional rules provide congestion control. Ideally, just before the application presents a new segment for transmission to a full window, an acknowledgment arrives that first opens the window and so permits the segment to be sent. If there is congestion in the network then the carrying capacity is smaller and so the window ought to be smaller.

Congestion control rules govern the behavior of the "congestion window" variable `cwnd` and the "slow-start threshold" `ssthresh`; the maximum value of `LBS` − `LBA` permitted is the minimum of `cwnd` and `adrw`. Updates to `cwnd` depend on whether congestion control is in "slow-start" or in "congestion avoidance" mode. In slow-start mode the protocol attempts to find the general magnitude of the proper window size. At initiation, `cwnd` is set to the size of one segment's data block, thereby allowing exactly one segment to be sent before the window is full. `ssthresh` is set to 65K. Then, for each segment that is acknowledged, `cwnd` is incremented by one data block size. The net effect is to effectively double the congestion window every round-trip delay. Consider–after one round-trip delay the acknowledgment for the first segment comes in. `cndw` is increased to two segments worth, and two segments are sent. After a round-trip-delay **two** more segments are acknowledged, `cwnd` increases to four segment's worth, and four segments are sent, and so on. Slow start mode ends when either `cwnd` reaches or exceeds `ssthresh`, or if (first) a segment is thought to be lost (through a time-out, or triple-duplicate acknowledgment). If a segment is lost, `ssthresh` is immediately reduced to half the value of `cwnd`. In both cases, "congestion avoidance" mode is entered. Variable `cwnd` grows much more slowly in this mode. An acknowledged segment causes `cwnd` to be increased by 1.0/`cwnd`. An agent transitions from congestion avoidance to slow-start mode in the event that that a segment is lost. This means that `ssthresh` is set to half the value of `cwnd`, `cwnd` is set to one (maximal) data block size, and `cwnd` growth rules are then governed by slow-start.

TCP must detect and retransmit lost segments. One detection mechanism is for a agent to remember the transmission time of each segment, then run a periodic timer whose firing checks whether unacknowledged segments have waited for longer than a timeout interval. If so, the segment

is immediately retransmitted. An agent also uses the "fast retransmit" mechanism. This works because a acknowledgment segment is sent for (nearly) every segment received. The acknowledgment number is the highest numbered *consecutive* byte received to date. Thus, if a segment is lost, subsequent segments received each trigger an acknowledgment segment that has the same acknowledgment number as the previous acknowledgment segment. Under fast retransmit, if an agent receives three consecutive segments with the same acknowledgment number, it infers that the segment following the last one acked is lost, and immediately retransmits it.

This sketch illustrates that the TCP specification is inherently discrete. Our challenge is to model its essential components with a fluid model.

4 FLUID MODELING OF TCP

The heart and soul of TCP is its control of the window. Whereas the TCP specification advances LBA and LBS by integers, a fluid formulation describes their advance with piece-wise linear functions of simulation time. These functions are derived from rate change descriptions generated and processed by the simulator, associated with a sending flow and an acknowledgment flow back from the recipient.

Formalizing what we described earlier, a *flow description* at simulation time t (at some implicitly understood location in the network) is a tuple $(\lambda_{byte}(t), p(t), \tau_{data}(t), \tau_{ack}(t), \phi(t))$ where $\lambda_{byte}(t)$ is the rate at which raw bytes are flowing in simulation time, $\tau_{data}(t)$ is the ratio of data bytes in the stream to transmitted bytes, and $\tau_{ack}(t)$ is the ratio of bytes acknowledged by the stream to transmitted bytes; we define both ratios to be zero if $\lambda_{byte}(t) = 0$. $\phi(t)$, the *flow position*, is the byte index of this position in the flow, set at the point of transmission and carried along in the flow description. $p(t)$ is the delivered fraction. Informally, the flow being received at time t corresponds to flow transmitted by the source at some time s (s being the time at which byte position $\phi(t)$ was transmitted). $t - s$ is the latency incurred by the transmitted flow. If we let $\lambda_{byte}(s)$ denote the rate at which the application offers data to TCP at time s, then $p(t) = \lambda_{byte}(t)/\lambda_{app}(s)$, the instantaneous fraction of flow transmitted at s, that is being received at time t.

