HNS: A Streamlined Hybrid Network Simulator

Benjamin Melamed
Rutgers University
Faculty of Management
Department of MSIS
94 Rockafeller Rd.
Piscataway, NJ 08854

Shuo Pan
Rutgers University
RUTCOR- Rutgers Center for Operations Research
640 Bartholomew Rd.
Piscataway, NJ 08854

Yorai Wardi
Georgia Tech
School of Electrical and Computer Engineering
Atlanta, GA 30332

Abstract

This paper is motivated by the need to speed up complex network simulation, especially in telecommunications settings, where high bandwidth translates into exorbitant numbers of packets that take inordinate CPU time to simulate. Since the simulation complexity of fluid workload is invariant under scaling bandwidth, flows of discrete units of workload may be replaced by (approximate) fluid streams for savings in CPU time and memory storage.

To this end, the paper outlines the design of a new hybrid simulator, called ***HNS*** (***Hybrid Network Simulator***). HNS simulates the movement of workload in a queueing network, where transactions may be of two types: traditional discrete transactions (e.g., packets) and continuous (fluid) transactions, all of which arrive discretely at the network in traffic flows, and each discrete arrival carries a workload. Arriving transactions only differ in the way their workload is transported: the user specifies whether the workload should be ***packetized*** or ***fluidized***, respectively, for transport across the network. The novel feature of HNS is that it admits models with both discrete and continuous traffic flows, and collects detailed statistics for both, including arrival, loss, buffer contents, departure, and delay statistics. HNS facilitates the fundamental trade off between the modeling accuracy of discrete flows and savings in simulation time and storage often afforded by continuous flows: it provides a common testbed for both discrete and continuous flows, where the user can readily select transport modes and achieve a good deal of variance reduction in assessing the accuracy and speeds of model versions with different mixtures of flow types. We caution, however, that while HNS is a generic hybrid simulator of discrete and continuous flows, mixed models should be handled with care, since they tend to counteract the advantages of pure-packet or pure-fluid models. In order to achieve a high degree of speedup for fluid-flow models, HNS introduces and utilizes the so-called ***streamlining*** methodology to identify and modify algorithms that cause "turbulent" fluid flow (namely, scenarios where fluid flow rates have a large number of minor fluctuations, which are computationally expensive but have a minor impact on simulation statistics). Streamlining removes these fluctuations so as to speed up the simulation at a moderate loss of statistical accuracy. HNS can be extended by writing additional classes; in particular, stream behavior can be modified by extending built-in protocol classes. For example, HNS already has dual implementations (packet and fluid) of various telecommunications-specific protocols, such as ATM, UDP and TCP, the latter with a streamlined fluid protocol approximation.

In this paper, we describe the architecture of HNS and its operational features, including traffic injection at network sources, generic workload transport, some telecommunications-specific protocols, and statistics collection and display via a graphic user interface (GUI). We then use HNS to validate the streamlining methodology and to study appropriate trade-offs of simulation speed and accuracy in telecommunications networks by comparing pure-packet, pure-fluid and mixed versions of the same network model.

Keywords: Fluid-Flow Models, Fluid-Flow Simulation, Fluid TCP Simulation, Hybrid Simulation, Mixed Models, Packet Models, Streamlining

1. Introduction

At a high-level of abstraction, queuing networks model the movement of transaction entities and their associated workload in a network of nodes. Nodes and links are stationary entities that can house buffers and servers representing network resources, while transactions are dynamic entities representing service demands in the form of the associated workload, which move among nodes and consume their resources. In an open network, transactions arrive at a network source as part of a traffic stream, and depart from the network at some network sink, where sources and sinks are entities associated with nodes, viewed as the network's "environment"; closed networks have no sources and sinks, and their initial population of transactions circulates in it forever. As workload is the most fundamental attribute of transactions, it is often convenient to conflate the two concepts and to use them interchangeably, context permitting; transactions, however, can have a myriad of additional attributes, such as priorities (to govern contention for buffer space and server processing resources), itineraries (deterministic or random paths through the network), etc.

From a modeling viewpoint, there are two variants of transactions, ***discrete*** and ***continuous***, depending on the nature and behavior of the associated workload. In the traditional queueing paradigm, transactions carry a discrete workload (e.g., models of jobs, packets, etc.), while in the stochastic fluid model (SFM) paradigm [see, e.g., Anick et al. (1982), Kobayashi and Ren (1992), Yan and Gong (1999)] workload is described in terms of a fluid metaphor (e.g., models of packet aggregates, bulk material, etc.) The differences lie in the way workload is stored, processed and moved around. Specifically, a discrete transaction moves from one node to another abruptly (discretely) on service completion, so that at any given time such a transaction resides in precisely one node; in contrast, a continuous transaction flows continuously like fluid from one node to another, and may reside in multiple nodes simultaneously. Note, however, that the notion of workload serves as the unifying concept for both transaction variants.

Monte Carlo simulation of traditional discrete-workload queueing networks has long been implemented within the discrete-event framework, primarily with variable clock increments for efficiency, while continuous-workload simulations have been implemented using constant clock increments [Bratley et al. (1987), Law and Kelton (1991)]. Recent work on fluid-flow simulation reflects renewed interest in fluid flow as a possible paradigm for speeding up telecommunications network simulation [Kesidis et al., (1996), Liu et al. (1999), Misra et al. (2000), Nicol (2001)]. We are not aware of simulation systems that have organically integrated discrete and continuous transactions in a queueing network setting within a discrete-event simulation framework with variable clock increments. HNS seems to be the first attempt in this direction.

From a high vantage point, Monte Carlo simulation of any type of flow can be viewed as workload processing, and its time complexity (CPU time expended) and space complexity (storage used) stem from events that handle ***workload fragmentation***. Workload transported as discrete transactions (***packets of workload***) represents a familiar form of fragmentation. Fluid transport gives rise to a less familiar form of fragmentation, where a continuous volume of fluid is fragmented into ***parcels of workload***, characterized by distinct flow rates. In the discrete-event fluid-flow paradigm, the simulator must keep track of such fluid parcels and their historical arrival rates in order to compute volumes of fluid mixtures in shared buffers; the relative proportions of such mixtures determine the effective departure rate resulting from a shared server. Thus, in either transport mode, events are engendered by workload fragmentation into packets or parcels, with the following properties:

1. Simulating the passage of a discrete transaction through a node typically calls for processing an incoming-transaction event and an outgoing-transaction event. Furthermore, the scope of state updating is typically limited to two nodes: the upstream node and the current node (for an incoming transaction), or the current node and the downstream node (for an outgoing transaction). Consequently, the time complexity of simulating a discrete-transaction stream grows linearly in its itinerary, and the traffic rate of transactions through a node. Space complexity is roughly proportional to the number of transactions in the network.
2. Simulating the passage of a continuous transaction through a node typically calls for processing ***flow-rate changes*** in the current node and the entire downstream sub-network, a phenomenon called the ***ripple effect*** (Liu et al., 1999). While that paper models propagation delay, HNS assume that rate changes propagate instantly (so the ripple effect is processed faster in a single event, rather than in multiple ones over time), and that the topology is feed-forward. Consequently, the time complexity of simulating a continuous-transaction stream grows quadraticly in its itinerary length (Liu et al., 1999). However, rate-change events are usually far less frequent than transaction-motion events, often by several orders of magnitude, and the workloads at nodes can be stored in scalar variables, rather than in elaborate transaction objects.

In summary, the time complexity tradeoff between network simulations with discrete and continuous flows, respectively, is that the former gives rise to more events than the latter, but events in the latter take more CPU time to process than in the former. Thus, one can imagine scenarios where the tradeoff between many computationally "cheap" events versus fewer computationally "expensive" ones would favor one variant or the other. The primary motivation for hybrid simulators stems from the differing (and largely complementary) scaling properties of discrete and continuous workload in terms of simulation complexity.

To illustrate the tradeoff more concretely, consider a telecommunications network, where nodes represent routers (or similar network elements), links represent transmission lines, transactions represent messages (e.g., requests and responses) that traverse some path, and workload represents data to be transported. The workload of a discrete message is packetized into a stream of packets (discrete workload), whereas the workload of a continuous transaction is fluidized and transported as fluid (continuous workload). Note that message identity is preserved in both variants. However, a fluid stream constitutes a cruder model of reality, being an approximation of a packet stream, where packets become mere "molecules" without individual identities. If we now scale the arrival rate and service rate of the workload simultaneously while keeping the relative traffic intensity constant, we see that the complexity of a discrete-flow simulation will increase concomitantly (there is a proportionately larger number of packets to process), but the time complexity of its continuous-flow counterpart will remain the same (the number of rate changes does not change)! Ceteris paribus, a continuous-flow simulation becomes more attractive as the bandwidth of the network and the offered traffic increase simultaneously, with the caveat that we must not trade off excessive modeling fidelity for simulation speedup. It becomes less attractive as the length of the paths grows and they traverse more network "real estate". In addition, the topology of all fluid flows should be feed-forward in order to simplify the re-computation of flow rates in downstream sub-networks, triggered by an upstream rate change.

