SANDIA REPORT

SAND2001-8086

Unlimited Release

Printed December 2000

Large-Scale Information Systems

D. M. Nicol, H. R. Ammerlahn, M. E. Goldsby, M. M. Johnson,
D. E. Rhodes, and A. S. Yoshimura

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of
Energy under Contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States
Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Prices available from (703) 605-6000
Web site: <http://www.ntis.gov/ordering.htm>

Available to the public from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

SAND2001-8086
Unlimited Release
Printed December 2000

RECEIVED
DEC 20 2000
OSTI

Large-Scale Information Systems

David M. Nicol
Department of Computer Science
Dartmouth College
Hanover, New Hampshire 03755

Heidi R. Ammerlahn, Michael E. Goldsby, Michael M. Johnson,
Daniel E. Rhodes, and Ann S. Yoshimura
Systems Studies and Research Departments
Sandia National Laboratories
Livermore, California 94550

ABSTRACT

Large enterprises are ever more dependent on their Large-Scale Information Systems (LSIS), computer systems that are distinguished architecturally by distributed components – data sources, networks, computing engines, simulations, human-in-the-loop control and remote access stations. These systems provide such capabilities as workflow, data fusion and distributed database access. The Nuclear Weapons Complex (NWC) contains many examples of LSIS components, a fact that motivates this research.

However, most LSIS in use grew up from collections of separate subsystems that were not designed to be components of an integrated system. For this reason, they are often difficult to analyze and control. The problem is made more difficult by the size of a typical system, its diversity of information sources, and the institutional complexities associated with its geographic distribution across the enterprise. Moreover, there is no integrated approach for analyzing or managing such systems. Indeed, integrated development of LSIS is an active area of academic research.

This work developed such an approach by simulating the various components of the LSIS and allowing the simulated components to interact with real LSIS subsystems. This research demonstrated two benefits. First, applying it to a particular LSIS provided a thorough understanding of the interfaces between the system's components. Second, it demonstrated how more rapid and detailed answers could be obtained to questions significant to the enterprise by interacting with the relevant LSIS subsystems through simulated components designed with those questions in mind. In a final, added phase of the project, investigations were made on extending this research to wireless communication networks in support of telemetry applications.

ACKNOWLEDGMENTS

In addition to the paper's listed authors, our thanks go also to the many Sandians who contributed to IDES and LSIS research, design and development, particularly T. J. Sa, T. D. Plantenga, M. F. Hawley, and E. L. Hoffman.

Portions of this paper appear in the following publications:

H.R. Ammerlahn, M.E. Goldsby, M.M. Johnson, and D.M. Nicol, 2000. A Geographically Distributed Enterprise Modeling System. In *Future Generation Computer Systems*, Elsevier, Amsterdam, The Netherlands.

D.M. Nicol, M.E. Goldsby, and M.M. Johnson, 1999, Fluid-based Simulation of Communication Networks using SSF, in *Proceedings of the 11th European Simulation Symposium and Exhibition, Simulation in Industry*, Erlangen-Nuremberg, Germany.

This research is supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-9808964, and the United States Department of Energy under Contract DE-AC04-94AL85000.

CONTENTS

1	Introduction to Large-Scale Information Systems	7
1.1	Technical Issues	7
1.2	Modeling Methodology	8
1.3	Accomplishments	8
2	Fluid-based Simulation of Communication Networks using SSF	10
2.1	Fluid Simulation in SSF	10
2.2	Statistical Measures	11
2.3	Experiments	12
2.4	Summary	14
3	Fluid-based Simulation Modeling of TCP	15
3.1	Overview	15
3.2	Modeling Assumptions	16
3.3	Quick Primer on TCP	17
3.4	Fluid Modeling of TCP	19
3.4.1	Overview	20
3.4.2	Modeling Data Arrival	21
3.4.3	Modeling Maximum Window Size	21
3.4.4	Modeling Raw Transmission Rate	23
3.4.5	Modeling Acknowledgements	24
3.4.6	Modeling Lost Traffic	25
3.4.7	Modeling Fast Retransmit	25
3.4.8	Modeling Timeouts	26
3.4.9	Modeling Retransmission	27
4	Distributed Network of Intelligent Sensing Modules	28
4.1	Integrating Technologies	28
4.2	Creativity and Innovation	29
4.3	Proposed Applications	30
4.3.1	Telemetry	31
4.3.2	Ubiquitous Computing	31
4.3.3	Health of Weapon Applications	32
4.3.4	Industrial Applications	32
4.3.5	Situational Understanding	33
	References	34

Intentionally Left Blank

1 INTRODUCTION TO LARGE-SCALE INFORMATION SYSTEMS

Large enterprises are ever more dependent on their Large-Scale Information Systems (LSIS), computer systems that are distinguished architecturally by distributed components – data sources, networks, computing engines, simulations, human-in-the-loop control and remote access stations. These systems provide such capabilities as workflow, data fusion and distributed database access. The Nuclear Weapons Complex (NWC) contains many examples of LSIS components, a fact that motivates this research.