We will find it useful to refer to the rates at which data bytes are delivered, and at which data bytes are being acknowledged, per unit simulation time. These rates are be $\lambda_{data}(t) = \tau_{data}(t) * \lambda_{byte}(t)$ and $\lambda_{ack}(t) = \tau_{ack}(t) * \lambda_{byte}(t)/p(t)$, respectively. (Our use of $p(t)$ here models the accumulating effect of acknowledgments, and will later be discussed in more detail.)

Because TCP is a full duplex algorithm, a segment carrying data may in its header carry an acknowledgment number for data the sender has itself received in its dual role as data recipient. For example, an implementation of HTTP that uses TCP may have data segments that carry another URL as data and an acknowledgment number for data segments *received*. We assume that each data-bearing segment transmitted by an agent has the same size, allowing for header-only segments that carry just acknowledgments. However, different agents may have different data segment sizes. Because we allow transmission of an arbitrary mixture of data-bearing and non-data-bearing segments, we support models of general applications using TCP, subject only to the assumption of constant data segment sizes in a given flow direction.

An agent's transmission rate is a function of the rate at which an application delivers bytes to TCP, the state of the maximum allowed window size, and the rate at which previously sent bytes are acknowledged. Thus the behavior of two flows define a window. To distinguish between outgoing and incoming flow descriptions at an agent, we use superscript (out) to denote the flow the agent transmits, and (in) to denote the flow it receives.

The model supposes that there are always two directional flows associated with a pair of agents, each agent serving as the source of one and destination of the other. Each flow defines a path through network devices. An agent alters the flow for which it is source by creating a time-stamped rate change and passes it to the next simulated network device on the path. The delivered fraction component of this rate change is always 1.0. The other components reflect the mixture of data and data acknowledgments that the agent is inserting into the stream, and the rate at which it is transmitting data.

An agent may end up retransmitting data, and may receive retransmitted data. An agent can calculate byte indices within a flow from a history (in simulation time) of rate changes, and a *reference point* (s, b) which asserts that at time s, byte index b passed this point. An agent needs reference points for both outgoing and incoming flows. When an agent retransmits, it attaches to the new rate change a cork containing the reset starting byte index, taking the current time and that index for its new reference point. When an agent receives a cork, it sets incoming flow's reference point to the time of receipt, and the the contained byte index. A flow's initial reference point is $(0, 0)$.

Given simulation time t, if a flow's reference point is (s, b), then the upper edge of the send window at time t is given by

$$\mathtt{LBS}(t) = b + \int_s^t \lambda_{data}^{(out)}(x)\, dx.$$

The integral is easy to evaluate on the fly because the rate function is piece-wise constant. The behavior of the lower

edge LBA() function is similar.

$$\texttt{LBA}(t) = b + \int_s^t \lambda_{ack}^{(in)}(x)\ dx.$$

The subsections to follow discuss diverse components of the TCP fluid model. Later in the paper we appeal to these descriptions to concisely document all the discrete events used by the simulation, and all the processing that occurs at each event.