It might appear that a mixed simulation model of discrete and continuous flows would be better than either of the pure-packet or pure-fluid versions alone, but unfortunately, this is ***not*** always the case. Keeping in mind that simulation complexity is driven by workload fragmentation, it should be apparent after some reflection that tossing numerous small packets into a pure-fluid

stream would "smash" the fluid workload into a correspondingly large number of fluid parcels, resulting in excessive fluid fragmentation. Worse still, since each packet embedded in fluid eventually translates into two rate changes at a server, this gives rise to a large number of computationally expensive fluid rate-change events. The end result is that mixtures of packets and fluid tend to nullify the speedup afforded by fluid alone. Thus, mixed models should be used with caution and the mixtures should be asymmetric: heavy on fluid streams and very thin on packet streams (often one packet stream at most). Fortunately, this guideline corresponds to the generic network model of Figure 1, dubbed the ***Target Traffic / Cross Traffic (TT/CT)*** model (also called the ***Foreground / Background*** model).

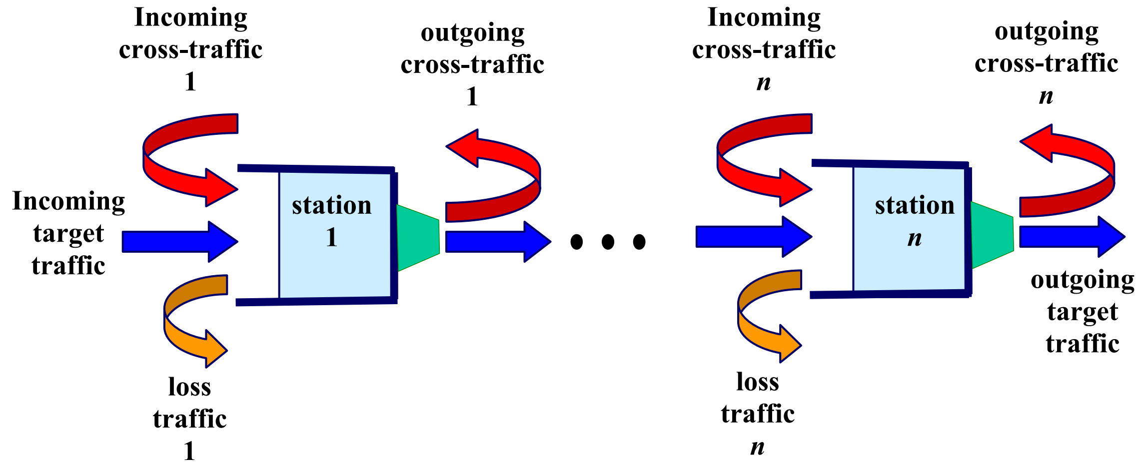

Figure 1. The generic TT/CT (Target Traffic / Cross-Traffic) network model

Continuing with the telecommunications network example, the target traffic might be a traffic stream of interest along some path, which needs to be modeled in detail to secure high modeling fidelity and predictive power. The target traffic would then be modeled as a packet stream (we may be interested in more than one such stream). In contrast, cross traffic streams are of secondary interest and are not required to give rise to accurate performance predictions. Their role is merely to capture all non-target traffic, which merely competes with target traffic for resources and reduces their availability. Since cross traffic is merely "interfering" traffic, which need not have high modeling fidelity, it can be modeled as fluid flows to take advantage of potential simulation speedups. More specifically, interfering traffic in feed-forward topology and of short span (relatively small downstream sub-networks) would be ideal candidates for continuous flows.

This paper consists of two main parts. The first part describes a novel hybrid discrete-event simulation tool, dubbed ***HNS*** (***Hybrid Network Simulator***), which admits both discrete and continuous flows in the same simulation model. It outlines the hybrid system worldview of HNS, its design and implementation. HNS is the sequel to an earlier version described in Melamed et al. (2001), where it is called HDCF-NS (Hybrid Discrete-Continuous Flow Network Simulator). The second part includes case studies of various telecommunications networks, which illustrate the tradeoffs realized by various assignments of flow types in TT/CT network setting.

The rest of the paper is organized as follows. Section 2 outlines the hybrid system worldview of HNS, and explains its streamlining methodology. Section 3 describes HNS, including its architecture and operation, and touches on its graphic user interface (GUI), and some implementation issues conducive to efficient runs. Section 4 discusses the implementation of

packet and streamlined fluid TCP in HNS. Section 5 studies the statistical accuracy and complexity of pure discrete, pure continuous and mixed network models, via three case studies. Finally, Section 6 contains the conclusions of the paper and outlines future directions for HNS, including the implementation of IPA (Infinitesimal Perturbation Analysis) gradients.

2. The Hybrid System Worldview

HNS adopts a hybrid system worldview that consists in the main of two fundamental component classes: a static ***network class***, and a dynamic ***transaction class***. Roughly speaking, the network component represents a layout of locations that house resources, while transactions are entities, which carry workload that moves among locations and consumes resources. The worldview is hybrid in that it accommodates seamlessly two flavors of transactions, discrete and continuous, with fundamentally different workload behaviors, to be explained below.

A ***network*** is a (fixed) directed graph consisting of ***nodes*** and ***links***, where a node represents a location in the graph and a link connects two nodes and houses a service facility (a shared server with a prescribed service speed and a shared buffer with a prescribed capacity) for processing (serving) ***workload*** and moving it on. The network interacts with an exogenous environment consisting of ***sources*** and ***sinks***, which are attached to nodes. A source is an ingress point for workload to enter the network, while a sink is an egress point for workload to drain from the network.

A ***transaction*** is an entity, whose fundamental attributes are the ***workload*** it brings to the network, its ***priority*** in contending for buffer and server resources, and its ***itinerary*** (path through the network starting in a source and terminating in a sink). Transactions come in two variants, ***discrete*** and ***continuous***, depending on the type of their workload and the way it moves in the network.

HNS transaction objects are called ***messages***, and these can be discrete or continuous. A message arrives discretely at an associated source, and its workload is injected into the network at the first node of its itinerary for transport through the network along an associated path. A source or a sink have precisely one message stream associated with each of them. While the HNS simulation engine knows how to "push" generic workload through nodes and links, additional rules that govern the movement of workload are implemented in ***protocol*** classes; normally, a protocol class has a dual implementation: a discrete version and a continuous counterpart that approximates the former. The transport of a message through the network depends on the associated workload type. A discrete workload is broken into packets, which are transported in the network in the usual discrete manner (see Section 1), subject to protocol rules. A continuous workload is transported as fluid (see Section 1), subject to protocol rules.

While the behavior of discrete flows is well known, that of continuous flows is based on a simple variant of the Continuous Flow Models (CFM) class [Wardi and Melamed (1999, 2000, 2001)]. A single-node CFM consists of a buffer that stores fluid, and a server that discharges fluid from the buffer. The fluid arrival rate is governed by a stochastic process $\{\alpha(t)\}$. Fluid is discharged by the server whose service rate is a stochastic process $\{\beta(t)\}$. The buffer capacity is generally a stochastic process $\{c(t)\}$, but in HNS it is a fixed quantity, c. It is assumed that the processes $\{\alpha(t)\}$ and $\{\beta(t)\}$ are only constrained to have ***piecewise-constant*** sample paths, but are otherwise quite general. This assumption is adopted in HNS, since it simplifies the integration of discrete and continuous flows into a common discrete-event framework.

Fluid flow simulations with propagation delays and topological loops (non feed-forward topologies) are subject to the drawback of ***echo effects***, where a rate change can trigger cascades of subsequent rate changes accompanied by network updates, which may not die down, thereby slowing simulation runs to a crawl. We mention that the ripple effect (Liu et al., 1999) is a special case of the echo effect. To avoid these problems, HNS does not model propagation delays and requires all fluid flows in the network to have a feed-forward topology; however, discrete flows are topologically unconstrained.

Flow control protocols, such as TCP, have an inherent feedback mechanism that couples a traffic flow from source to sink with an acknowledgement flow from sink to source. The resultant feedback effect is present even if the acknowledgment stream is virtual rather than actual, so as not to violate the aforementioned feed-forward restriction on the topology of fluid flow streams. Naïve implementations of fluid feedback effects result in computationally unpleasant phenomena in the simulator, which we term ***turbulence*** in keeping with the fluid-flow metaphor. The onset of turbulence is characterized by a temporary excessive fragmentation of fluid into tiny parcels, resulting from minor rate changes. The simulator computes the fluid-flow furiously, but the simulation clock is slowed to a crawl, and the computational advantage of fluid over packets is frittered away. In order to reclaim the computational advantage of fluid-flow simulation, we have introduced and implemented a stabilizing methodology, dubbed ***streamlining***, that removes turbulence and restores speedup at an acceptable cost in model accuracy by ignoring minor fluctuations (rate changes) in a fluid stream. For example, the flow control approximation of fluid-TCP ignores minor rate changes in an acknowledgement flow; roughly speaking, a change in the source injection rate takes effect only when a threshold is crossed. The details of fluid-TCP streamlining are given in Section 4.2.1.

3. The HNS Simulator

This section describes the architecture of HNS, including software layering, simulator operation, the graphic user interface (GUI), and optimization issues. HNS is implemented in Java to afford a seamless integration of the simulator engine back-end and its GUI front-end, as well as platform portability.

3.1 Outline of Architecture

HNS software is organized into five logical layers.

1. The **Network Layer** encapsulates input services for specifying simulation network-models, select statistics, and simulation control information. It then maps the input into an internal representation.
2. The **Transport Layer** encapsulates generic simulation services, such as initialization, and event execution of the underlying transport mechanism to “push” generic workload through the network, be it discrete or continuous.
3. The **Protocol Layer** implements built-in protocols logic. These are viewed as additional operational constraints on top of the generic transport mechanism implemented by the Transport Layer. Note that this layer is a natural candidate for extensions by expert programmers to accommodate new functionalities (e.g., experimental flow-control mechanism).

4. The **Statistics Layer** collects source and link statistics in a simulation run and computes summary model statistics within and across replications. It further collects simulator run statistics on elapsed CPU time, overall and per event class.
5. The **Presentation Layer** implements the GUI, including run control from the simulation console and display of dynamic statistics, in the course of a simulation run.

We now proceed to review each layer in more detail.

3.1.1 Network Layer

The network layer is responsible for constructing a network model from an input-file description. An HNS model input file has three parts: metadata (model name, storage unit and time unit), simulation control information (number of replications, simulation interval, warm up interval and initial seed for the random number generator), and network components.

Network components include nodes, links, sources and sinks (the last two model the network "environment"). While a node in HNS is just a geographical place in the network, links constitute network resources that process and transport workload through the network (e.g., transmission lines). Each link has a server with a prescribed service rate, and may or may not have a buffer (in the latter case the server can still accommodate packets or incoming fluid streams as long as it can cope with the arrival rate). We mention that bufferless links can be used to model optical transmission. Statistics collection is specified as part of node/link description. A network node can serve as an endpoint for any number of incoming and outgoing links. The service of a link is described by the stochastic process of its service rate, and the stochastic process of its rate duration.

Every message stream (discrete or continuous) has a priority and an itinerary, both inherited from the associated source. A stream is described by two stochastic processes: one for the inter-arrival interval process (iid or autocorrelated) and the other for the workload magnitude process (usually iid). Messages are queued at a source, but once they reach the head of the queue, they are injected into the network at the first node of their itinerary according to some rate process consisting of magnitude and duration (e.g., iid magnitude and on-off durations). The injection process is described in Section 3.1.2. Workload exits the network at a sink (the last node of a stream's itinerary). The topology of the network is implied by the itineraries of all streams. HNS performs consistency checks on network descriptions, and verifies that the topology of fluid streams is feed-forward.

3.1.2 Transport Layer

The transport layer is responsible for transporting generic workload (discrete or continuous) in the course of a simulation run, independently of stream protocols (if any). It handles message arrivals in each message stream, injects their workload into the network (as packets or fluid, depending on workload type), processes message workload at link servers according to their priorities, and routes the flows according to their itineraries until they drain out of the network. In other words, this layer "pushes" workload on in the network, from sources to sinks.

3.1.3 Protocol Layer

The protocol layer is designed to customize and extend the behavior of traffic streams beyond their generic behavior. Thus, protocols represent additional constraints on workload transport, implemented on top of the generic flow behavior at the transport layer (e.g., retransmission of lost

workload, admission control, flow control and congestion control). Since the main purpose of HNS is to facilitate the simulation of telecommunications networks, it already includes a protocol suite consisting of ATM, UDP and TCP as built-in protocols (see Section 4 for more details).

Each implemented protocol has two variants: a discrete variant and a continuous variant. Typically, discrete protocols are intended to model faithfully real-world protocols, while the continuous ones are approximate versions. Recall that this layer is extensible by expert programmers in that new protocols may be coded and added to the HNS protocol suite.

3.1.4 Statistics Layer

HNS collects statistics of two broad classes: user-specified model statistics and built-in simulator run statistics. The first class includes model-specific time series and summary statistics, while the second includes only summary statistics that may be useful in gauging model efficiency. HNS allows the user to run multiple replications and to collect user-specified model statistics and their summaries within each replication, as well as the grand summary statistics across replications.

User specified model statistics fall into two types: source statistics and link statistics. To improve simulation performance, the user is allowed to pick and choose the collection of statistics of interest by specifying them in the model input file.

Source statistics are stream-oriented and include message sojourn time and lost workload per message in the stream associated with a selected source. Each observation is collected when a message departs from the network, yielding a discrete time series. Link statistics are network-oriented and include the inflow rate, loss rate, buffer workload and outflow rate. These statistics are collected continually over the simulation interval, yielding continuous stochastic processes.

For each statistic selected, HNS collects the corresponding time series, computes its histogram, and a set of summary sample statistics, including the minimum, maximum, mean (time average in the continuous case), variance, standard deviation and coefficient of variation. All user-selected statistics are computed on the fly, and can be dynamically displayed in the course of a simulation run.

Simulator run statistics are collected for the entire simulation run (across replications) and include the total CPU time of the simulation run, number of events executed for each class of simulation event, and the CPU time spent in executing each class of simulation event.

3.1.5 Presentation Layer

The presentation layer is a graphic user interface (GUI) implemented in Java Swing classes. It is used for interactive simulation control and display of statistical output. This output is also saved in an HTML file. A brief description of the GUI may be found in Section 3.3.

3.2 HNS Operation

HNS events can be classified into three types:

- **Workload transport events** implement the generic transport of all traffic streams in the course of a simulation run.
- **Protocol events** implement protocol-specific behavior that refines the generic transport of workload.

- **Simulation control events** execute non model-specific events, such as resetting statistics and ending the simulation run.

We now proceed to describe the operation of the HNS back-end in the course of a typical simulation run. The operation consists of two stages: the preprocessing stage, and the simulation run stage, including statistics collection.

3.2.1 The Preprocessing Stage

After an HNS model file is read and parsed, HNS first validates the model. The validation process includes checking if the network configuration is feed-forward for continuous flows. The validation process also checks for network model consistency, e.g., detecting undefined nodes, links or sources.

Next, the HNS simulator initializes all network components and constructs a ***fluid network update list*** for each link. This list consists of all downstream links that can be affected by rate changes triggered by continuous streams in a given link. Since the topology of continuous streams is feed-forward, which precludes loops, all network paths that carry continuous streams can be sorted in topological order as follows: if a continuous stream flows through link A followed by link B, then link B is a downstream link with respect to link A, and is assigned a higher topological rank than link A. The update of downstream links will follow this topological ordering. For example, a network update often starts at a source, and the rate change propagates downstream as each is updated in turn. The preclusion of loops guarantees that the update procedure terminates.

In contrast, paths that carry only discrete streams have no topological constraints, so that a packet that leaves link A can be freely routed to link B, as specified in its itinerary. In the worst case, HNS has to update all downstream links of both link A and link B (that carry continuous flows). These two downstream sub-networks are combined so as to maintain the topological ordering in a combined list, called a ***packet routing update list***. For any given link, HNS constructs a packet routing update list for each adjacent link that carries packet flows, and stores them in a hash table.

Finally, HNS constructs a tree structure for the user-specified model statistics, so that they are readily accessible for dynamic display during a simulation run.

3.2.2 The Run Stage

The run stage consists in the main of workload transport and statistics collection. The workload transport mechanism in HNS operates differently at network sources and network interior (nodes and links). Starting with the network boundary, the source associated with a message stream queues up each incoming message in an arrival queue (using the traditional queueing paradigm). Once a message reaches the head of the queue, the source funnels that message workload to the first link on its itinerary in a process called ***injection***. Although injection is different for discrete and continuous messages, the notion of workload serves as the unifying element as follows. A discrete message is packetized (the packet size is specified in the input file), and the resultant packets attempt to enter the first itinerary link. If a packet cannot be accommodated in the (first) link's buffer, that packet is lost. The current dispatching interval (time between consecutive injections of packets) is determined as the ratio of the packet workload to the source's current injection rate. For a continuous message, the injection rate is simply determined as the current value of the source's injection rate process; again, the injected message may experience loss when the buffer at the first itinerary link is full. Thus, the injection rate represents the offered rate,

while the effective rate is the offered rate minus the loss rate. All in all, note that the injection rate is an invariant regardless of stream type. Finally, when injection is completed, the message is moved from the arrival queue to a linked list of messages pending transport completion. Later on, when its workload drains out of the network, its sojourn in the network is completed, and it will be discarded after all requested statistics are collected.