4.1 Overview

While still lacking precise definitions, we can still get a sense of how a fluid model operates. In TCP the decision to transmit a segment depends on the relationship of $\texttt{LBS} - \texttt{LBA}$ to the maximum data window size $\min\{\texttt{adrw}, \texttt{cwnd}\}$. In the fluid formation, the data transmission rate at time t depends on the relationship of $\texttt{LBS}(t) - \texttt{LBA}(t)$ to $\min\{\texttt{adrw}, \texttt{cwnd}(t)\}$. To illustrate, consider Figure 1. At the beginning of the period depicted, $\lambda_{ack}^{(in)}(t) > \lambda_{data}^{(out)}(t)$ for $t < 2$. In this interval $\texttt{LBS}(t) - \texttt{LBA}(t)$ actually decreases as a function of t. At time 2 however the acknowledged byte rate diminishes so that $\texttt{LBA}(t)$ grows more slowly in t, and at time 4 the acknowledged byte rate drops to zero and $\texttt{LBA}(t)$ stops growing in t entirely. The value of $\texttt{LBS}(4) - \texttt{LBA}(4)$ is presumed here to be less than $\min\{\texttt{adrw}, \texttt{cwnd}(4)\}$, and so transmission can continue unabated until the window is full, at time 6, when $\texttt{LBS}(6) - \texttt{LBA}(6) = \min\{\texttt{adrw}, \texttt{cwnd}(6)\}$; at this point transmission stops. The acknowledged byte flow starts up again at time 8, which "opens" the window to allow transmissions, at a rate no larger than that at which the window is being opened.

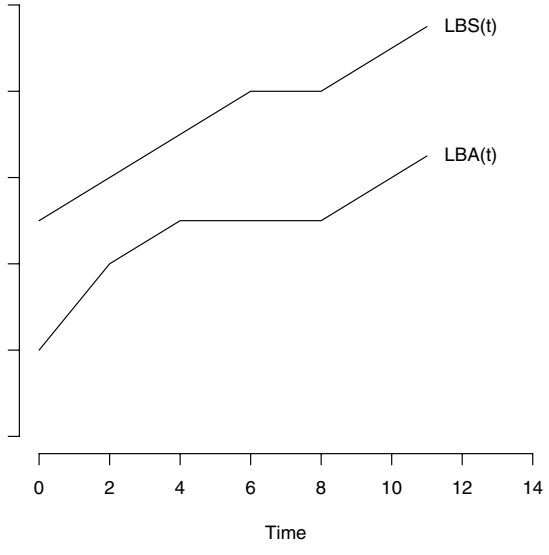

Figure 1: Sender window behavior in fluid formulation

Figure 1 also serves as a background for describing the nature of event processing in the fluid formulation. The event at time 2 is triggered by the arrival of a rate change on the acknowledged byte stream. Event processing recognizes that in the absence of further rate changes (of which it has no foreknowledge) $\texttt{LBS}(t) - \texttt{LBA}(t)$ grows linearly in t. This cannot continue ad infinitium for eventually the window limits will be reached. Using this function, the earliest time s such that $\texttt{LBS}(s) - \texttt{LBA}(s) = \min\{\texttt{adrw}, \texttt{cwnd}(s)\}$ is computed, and a "window-filled" event to deal with this occurrence is scheduled. However, the next event to occur happens before s, at time 4, when a rate change arrives reporting that the acknowledged byte ratio drops to zero (hence the flow rate drops to zero). The event at s is canceled, no longer being valid. $\texttt{LBS}(t) - \texttt{LBA}(t)$ is still linear in t, only now with different coefficients, and so we compute the intersection with maximal window size again and schedule a window filled event at 6. This event executes, because no other events intervene. It sets the transmission rate to zero (and generates a rate change), because the window is full, and not moving. This state can be changed only when the acknowledged byte flow starts again, which it does at time 8. Transmission can begin again, but at a rate that does not cause the window limits to be exceeded. The window remains in this state until either the sending application interrupts its data supply, or the acknowledged byte rate changes. It is in such a state that the fluid model achieves its greatest computational savings over segment-oriented models.

4.2 Modeling The Data Arrival Process

We assume that a large buffer N-byte buffer exists between TCP and an application using TCP to send data. The application writes data into it, and TCP imposes its send window pointers within it. The buffer is assumed to be at least as large as the maximum window size TCP allows. We let $S(t)$ denote the number of bytes buffered at time t, and $X(t)$ be the index of the last byte written into it, as measured at time t.