HNS manages discrete-workload events in the standard way, but the management of continuous workload is more complicated. Unlike the discrete case, continuous flows require not only the queueing-oriented tracking of fluid messages, but also a detailed tracking and bookkeeping of "historical" arrival rates at each buffer. To see why, observe that piecewise-constant rates of arrival and service of fluid imply that all flow rates in the network are piecewise-constant as well. Let the volume of a fluid inflow that arrives at some constant rate into a buffer be called a ***parcel***. Each link in the itinerary of a continuous stream has a so-called ***fluid manager*** attached to it, which is responsible for creating stream parcels. These managers have the same priority as the associated stream, and are organized in a list of ***manager priority hash sets***, where each hash set consists of all fluid manages of the same priority.

In a similar vein, let the vector of volumes of multiple inflows into a buffer, each with its own piecewise-constant inflow rate, be called a ***multiparcel***, and the period of simultaneous constancy be called a ***duration***. Clearly, a multiparcel is a mixture of individual parcel volumes, which are proportional to their inflow rates over the associated duration. Furthermore, if a constant service rate is applied to a multiparcel, individual constituent parcels will be depleted proportionately to the "historical" (constant) inflow rates, and similarly for the individual parcels' outflow rates. Whenever a rate change occurs in some inflow stream, a new multiparcel is started and evolves as a different mixture (with fixed proportions of parcel volumes). HNS maintains parcel objects, which record their flow affiliations and inflow rates. It maintains multiparcel objects as lists of parcel objects, where each multiparcel tracks the begin-time and end-time of its duration. The buffer workload associated with a multiparcel object is inferred for each constituent flow from the current time (simulation clock), the duration's begin time and end time, and the prevailing service rate allocated to the multiparcel (recall that this service rate is shared among all its constituent flows proportionally to their "historical" inflow rates and coincides with the respective outflow rates). We mention that HNS views a packet as a special case of a parcel with zero duration, so that the parcel concept unifies discrete and continuous workload. However, a multiparcel does ***not*** mix discrete and continuous workloads: it can either consist of fluid parcels or a single packet.

HNS employs a flow management scheme, called ***parceling***, which is at the heart of buffer management and workload transport. As the name suggests, parceling organizes incoming flows into parcels and multiparcels, queues them up for FIFO service within priority classes, and while being served, arranges to discharge each multiparcel constituent outflow proportionally to its original inflow rate. To this end, HNS maintains a 2-dimensional linked list, called ***multiparcel priority queue list***, as a variable-size matrix of multiparcels, whose rows correspond to priority queues (sets of flows with the same priority), and each queue consists of multiparcels in the chronological order of their creation. This data structure is shown in Figure 2, with incoming and outgoing streams, including loss streams.

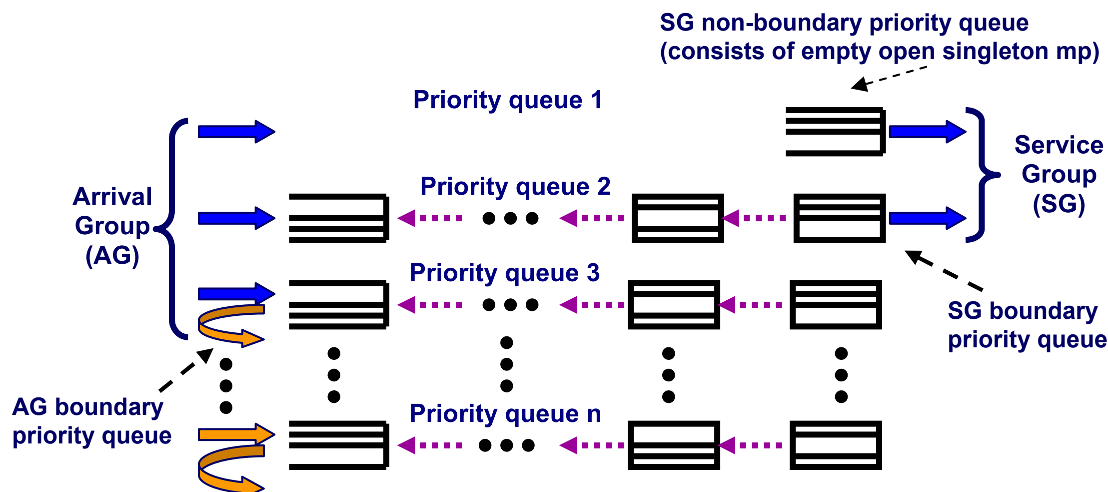

Figure 2. Structure of the multiparcel priority queue list

The multiparcel priority queue list consists of two kinds of (logical) multiparcels: *open* and *closed*. An open multiparcel captures the current (open-ended) multiparcel in progress at the tail of the queue, and as such it records a begin-time but no end-time. As soon as the flow rate of one of its constituent stream changes, the multiparcel gets "closed" (in the sense that the current simulation clock value is assigned as its end-time value), and a new (open) multiparcel is "opened" (in the sense that a new multiparcel is created with an updated vector of parcels, and the current simulation clock value is assigned as its begin-time value). The arrival of a packet at a buffer is treated as a flow-rate change. Note that the creation of a new flow is also viewed as a flow-rate change, and if there are no prior flows through a buffer, a singleton open multiparcel is created in the multiparcel matrix. Conversely, when a multiparcel's workload is depleted, that multiparcel is removed from the matrix and discarded.

For any buffer, the associated ***arrival group (AG)*** is the set of incoming streams that either completely or partially enter the buffer. As long as the buffer is not full, all streams can enter the buffer with no losses, and consequently, they all belong to the AG. When a buffer is full, the buffer can accommodate incoming streams commensurate with its service rate. Consequently, there exists an ***AG boundary priority queue***, such that

- All streams in this queue are either fully admitted into the buffer or suffer partial loss.
- All streams in queues with strictly higher priority can be fully admitted into the buffer.
- All streams in queues with strictly lower priority are fully lost.

Note that only the multiparcels at the head of each priority queue are eligible for service, since they represent the "oldest" workload in the buffer. For any buffer, the associated ***service group (SG)*** is the set of streams that are currently being served (have positive outflow rates). A multiparcel can be empty (no workload is present in the buffer, though fluid workload may pass through it) or non-empty. When a non-empty multiparcel is served, it absorbs all the remaining service rate. Consequently, there exists a ***SG boundary priority queue***, such that

- The first multiparcel in this queue absorbs all available service rate. It may be either open (in which case the buffer workload is not decreasing) or closed (in which case the buffer workload is not increasing).
- All streams with strictly higher priority necessarily contribute only an empty open singleton multiparcel.
- All streams with strictly lower priority are allocated zero service rates.

The following observations are straightforward consequences of the parceling management scheme:

a) An open multiparcel may contain fluid or may be empty, but a closed multiparcel cannot be empty.
b) Fluid parcels in a multiparcel can be either open or closed, but a packet multiparcel must be closed and consist of a singleton packet.
c) If a priority queue of multiparcels is not empty, it consists of zero or more closed multiparcels, followed by a trailing multiparcel, which may be an open or closed fluid multiparcel, or a packet multiparcel.
d) If a priority queue is allocated a positive service rate, then necessarily all higher-priority queues (if any) consist each of a singleton open empty fluid multiparcel, but not a packet multiparcel.

In discrete queueing systems, buffer state and server state carry the same information in the sense that an empty buffer is equivalent to an idle server and a non-empty buffer is equivalent to a busy server. The situation in continuous queuing system is more complex. For example, the buffer may be empty, but the server may well be busy. Since server/buffer information is heavily used to process flow-related events, HNS maintains for each link the state of the associated (server, buffer) pair as follows:

- State ***IDLE-EMPTY*** corresponds to an idle server and empty buffer. Note that for standard queues this state coincides with the standard idle state. However, this is not the case for fluid-flow as explained in the next item.
- State ***BUSY-EMPTY*** corresponds to a busy server and empty buffer. This state prevails only in fluid-flow queues when the buffer is empty and the arrival rate does not exceed the service rate.
- State ***BUSY-PARTIAL*** corresponds to a busy server and a buffer that is neither empty nor full.
- State ***BUSY-FULL*** corresponds to a busy server and full buffer. If the arrival rate exceeds the service rate in this state, some flows will sustain loss.