We model the behavior of the application with a offered load rate function $\lambda_{app}(x, f(t))$, where x is byte index, and $f(t)$ is a flag indicating whether the flow is on or off at time t. $\lambda_{app}(x, 0) = 0$ whereas $\lambda_{app}(x, 1) > 0$. The application and TCP are interlocked, in the sense that if the buffer is full at time t with greatest byte index x, then the application writes data into it at the minimum of $\lambda_{app}(x, f(t))$ and the rate at which TCP is emptying the buffer at time t. If the buffer is not full, the application fills it in accordance with the offered load function (assumed to be piecewise constant in x).

We also define function $\lambda_{src}(t)$ to describe the maximum rate at which data can be transmitted from the buffer, as a function of simulation time t. If $S(t) > 0$, $\lambda_{src}(t)$ is defined

to be outgoing data bandwidth (accounting for bandwidth consumed by non-data header bytes) ; if $S(t) = 0$ it is defined to be $\lambda_{app}(X(t), f(t))$, which simply means data cannot leave any faster than it enters. The application toggles the source on-off flag asynchronously from the TCP simulation. Notation $f(t)$ simply identifies the value of that toggle at t.

4.3 Modeling Maximum Window Size

One of the distinctive features of TCP is its congestion control mechanisms and the effect they have on traffic. We now describe how we can capture those features in a fluid framework.

Congestion control is reflected in the TCP variable `cwnd`, which we treat as a piecewise-linear function `cwnd`(t). Slow start and congestion avoidance modes are defined just as in TCP. We describe `cwnd`'s growth in slow-start with rate function $\lambda_{cwnd}(t) = \lambda_{ack}^{(in)}(t)$. This just says that for every acknowledged byte, `cwnd` increases by one byte. Whenever an event at time t is processed in slow-start mode, the change in `cnwd` since its last update is computed from the product of $\lambda_{cwnd}()$ just prior to time t, and the length of time since the last update.

Figure 2 illustrates the effect. Suppose that at bandwidth speeds transmission of a segment takes one unit of time, and that the round-trip delay is 20. `cwnd`(0) is initialized to one segment's worth of data bytes. The transmission stops at time 1, and the start of the acknowledgment flow is received at time 20, which turns the transmission flow back on. In this case we assume that data and acknowledgment flows are at bandwidth speed so bytes are acknowledged at the rate they were sent. The acknowledged bytes flow stops after one unit of time, but between times 20 and 21 $\lambda_{cwnd}(t)$ equals the acknowledged byte rate. Event processing at time 21 determines `cwnd`(21) by adding to `cwnd`(20) the integral of $\lambda_{cwnd}(t)$ over $[20, 21]$, yielding 2. It then recognizes that $\texttt{LBS}(21) - \texttt{LBA}(21) < \texttt{cwnd}(21)$ and schedules a "window filled" event to occur at 22. The window filled event stops the transmission because the acknowledgment rate is zero. `cwnd`(22) is updated using the same logic as was `cwnd`(21), but with no change effected as its rate of change is zero over $[21, 22]$. The same sequence is triggered at time 40 with the arrival of an acknowledged byte rate change. However, now the acknowledgment rate stays non-zero for 2 units of time because two segments are being acknowledged. At time 42, when the acknowledgments stop, `cndw`(42) has grown to reflect four segments, enabling the transmission to continue until time 44. This pattern continues every 20 units of time, each time doubling the volume of data sent out, just as does real TCP in slow-start mode.

We define the maximum allowed window size at time t as

$$\texttt{mxwnd}(t) = \min\{\texttt{adrw}, \texttt{cwnd}(t)\}, \qquad (1)$$

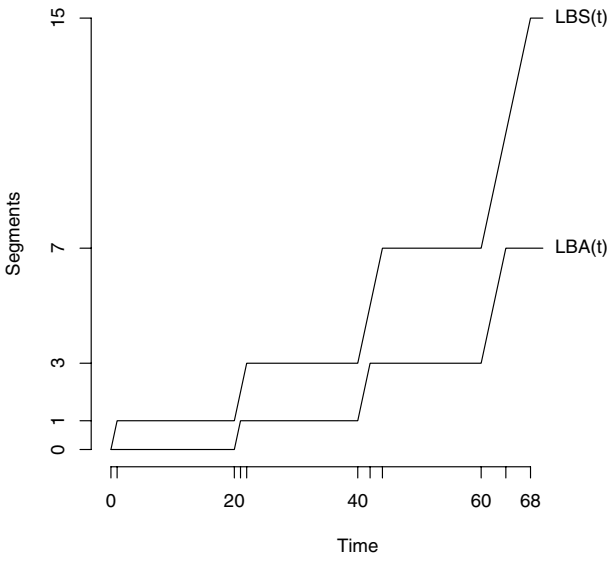

Figure 2: Behavior of Slow-Start in Fluid Model

and define its rate function as

$$\lambda_{mxwnd}(t) = \begin{cases} \lambda_{ack}^{(in)}(t) & \text{if } \texttt{cwnd}(t) < \texttt{adrw} \\ 0 & \text{if } \texttt{cwnd}(t) \geq \texttt{adrw} \end{cases} . \qquad (2)$$

The model logic has to account for the possibility of `cwnd`(t) exceeding `adrw`. When scheduling the next window filled event, two times are computed. One is the earliest time s at which $\texttt{LBS}(s) - \texttt{LBA}(s) = \texttt{adrw}$, given the present rates of flow, the other is the earliest time t when $\texttt{LBS}(t) - \texttt{LBA}(t) = \texttt{cwnd}(t)$, given the present rates of flow and growth in `cwnd`(). The earlier of s and t define the time of the event. It is possible for $s = t = \infty$, in which case the event is not scheduled.

In congestion avoidance mode `cwnd` grows by a segment's worth every time a full window of data is acknowledged. As `cwnd` grows, the window size grows, so the addition to `cwnd` becomes less and less frequent. Consequently the growth of `cwnd` is non-linear in the number of acknowledged bytes. It is important for us to model the instantaneous growth of `cnwd`, so we define $\lambda_{cwnd}(t)$ to *approximate* the real non-linear growth, with a piece-wise constant function. As the growth in `cwnd` is small in this regime, piecewise constant approximations are fairly accurate, and the constants can be chosen to change only after significant numbers of segments have been acknowledged.

4.4 Modeling Raw Transmission Rate

An agent maintains a flow description for its outgoing flow. The components of the flow description are a function of four binary conditions.

- *Source outflow* state : Either **on** ($\lambda_{app}(t) > 0$), or **off** ($\lambda_{app}(t) = 0$).
- *Acknowledged bytes inflow* state : Either **on** ($\lambda^{(in)}_{ack}(t) > 0$,) or **off** ($\lambda^{(in)}_{ack}(t) = 0$.)
- *Data inflow* state : Either **on** ($\lambda^{(in)}_{data}(t) > 0$,) or **off** ($\lambda^{(in)}_{data}(t) = 0$.)
- *Window* state : Either **open** ($\texttt{LBS}(t) - \texttt{LBA}(t) < \texttt{mxwnd}(t)$), or **saturated** ($\texttt{LBS}(t) - \texttt{LBA}(t) = \texttt{mxwnd}(t)$).

We will describe an agent's state as a vector of these, e.g., (on,on,off,open), with components in the order given above. It is a bit tedious to go through all sixteen possible states. We can however give the general principles used to implement transitions between vector states.

The first principle is that if state constraints force the data outflow ratio to be zero, (i.e. $\tau^{(out)}_{data}(t) = 0$) while the data inflow state is on, then the raw transmission rate is crafted to reflect sending of data-free headers that acknowledge the incoming data. The raw transmission rate is modeled as the header size divided by the time between successive segment arrivals :

$$\lambda^{(out)}_{byte}(t) = \frac{B_{hdr}}{(B^{(in)}_{data}/\lambda^{(in)}_{data}(t))} = \frac{B_{hdr}}{B^{(in)}_{data}}\lambda^{(in)}_{data}(t).$$

If on the other hand logical conditions call for positive outgoing flow of data, then any acknowledgment traffic is assumed to be piggy-backed onto the data segments. The raw transmission rate is entirely defined by other conditions, and we set $\tau^{(out)}_{data}(t) = B^{(out)}_{data}/B^{(out)}_{seg}$.