A simulation run can have multiple replications, as specified in the model input file. To ensure that replications are statistically independent, HNS initializes each replication by resetting the state of the network, but without resetting the random number generator. When all replications are completed, HNS computes all grand summary statistics (across replications) and stops.

3.3 The HNS Graphic User Interface

HNS includes a GUI, implemented using Java Swing classes. The GUI has the usual assortment of menus at the top to load and save simulation models, and to create panels for statistical display. A simulation control panel at the bottom allows the user to run simulations in various modes, display and hide dynamic statistics in the course of a simulation run, and show model grand summary statistics and simulator run statistics when the simulation run is completed. Statistics are displayed in a central canvas between the menus and the simulation console (see Figure 3).

The HNS simulation console permits extensive interaction with simulation runs and their statistics. Its run modes include

- ***batch mode*** for maximal simulation speed with minimal interaction
- ***step mode*** to run designated time intervals or events and obtain a new snapshot with each button prompt

- ***auto mode*** is similar to step mode, except that no subsequent prompts are required.

A user-specified interval may be entered in a console data field to specify the time between snapshot refreshes of statistics. Other run controls are VCR-oriented (pause, reset, and abort). A separate navigation panel allows the user to view statistical panels dynamically, during the simulation run. Visual statistics available are time series and histograms, while textual statistics include replication summary statistics (see Section 3.1.4). After a simulation is completed, the user can rerun the current model (with a new seed for the built-in random number generator). Otherwise, the user may elect to click buttons to display simulation grand statistics (across replications) or simulator run statistics.

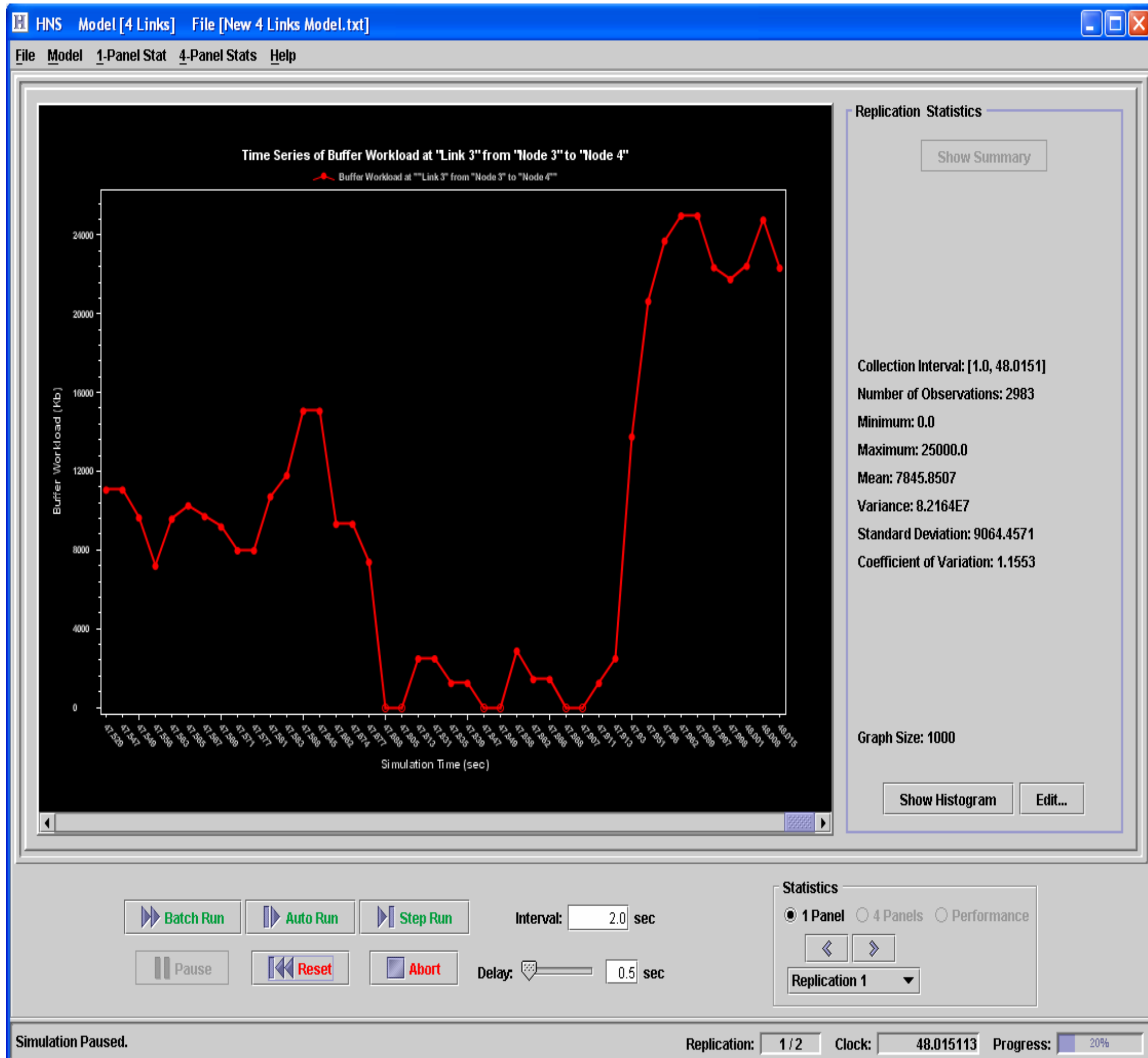

Figure 3. A snapshot of the HNS GUI

3.4 Efficiency Issues

Since one of the stated goals of HNS is to speed up discrete workload simulation, it is essential that efficiency issues play a central role in its design and implementation. We mention that unlike discrete-workload simulations, storage complexity is a relatively minor issue in

continuous-workload simulation, since the number of multiparcels in a pure-fluid version is typically much smaller than the number of packets in its pure-packet counterpart. Consequently, we discuss only speedup issues.

3.4.1 Event Reuse

To save execution time on instantiating new events, HNS strives to reuse event objects whenever possible. To this end, a number of event classes were identified, such that at any given (simulation) time, at most one instance of the event can exist in a specific part of the simulation. Examples of such events that are source-related include message arrival, injection-rate change and injection completion. Examples of events that are link-related include service-rate change, multiparcel service completion, and change of link state to BUSY-EMPTY or to BUSY-FULL. Accordingly, each source or link object has only a single instance of the associated event class as a member, all of which are created statically with their attributes during simulation initialization. When scheduling these events in the event list, only the event time needs to be updated. Furthermore, these events can be readily located in the corresponding source or link object (e.g., for removal from the event list or for rescheduling), thereby obviating time-costly searches of the event list.

3.4.2 Event coverage

We say that simulation event A ***covers*** simulation event B, if the processing tasks of event A include those of event B. Thus, the concept of event coverage is useful for analyzing task commonality in different event classes. When such coverage can be established for two event classes, it is possible to arrange for the common tasks to be performed exactly once, using the event priority attribute to schedule the covering event before the covered one. For example, when the link state changes from BUSY-PARTIAL to IDLE-EMPTY, then the multiparcel service-completion event covers the event that detects an empty buffer. The HNS implementation contains code fragments that check for event-coverage conditions that save execution time.

3.4.3 Topological Ordering of Network Links

Recall that a rate change at a network link triggers a sequence of subsequent link updates that cascade downstream of the former link following the network's feed-forward topology of continuous flows. The proper order of link updates is determined by the topology of the downstream portion, but is not unique. Furthermore, a transition of a packet from one link to another triggers rate changes in ***both*** links (origination and destination), followed by subsequent link updates downstream ***both*** of them. Since a feed-forward topology permits an overlap of the two downstream portions, one should be careful not to update a downstream node more than once.

HNS takes advantage of the static topology of simulated networks to determine and save the order of node updating as part of the preprocessing stage of a network simulation run (see Section 3.2.1). To this end, all network links are completely ordered (sequenced) in compliance with the partial ordering defined by link ***reachability*** via continuous flows. Using this sequence, a (static) subsequence of link updates is precomputed for each network link, and stored in the corresponding object. During a simulation run, each link experiencing a rate change simply uses its associated sequence to control the order of updates in downstream links, thereby eliminating redundant computations.

3.4.4 Selective Update

Updating the state of network objects (sources, links, statistics, etc.) is generally computationally expensive. Consequently, it is desirable to defer state updating whenever possible. HNS checks if such deferral is possible and performs only selective update of objects. To this end, these objects maintain a member that keeps track of the last update time.

4. Implementation of Fluid and Packet Protocols

This section describes the implementation of several built-in telecommunications protocols in HNS. Each protocol has a dual implementation: the standard packet-based implementation for discrete workload and a fluid-based implementation for continuous workload, to serve as an approximation for the purpose of simulation speedup.