A second principle is that when the window state is saturated, the data transmission rate is constrained by the rate of received acknowledged bytes. Our model of raw transmission rate in these states captures the obvious coupling between receipt of acknowledgments, and transmission of new segments. If the window size were constant, then a saturated window would correspond to behavior where the TCP window is full, an acknowledgment comes in, releasing the transmission of another segment and the window is immediately full again. The intuition then is to define the data transmission rate to be equal to the acknowledged byte rate, and inflate the raw transmission rate to include header bytes. We amend the intuition to account for growth in the window size. In either of the congestion control modes, acknowledgment of one segment allows the transmission of strictly more than one segment—in slow-start it enables two, in congestion avoidance it enables one plus $1.0/\texttt{cwnd}$. The data transmission rate is designed to retain the equality $\texttt{LBS}(t) - \texttt{LBA}(t) = \texttt{mxwnd}(t)$ as t grows. The precise details of this depend on whether either of the window boundaries are moving at time t. If the acknowledged byte boundary is moving (e.g. $\lambda^{(in)}_{ack}(t) > 0$), we need

$$\lambda^{(out)}_{data}(t) = \left(\lambda^{(in)}_{ack}(t) + \lambda_{mxwnd}(t)\right),$$

so that we define $\lambda^{(out)}_{byte}(t) = (B^{(out)}_{seg}/B^{(out)}_{data})\lambda^{(out)}_{data}(t)$, and $\tau^{(out)}_{data}(t) = B^{(out)}_{data}/B^{(out)}_{seg}$. On the other hand, if the acknowledged byte boundary is not moving, neither can the sent byte boundary, so we require $\lambda^{(out)}_{data}(t) = 0$. The the raw transmission rate is zero, unless the first principle applies.

The third principle is that if the window is not saturated, then the data transmission rate is limited by the state of the buffer between application and TCP, and the application's own data production rate. We earlier defined $\lambda_{src}(t)$ to reflect maximum data transfer from the buffer, the raw transmission rate is scaled to account for header information.

4.5 Modeling Flow State

Our description of flows relies on the classification of an agent's incoming flow. First, the inflow is either *sullied* or unsullied. It is sullied at time t, if at some time s a rate change arrived with $p^{(in)}(s) < 1.0$ *and* $\tau^{(in)}_{data}(s) > 0$, and no "reset" cork has arrived between times s and t. To be sullied is to have lost data bytes and not yet have recovered from that.

In addition, at time t, each of the data and acknowledged byte components may be **on** (> 0) or **off** ($= 0$). This leads us to describe the state of a flow as a vector of three binary components : sullied state, data flow state, and acknowledged byte flow state.

4.6 Modeling Acknowledgments

The role acknowledgments play in TCP is paramount— they move the send window, and increase its size. We judge therefore that timeliness of response to received data is critical. Our fluid model acknowledges flow at the instant it is received—it might be thought of as acknowledging bytes rather than segments. If the receiving agent is transmitting data, we model piggy-backing the acknowledgment flow onto the data. If however the receiving agent is not transmitting data, we model transmission of data-free TCP headers carrying acknowledgment information.

In real TCP, any received segment carries acknowledgment information, even if the receiver has detected a "hole" in the incoming sequence of segments. We model this by accepting *transmitted* acknowledgments, not the received ones. The idea is that like real TCP, any acknowledgment that is received serves as well as previous acknowledgments that were not. To accomplish this we have defined $\lambda^{(in)}_{ack}(t) = \tau^{(in)}_{ack}(t) * \lambda^{(in)}_{byte}(t)/p(t)$ for $p(t) > 0$. The factor $p(t)$ inflates the rate to account for any data lost after

transmission. When $p(t) = 0$ then none of the sent data is being received and so in this case we define $\lambda_{ack}^{(in)}(t) = 0$.