4.1 UDP and ATM Implementation

The implementations of the UDP and ATM protocols are straightforward, and in fact differ very little from their generic implementation as packet or fluid streams. The workload of a UDP or ATM message is considered as payload. The packet implementation just adds the requisite header overhead when packetizing the workload (packet size is user-specified for UDP and predetermined at 53 bytes for ATM). The fluid version adds header overhead to the transported workload.

4.2 TCP Implementation

As TCP is a far more intricate protocol than UDP and ATM, its fluid implementation requires extensive approximation. Our version differs somewhat from other fluid approximations of TCP, such as Nicol (2001). In order to elucidate the fluid approximation, we will include a brief description of various features of the generic TCP implemented in HNS, and will explain how they map from the HNS packet version to an approximating HNS fluid version. These features are: congestion window management for flow control, stream transport, and loss retransmission.

4.2.1 Congestion Window Management

TCP is a sliding window flow control protocol, where each packet transmitted by the sender is acknowledged by the receiver. Whenever the sender sends a packet, it starts a timer, and if the timer expires for any reason before its acknowledgment is returned to the sender, a retransmission of that packet is initiated by the sender. The sender maintains the following information

- A congestion window (***cwnd***) variable for the total amount of workload the sender is allowed to send (initially set to one packet worth)
- A last-byte-sent (***LBS***) variable and a last-byte-acknowledged (***LBA***) variable, such that ***LBS* – *LBA*** does not exceed ***cwnd*** (but should come as close to ***cwnd*** as possible).
- An advertised receive window (***adrw***) is a prescribed constant, which cannot be exceeded by ***cwnd***.
- A so-called ***slow-start threshold*** is a variable, initially set to 65 KB.
- A so-called ***timer interval*** variable for acknowledgements to be returned to the sender.

The sender can alternate between two modes, subject to the following operational rules:

- Initially, the sender is in the ***slow start*** mode. In this mode, ***cwnd*** is increased by one packet worth for each packet acknowledged. Therefore it is doubled each time a full window worth of workload is acknowledged.
- When ***cwnd*** reaches the ***slow-start threshold***, the sender switches to the ***congestion avoidance*** mode. In this mode, ***cwnd*** is increased only by one packet worth when a full window worth of workload is acknowledged.
- The sender switches back to the ***slow start*** mode when retransmission of a packet is triggered by the firing of the associated timer.

The packet implementation of TCP in HNS includes all of the above features, but the fluid implementation is an approximation of the packet version. At each TCP source, a TCP protocol object is attached to manage TCP streams generated at the source. The fluid TCP protocol object maintains two data fields: ***fluid-cwnd*** (***fcwnd***) as a fluid counterpart of ***cwnd***, and ***fluid-unacknowledged-workload*** (***fuw***) as the fluid counterpart of ***LBS - LBA***, both in terms of workload. The sliding window flow control mechanism of TCP is mimicked by properly updating these two data fields.

For a fluid TCP stream, the ***fuw*** variable is updated using the (piecewise-constant) rates of workload entering and leaving the network (i.e., injection rate at the source and departure rate at the sink). HNS further keeps track of a virtual (fluid) acknowledgement stream, identical to the departure rate. We point out that actual modeling of acknowledgments is detrimental in fluid TCP simulation. First, fluid acknowledgement streams would violate the feed-forward restriction on fluid stream topology. On the other hand, packet acknowledgments would slow it down a great deal, thereby negating the declared purpose of fluid simulation. We note, however, that acknowledgments carry very little workload, so their omission would normally have a small effect on observed network statistics. Their only appreciable effect on performance occurs when acknowledgments are lost in a congested network (we assume error-free transmission). However, we are willing to trade their accuracy for simulation speed, especially when such streams are used to model cross traffic.

The ***fcwnd*** variable is updated as follows:

- In ***slow start*** mode, the growth rate of ***fcwnd*** is defined as the acknowledgement rate; note that since rates are piecewise-constant, the value of ***fcwnd*** is piecewise-linear. Consequently, the value of ***fcwnd*** is effectively doubled when a full window worth of workload is acknowledged.
- In ***congestion avoidance*** mode, the value of ***fcwnd*** is increased by one packet worth of workload, whenever a full window worth of workload is acknowledged.

HNS ***streamlines*** the injection rate at a source, using a prescribed threshold percentage parameter ***p*** (typically, ***p*** = 1%). The injection rate at the source is computed by the following algorithm.

- Initially, HNS considers the congestion window to be ***not filled*** (***fuw*** < ***fcwnd***).
- While TCP is in a period satisfying |***fcwnd* – *fuw***| / ***fcwnd* > *p***, then HNS considers the congestion window to be ***not filled*** in that period, and the source injects workload into the network at its full nominal rate.
- While TCP is in a period satisfying |***fcwnd* - *fuw***| / ***fcwnd* <= *p***, then HNS considers the congestion window to be ***filled*** in that period, and the effective injection rate depends on the sender's mode as follows:
  1) The first time TCP enters ***slow start*** mode in the current period, the effective injection rate is twice the acknowledged workload rate. While in this mode, subsequent

changes of the acknowledged workload rate do not change the injection rate in the current period.

2) The first time TCP enters ***congestion avoidance*** mode in the current period, the effective injection rate is the acknowledged workload rate. While in this mode, subsequent changes of the acknowledged workload rate do not change the injection rate in the current period.

Note carefully that HNS stabilizes the injection rate independently in each period. The injection rate changes when a new period is entered. Furthermore, streamlining in HNS permits the congestion window to be slightly overfilled (a situation that does not happen in the packet version), but this (abnormal) overfilling is fleeting and limited by the threshold parameter ***p***. In other words, the injection rate is forced to change when the window overfilling reaches its prescribed threshold.

4.2.2 Stream Transport

The implementation of packet TCP transport uses sink objects to send acknowledgment packets back to their associated sources. In fluid TCP transport, sinks are used to keep track of the rate of acknowledged workload to govern the fluid sliding window mechanism at the source. While the acknowledgment seems to violate the feed-forward requirement on fluid stream topology, the HNS algorithm to revise stream rates terminates in two passes. Fluid TCP transport along an itinerary is moderately more complex than the packet case. Additional processing is handled by a series of ***fluid TCP manager*** objects that reside at each link along the stream's itinerary. While no losses are experienced, these managers govern fluid transport as in the generic case. The behavior of the managers when loss is experienced is described in the next section.

4.2.3 Loss Retransmission

Retransmission of lost workload in TCP is triggered by timer expiration (timeout) at the sender. HNS detects lost packets and fluid, and deduces when and how much workload to retransmit. In HNS, timer behavior is implemented for each stream (packet or fluid) by a single separate retransmission event. Retransmission is scheduled on loss detection by different mechanisms for packet and fluid streams.

On packet loss, HNS computes the timer expiration time (via the residual ***timer interval***), and schedules retransmission of that packet at the computed expiration time. Fluid loss retransmission is more complex, and is mainly handled by the fluid TCP managers at the links. These managers are responsible for the creation of two additional variants of fluid parcels, specific to fluid TCP:

- ***lossy parcels*** are used to mark parcels that suffered some loss at the current or some upstream link.
- ***retransmit parcels*** are created by each TCP manager at every link in the message itinerary to mark the corresponding first parcel with retransmitted workload. When the retransmit parcel at the last itinerary link enters service, the departure rate of that parcel becomes the resumed rate of acknowledged workload.

A fluid sink manages retransmission by keeping track, for each message, of the total message workload acknowledged. It schedules retransmission whenever the rate of acknowledged workload drops to zero for any reason, or a lossy parcel enters service at the last itinerary link. Retransmission is then scheduled at the estimated timer expiration time, which is always half the ***timer interval*** later.

5. Experimentation with TT/CT Network Models

In this section, we describe some simulation experiments with telecommunications network models using HNS. The models are viewed as TT/CT models with designated foreground and background flows. Each model was run in three versions:

1. A pure-packet version, where all flows are discrete. This version is the most accurate and always serves as the reference (baseline) model.
2. A pure-fluid version, where all flows are continuous.
3. A mixed version, where the target traffic flows are discrete and the cross traffic flows are continuous.

For each version, a simulation run over the same time interval was made on the same computer (WINTEL desktop with a 2.66 GHz Pentium 4 CPU and 512 MB RAM). For each run, message sojourn time statistics were collected for comparison, as well as simulator run statistics, including simulation run complexity (number of simulation events and seconds of CPU time) and relative complexity in the form of the speedup factor with respect to the pure packet reference version (the ratio of the pure packet CPU time to the version CPU time).

5.1 A simple Telecommunications Network

The first model is a simple 5-node, 4-link network (Figure 4) with 5 sources (filled circles) of which 4 generate background UDP flows (dashed lines) and one TCP foreground flow (solid lines). Queueing nodes consisting of buffers and servers represent transmission links, and sinks are denoted by hollow circles.

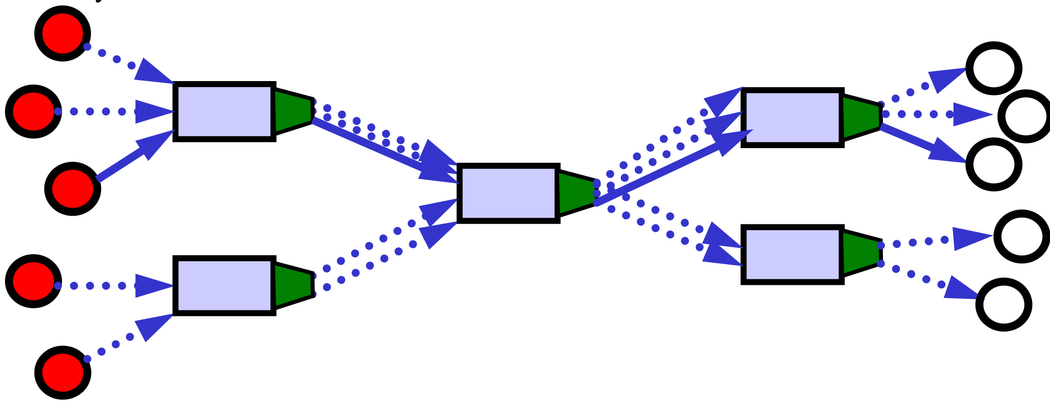

Figure 4. Simple telecommunications network model

Network model parameters are as follows:

- UDP message workloads are infinite (so only one message arrives at each UDP source).
- UDP sources inject their workload according to an on/off process with iid exponential durations of mean 0.05 second and an on rate of 655 Mbps.
- TCP message interarrivals are iid exponential with mean 1 second.
- TCP message workload is deterministic of 1 Mb each.
- The TCP source injection rate is deterministic at 655 Mbps.
- TCP packets are of size 8 Kb.
- All link rates are 1310 Mbps.

Network parameters were chosen so that the network is not congested and no traffic losses are incurred.

Table 1 displays sojourn time statistics over a simulation interval [10,130], for each model version.

Sojourn Time Statistics	Pure Packet	Pure Fluid	Mixed
Number of observations	114	114	114
minimum	0.0127	0.0127	0.0127
maximum	0.0887	0.0887	0.0887
mean	0.0203	0.0192	0.0202
variance	1.0 E-4	1.0 E-4	1.0 E-4

Table 1. Comparison of message sojourn time statistics for the simple network

Recalling that the pure packet version serves as a reference version for comparison with the pure fluid and mixed versions, Table 1 shows that the latter yield quite accurate statistics, though the mixed version yields a slightly more accurate mean. This is due to the fact that this network is not congested and no retransmission occurs.

Table 2 displays the simulator run statistics for each model version.

Simulator Run Statistics	Pure Packet	Pure Fluid	Mixed
Number of Events	64,734,354	14,328	1,178,758
CPU Time (seconds)	696.063	0.875	20.531
Speedup Factor	1	795.5	33.90

Table 2. Comparison of simulator run complexities for the simple network

In this case, the simulator run complexity varies dramatically among versions, even though all yield comparable statistical accuracy. As expected, the pure fluid model is the fastest, giving rise to the fewest number of events and a speedup factor of almost 3 orders of magnitude. The mixed version's complexity falls between that of the pure packet and pure fluid versions.

5.2 A Telecommunications Network with a Bottleneck Link

This network model consists of a target traffic path with interfering cross traffic through a common bottleneck link, as depicted in Figure 5.

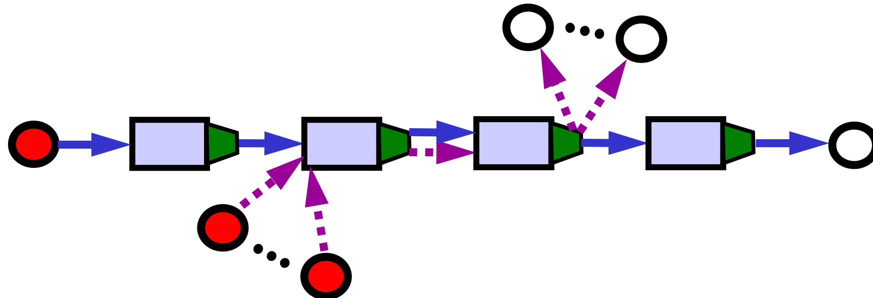

Figure 5. Telecommunications network model with a bottleneck link

The graphical objects in this model have the same meaning as in the previous one. Network flows consist of one target TCP stream and 100 UDP flows as cross traffic.

Network model parameters are as follows:

- UDP message workloads are infinite (so only one message arrives at each UDP source).
- UDP sources inject their workload according to an on/off process with iid exponential durations of mean 0.05 second and an on rate of 9.9 Mbps.
- TCP message interarrival time is iid exponential with mean 1 second.
- TCP message workload is deterministic of 8 Mb each.
- The TCP source has a constant injection rate of 9.9 Mbps.
- The bottleneck link rate is 500 Mbps and all others link rates are 10 Mbps.
- All packets are of size 8 Kb.

Table 3 displays sojourn time statistics over a simulation interval [10,130], for each model version.

Sojourn Time Statistics	Pure Packet	Pure Fluid	Mixed
Number of observations	128	126	128
minimum	0.0841	0.0842	0.0841
maximum	8.3335	9.2815	8.3335
mean	1.3237	1.9318	1.312
variance	3.2513	4.9076	3.2415

Table 3. Comparison of message sojourn time statistics for the bottleneck network

Here the pure fluid version yields higher sojourn time statistics than the pure packet (reference) version, but as expected, the mixed version's statistics are much closer to those in the pure packet version.

Table 4 displays the simulator run statistics for each model version.

Simulator Run Statistics	Pure Packet	Pure Fluid	Mixed
Number of Events	488,748,693	22,134	158,102
CPU Time	4,068.079	3.859	17.438
Speedup Factor	1	1054	233.3

Table 4. Comparison of simulator run complexities for the bottleneck network

Again, the pure fluid version yields a dramatic speedup of 3 orders of magnitudes, while the mixed version gives rise to a more moderate, though substantial, speedup. Taken together, Tables 3 and 4 illustrate the fundamental tradeoff between simulation speedup and simulation accuracy.

5.3 A Telecommunications Network with Target Traffic Across Clouds

This network model consists of a target traffic path traversing multiple clouds (subnetworks), as depicted in Figure 6.

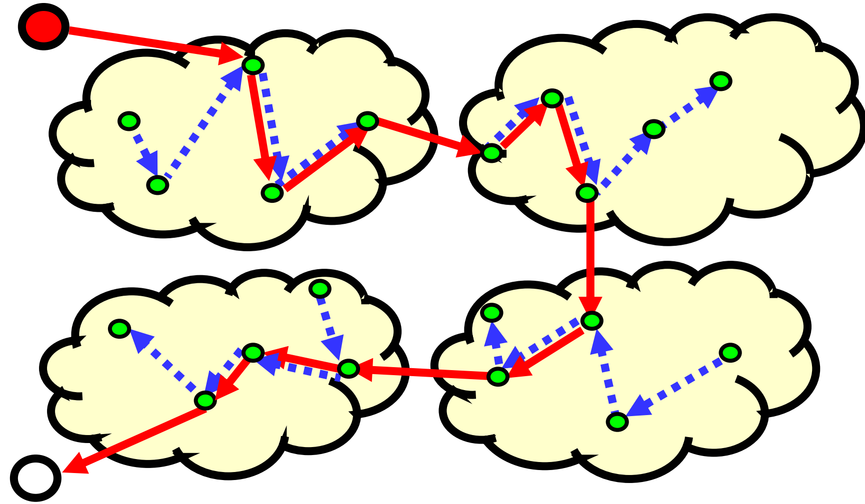

Figure 6. Telecommunications network model with target traffic across clouds

Here, the small circles in the clouds represent nodes and arrows represent transmission links: solid arrows trace the target-traffic path and dashed arrows are local-cloud cross traffic paths. The external large filled circle at top left represents the sources of a set of target streams, all of which follow the same target-traffic path, while the external large hollow circle at bottom left represents the corresponding set of sinks.

Network components are as follows:

- The network consists of 4 clouds, each with 32 nodes.
- Each network boundary node contains 10 TCP/UDP flows for a total of 80 flows per cloud.
- One target traffic path consisting of 10 TCP/UDP flows traverses the network.