We have already described how the outgoing acknowledgment flow affects the outgoing raw transmission rate; we need still to define component $\tau_{ack}^{(out)}(t)$. This is simply done. Since the acknowledged byte rate is identically the rate at which data bytes are received in an un-sullied flow, then we require that

$$\lambda_{byte}^{(out)}(t)\tau_{ack}^{(out)}(t) = \lambda_{byte}^{(in)}(t)\tau_{data}^{(in)}(t).$$

Re-writing, we have $\tau_{ack}^{(out)}(t) = (\lambda_{byte}^{(in)}(t)\tau_{data}^{(in)}(t))/\lambda_{byte}^{(out)}(t)$ for an un-sullied flow where $\lambda_{byte}^{(out)}(t) > 0$, and $\tau_{ack}^{(out)}(t) = 0$ when $\lambda_{byte}^{(out)}(t) = 0$ or the incoming flow is sullied at t.

These relations define the description of an agent's outgoing flow (with the exception of $p^{(out)}(t)$, which is always 1.0 leaving an agent) as a function of agent state.

4.7 Modeling Lost Traffic

Traffic can be lost in the interior of the network. In discrete TCP an agent can infer that one or more segments are lost (or out of order) by analyzing sequence numbers on segments it receives. We accomplish the same thing by maintaining the delivered fraction and flow position components in all rate changes. Suppose a flow with description ($\lambda_{byte}^{(in)}(t)$, $p^{(in)}(t)$, $\tau_{ack}^{(in)}(t)$, $\tau_{data}^{(in)}(t)$) enters a device where some of the flow is lost, e.g. due to buffer overflow. If bytes are lost at rate $\lambda_{loss}(t)$, then the flow description immediately past the point of loss is ($\lambda_{byte}^{(in)}(t) - \lambda_{loss}(t)$, $p^{(in)}(t) * (1.0 - \lambda_{loss}(t))/\lambda_{byte}^{(in)}(t))$, $\tau_{ack}^{(in)}(t)$, $\tau_{data}^{(in)}(t)$, $\phi(t)$). Here $\phi'(t)$ must be computed by the device model. If the device retained the last flow descriptor received before introduction of the loss, say at time s, it can compute $\phi(t)$ as

$$\phi(t) = \phi(s) + (t - s)\lambda_{byte}^{in}(s)/p^{in}(s).$$

These modifications allow a device receiving the flow description to see what the corresponding raw transmission rate was at the source, just by accessing the flow position.

An agent detects loss by examining the delivered fraction component of incoming rate changes. If a value less than 1.0 is observed at time t on a previously un-sullied flow, the agent sets $\tau_{ack}^{(out)}(t) = 0$ and generates an outgoing rate change. Its outgoing acknowledged byte ratio remains zero until the incoming flow is reset (implying that the lost data is being retransmitted). The eventual consequence of setting $\tau_{ack}^{(out)}(t) = 0$ is to set into motion TCP's mechanisms for detecting loss, and retransmitting.

4.8 Modeling Fast Retransmit

Under fast retransmit logic, when a TCP agent receives three successive headers containing the same acknowledgment index, it infers that the segment whose first byte is identically the acknowledgment index is lost, and engages in retransmission logic.

In our model an agent detects "retransmission" of acknowledgment indices at time t upon receiving a rate change with component $\tau_{ack}^{(in)}(t) = 0$. In response, if there is not already a retransmission event scheduled, one will be scheduled at t', which is the projected time by which three error-free segment headers will be received. The projection is based on the assumption that the $\lambda_{byte}^{(in)}$ and $\tau_{data}^{(in)}$ components of the incoming flow will not change. The event may have to be rescheduled if inflow characteristics change before t'. The rescheduling takes into account the volume of headers that have been received in error-free flow since the last rescheduling.