Network model parameters are as follows:

- All message interarrival times are iid exponential with mean 1 second.
- UDP message workloads are infinite (so only one message arrives at each UDP source).
- TCP message workload is deterministic of 4 Mb each.
- All packets are of size 4 Kb.
- For each TCP source, the injection rate process is constant at 6.55 Mbps.
- For each UDP source, the injection rate process is on-off. The on rate is constant at 6.55 Mbps, and all durations are iid exponential of mean 1 second.
- All links have transmission rates of 100 Mbps.

Table 5 displays sojourn time statistics over a simulation interval [10,130], for each model version.

Sojourn Time Statistics	Pure Packet	Pure Fluid	Mixed
Number of observations	129	130	131
minimum	0.6626	0.6624	0.6624
maximum	5.7853	3.888	5.0892
mean	2.0567	1.4736	1.766
variance	1.2624	0.5545	0.8795

Table 5. Comparison of message sojourn time statistics for the four-cloud network

In this case, the pure fluid version yields lower sojourn time statistics than the pure packet (reference) version, and again, as expected, the mixed version's statistics are closer to their counterparts in the pure packet version.

Table 6 displays the simulator run statistics for each model version.

Simulator Run Statistics	Pure Packet	Pure Fluid	Mixed
Number of Events	400,616,679	179,655	24,239,178
CPU Time	3,668.282	6.907	2,630.0
Speedup Factor	1	531.1	1.395

Table 6. Comparison of simulator run complexities for the four-cloud network

As expected, the pure fluid version gives rise to a major speed up with respect to the pure packet version. However, the mixed version yields in this case only a minor speedup. The reason for this phenomenon is that the long target-traffic path causes a major fragmentation of fluid streams into a very large number of small fluid multiparcels. To make things worse, this fragmentation generates, in turn, a very large number of (expensive) fluid-based events. The result is that the simulation advantages of fluid workload are diminished to the point that the mixed version is nearly as slow as the pure packet version. In fact, our experience shows that when the number of target packet streams is increased further, the mixed version may well run ***more slowly*** than the pure packet version! Thus, the mixed version cannot always be relied upon to provide a reasonable tradeoff between simulation speed and accuracy. Rather, it is advisable to make a number of pilot runs to gauge the relative and absolute simulation speeds of all three versions: pure packet, pure fluid and mixed.

6. Conclusions and Future Directions

This paper described the design and implementation of a hybrid network simulator, dubbed HNS, which admits mixtures of packet and fluid streams within a unified worldview. HNS can be used to trade off simulation speed and accuracy by allocating judiciously the type of traffic flows (discrete or continuous). Stream type allocations can be based on pilot runs, especially for mixed simulation models. Such models must be handled with caution, since as evidenced by Section 5.3, mixed model versions can underperform pure packet versions in terms of both speed and accuracy.

HNS is extensible in the sense that additional protocols, extant or experimental, can be added as new Java classes. While current built-in HNS protocols belong to the telecommunications domain, HNS has a sufficiently abstract worldview to accommodate other domain areas, such as bulk material flow and handling.

We plan to enhance HNS in two directions. First, additional protocols can be added. New telecommunications protocols are natural candidates, for example, protocols for emerging optical telecommunications networks.

The second enhancement planned for HNS is statistical: the incorporation of IPA (Infinitesimal Perturbation Analysis) to which the fluid-flow setting is amenable; we mention in passing that IPA in packet setting is also feasible by translating packet workload into fluid workload and vice

versa. IPA is a technique for computing sample path derivatives (gradients) of performance metrics (random variables) with respect to parameters of interest (e.g., the derivative of the loss volume or time average of buffer contents as function of buffer size). Formally, the IPA derivative of a real-valued random variable $L(\theta)$, where θ is a parameter in a bounded interval, is the random variable $L'(\theta) = (\partial / \partial\theta) L(\theta)$, provided it exists. To be of practical value, it is necessary for $L'(\theta)$ to be ***unbiased*** [Ho and Cao (1991), Cassandras (1993)], and nonparametric in the sense that the formula for computing $L'(\theta)$ is independent of the underlying probability law. While this is usually not the case in traditional discrete queueing systems, recent work has shown that this is the case in fluid-flow setting [Miyoshi (1998), Wardi and B. Melamed (1999, 2000, 2001), Cassandras et al. (2002), Wardi et al. (2002)]. Furthermore, the computational algorithms for IPA derivatives are fast and simple, allowing them to be computed in real time from actual network observations and not only from simulation sample paths (histories). Finally, IPA derivatives can be computed in packet setting as well by adopting a fluid-oriented viewpoint to workload. Consequently, IPA derivatives could be used in stochastic optimization, and have potential applications to decision making on network design, replenishment, management and control. Their incorporation into HNS would provide a tool to assess their efficacy.

Acknowledgments

This work was supported in part by DARPA Agreement F30602-00-2-0556.

References

[1] D. Anick, D. Mitra, and M.M. Sondhi (1982) "Stochastic Theory of a Data-Handling System with Multiple Sources", *Bell System Technical Journal* **61**, 1871--1894.

[2] P. Bratley, B.L. Fox and L.E. Schrage (1987) *A Guide to Simulation*, Springer-Verlag.

[3] C.G. Cassandras (1993) Discrete *Event Systems: Modeling and Performance Analysis*, Aksen Associates Publishers, Irwin, Boston, MA.

[4] C.G. Cassandras, Y. Wardi, B. Melamed, G. Sun and C.G. Panayiotou (2002) "Perturbation Analysis for On-Line Control and Optimization of Stochastic Fluid Models" *IEEE Trans. On Automatic Contro*l, **AC-47**(**8**), 1234--1248.

[5] Y.C. Ho and X.R. Cao (1991) *Perturbation Analysis of Discrete Event Dynamic Systems*, Kluwer Academic Publishers, Boston, MA.

[6] G. Keisidis, A. Singh, D. Cheung, and W.W. Kwok (1996) "Feasibility of Fluid-Driven Simulation for ATM Network", *Proc. IEEE Globecom* **3**, 2013--2017.

[7] H. Kobayashi and Q. Ren (1992) "A Mathematical Theory for Transient Analysis of Communications Networks", *IEICE Trans. on Communications* **E75-B**, 1266--1276.

[8] A.M. Law and W.D. Kelton (1991) *Simulation Modeling \& Analysis*, (second edition), McGraw-Hill.

[9] B. Liu, Y. Guo, J. Kurose, D. Towsley, and W.B. Gong (1999) “Fluid Simulation of Large Scale Networks: Issues and Tradeoffs”, Proc. *Intl. Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada.

[10] B. Melamed, S. Pan and Y. Wardi (2001) “Hybrid Discrete-Continuous Fluid-Flow Simulation”, *Proc. of the SPIE International Symposium on Information Technologies and Communications (ITCOM 01), Scalability and Traffic Control in IP Networks*, 263--270, Denver, Colorado.

[11] V. Misra, W. Gong and D. Towsley (2000) “Fluid-Based Analysis of a Network of AQM Routers Supporting TCP Flows With an Application to RED”, *SIGCOMM ’00*, Stockholm, Sweden.

[12] N. Miyoshi (1998) “Sensitivity Estimation of the Cell-Delay in the Leaky Bucket Traffic Filter With Stationary Gradual Input”, *Proc. International Workshop of Discrete Event Systems (WoDES'98)*, 190--195, Cagliari, Italy.

[13] D.M. Nicol (2001) “Discrete Event Fluid Modeling of TCP”, *Winter Simulation Conference (WSC 01)*, Arlington, Virginia.

[14] Y. Wardi and B. Melamed (1999) “Continuous Flow Models: Modeling, Simulation and Continuity Properties”, *Proceedings of 38-th IEEE CDC*, 34--39, Phoenix, Arizona.

[15] Y. Wardi and B. Melamed (2000) “Loss Volume in Continuous Flow Models: Fast Simulation and Sensitivity Analysis”, *Proc. of 8-th IEEE Mediterranean Conference on Control and Automation (MED-2000)*, Patras, Greece.

[16] Y. Wardi and B. Melamed (2001) “Variational Bounds and Sensitivity Analysis of Continuous Flow Models”, *J. of Discrete Event Dynamic Systems* **11**(**3**), 249-282.

[17] Y. Wardi, B. Melamed, C.G. Cassandras and C.G. Panayiotou (2002) “On-Line IPA Gradient Estimators in Stochastic Continuous Fluid Models”, *J. of Optimization Theory and Applications* **115**(**2**), 369-405.

[18] M. Yoo, C. Qiao and S. Dixit (2001) “Optical Burst Switching for Service Differentiation in the Next-Generation Optical Internet” *IEEE Communications* **39**(**2**), 98--104.

[19] A. Yan and W.B. Gong (1999) “Fluid Simulation for High Speed Networks with Flow-Based Routing”, *IEEE Trans. on Information Theory* **45**, 1588-1599.