The three consecutive segments TCP uses to trigger fast retransmission need not be contiguous in the data stream—one or more segments may have been lost between them. Likewise, as we compute three segment's worth of headers before retransmitting, we skip over sullied flow and resume the accumulation when (and if) the delivered fraction returns to value 1.0.

Once a retransmission event is scheduled, it will eventually be executed.

4.9 Modeling Timeouts

TCP will retransmit a segment if an acknowledgment for it is not received within a certain period of time (which is dynamically computed as a function of measured round-trip delay times). Some variants of TCP implement the timeout mechanism at a fairly coarse level of granularity. Every 500 milliseconds a check is made for timed-out segments, and any are retransmitted at that instant. An approximation of this mechanism is easy to implement. When the timeout timer fires at t, we compute the time s at which byte byte $LBA(t)$ was transmitted, and if t exceeds s plus the timeout interval, the entire window is retransmitted. Finer resolution mechanisms are possible.

4.10 Modeling Retransmission

Execution of the retransmission event changes the outgoing flow description, and the congestion control state. We use the same rules as TCP to transition between slow start and congestion avoidance states, and to adjust values of `cwnd` and `ssthresh`.

Suppose the retransmission event executes at time t. All data bytes with indices between `LBA`(t) and `LBS`(t) are considered lost; the stream resumes at `LBA`(t), so we reset

$\texttt{LBS}(t) = \texttt{LBA}(t)$. All of these bytes will be retransmitted at bandwidth speed (since they reside already in memory). A rate change event is created, with raw byte and data ratio components for states (on,off,−,open). The acknowledged byte ratio does not change, its definition being orthogonal to the data being retransmitted. A "reset" cork containing byte index $\texttt{LBA}(t)$ is embedded in the rate change, and the whole message is sent to the next device in the path.

5 SUMMARY

Our approach to fluid modeling of TCP is a work-in-progress. This paper demonstrates the feasibility of modeling a rich variant of TCP when traffic flow is described by piece-wise constant rate functions. Significant tasks remain, including development of fluid models of switching and routing that do not suffer from the event explosion problem noted in (Nicol, Goldsby, and Johnson 1999), validation and verification under diverse system models, and analysis of the performance tradeoffs of this approach.

ACKNOWLEDGMENTS

This research is supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-98 08964, NSF Grant EIA-98-02068, NIJ contract 2000-DT-CX-K001, and Sandia National Laboratories under Department of Energy contract DE-AC04-94AL85000.

REFERENCES

Bonald, T. 1998, November. Comparison of tcp reno and tcp vegas via fluid approximation. Technical Report 3563, INRIA.

Kesidis, G., A. Singh, D. Cheung, and W. Kwok. 1996, November. Feasability of fluid-driven simulation for atm networks. In *Proceedings of IEEE GLOBECOMM*, Volume 3, 2013–2017.

Kumaran, K., and D. Mitra. 1998, March. Performance and fluid simulations of a novel shared buffer management system. In *Proceedings of IEEE INFOCOM 98*.

Liu, B., D. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. 2001, March. A study of networks simulation efficiency : Fluid simulation vs. packet-level simulation. In *Proceedings of the 2001 INFOCOM Conference*.

Nicol, D., M. Goldsby, and M. Johnson. 1999, October. Fluid-based simulation of communication networks using ssf. In *Proceedings of the 1999 European Simulation Symposium*. Erlangen, Germany.

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

AUTHOR BIOGRAPHY

DAVID M. NICOL is Professor and Chair of the Department of Computer Science at Dartmouth College. He is co-author of the text *Discrete Event Systems Simulation*, 3^{rd} edition (with Banks, Carson, and Nelson), and of over 120 technical articles. A frequent contributor to WSC for the last 15 years, he will be General Chair of the WSC in 2006. His research interests are in parallel simulation, networking, and computer security.