However, most LSIS in use grew up from collections of separate subsystems that were not designed to be components of an integrated system. For this reason, they are often difficult to analyze and control. The problem is made more difficult by the size of a typical system, its diversity of information sources, and the institutional complexities associated with its geographic distribution across the enterprise. Moreover, there is no integrated approach for analyzing or managing such systems. Indeed, integrated development of LSIS is an active area of academic research.

This work developed such an approach by simulating the various components of the LSIS and allowing the simulated components to interact with real LSIS subsystems. This research demonstrated two benefits. First, applying it to a particular LSIS provided a thorough understanding of the interfaces between the system's components. Second, it demonstrated how more rapid and detailed answers could be obtained to questions significant to the enterprise by interacting with the relevant LSIS subsystems through simulated components designed with those questions in mind. In a final, added phase of the project, investigations were made on extending this research to wireless communication networks in support of telemetry applications.

1.1 TECHNICAL ISSUES

In the systems design area the technical issue is one of interfaces. Design and implementation of LSIS is challenging both because of the complexity of the systems and the rapidity with which the technology used to implement them changes. Moreover, no LSIS system currently exists that unifies both data and process or simulation oriented models. To tie these elements together, to support seamless integration of working system elements and simulated system elements, requires detailed attention to interfaces. These interfaces must capture the essential semantics of the subsystems while at the same time support, for simulated subsystems, flexibility of model fidelity. In addition, these interfaces must support the myriad of data and process-oriented models that exist – commercial simulation packages, database applications, etc.

In the systems modeling methodology area the technical issue is one of abstraction. To support faster-than-real-time system control will require not only parallel processing but expression of the model at a high level of abstraction to support rapid evolution of the simulation model state. We are challenged then to rapidly transform one level of model abstraction into another at interfaces between subsystems. For example, if one subsystem is modeled at a high level of abstraction and interacts with another subsystem's actual implementation, then the abstraction's high-level data flow must be transformed in a semantically consistent form understood by the actual subsystem. Likewise, data and command flows from an actual implementation (or a high fidelity model of the same) must be transformed across an interface to a subsystem modeled at a high level of abstraction.

An issue of time evolution arises when combining process, data, and human-in-the-loop models under a single LSIS – three different interpretations of time must be managed in the combined system. A parallel discrete event simulation (PDES) runs in simulation time, which may advance in discrete jumps. Moreover, the different nodes of a parallel simulation may have different simulation times that are synchronized only occasionally. Events in a PDES can be rolled back, unlike the events in the non-simulated parts of the system. Data models run in “model time”, which advances monotonically (continuously or discretely). Finally, a human-in-the-loop element operates in real time. The notion of proper interaction of these three interpretations of time must be defined, and the definition must be implementable.

1.2 MODELING METHODOLOGY

We extended Sandia National Laboratories' expertise in distributed, discrete-event simulation into multi-mode, multi-fidelity modeling environments where a large, discrete-event simulation is connected to external legacy simulation and database models. The system was used to model and analyze LSIS, and was designed in such a way that actual components of the system are inter-operable with simulated components. This was intended to permit key elements of the system model to control a working system, and allow an integrated simulation model to make faster-than-real-time decisions to optimize that control.

The simulation of LSIS subsystems was done using IDES (Infrastructure for Distributed Enterprise Simulation), a parallel discrete-event simulation engine developed in cooperation between Sandia and Dartmouth College. IDES is written in Java and presents a simple, object-oriented interface to the user. Its design confers on it the properties of robustness, scalability and portability.

A key element of our system design approach was the choice of the Java language. In addition to its well-known platform-independence, its degree of object orientation supported all of the functionality we required. The need for a spectrum of abstraction levels is supported within the language by its inheritance and interface features and its ability to query the nature of an object at runtime. Legacy subsystems can be interfaced to the rest of the system through a wrapper written in Java or replaced by a proxy written in Java.

Geographically remote subsystems (confederates), whether data or process, were integrated regardless of their architecture of implementation language using Sandia's Product Realization Environment (PRE). PRE is based on the open standards of CORBA and DCE and provides a platform-independent intercommunication infrastructure for linking LSIS components. For widest applicability, our implementation supported external interfaces to a number of commercial off-the-shelf (COTS) applications, including the iThink and Extend simulation packages, Microsoft Access and Excel, and standardized databases interfaces like ODBC and SQL.

We developed novel modeling methods that treat information as a "fluid." One of the attractions of a fluid-based approach is that it is amenable to formal methods of abstraction and transformation. One represents a fluid flow as a set of discrete instants in simulation time where the flow changes, such that the description of the flow allows it to be interpolated between those instants. To transform from a finer description to a coarser one at an interface might require averaging and preservation of extrema; to transform from coarser to finer might utilize a stochastic model to add variation while preserving the essential characteristics of the flow. We developed a library of IDES components that manipulate fluid flow: buffers, split-points, fusion points, synchronization points, database and network elements. All included the sort of transformation techniques just described and were used in the construction of simulated LSIS components.

The key technical problems were in software system design, and development of new system modeling methodologies to support distributed, information system connectivity. The system design problem is one of linking an optimistically synchronized distributed simulator with external sub-models, data sources, and human-in-the-loop components. The design methodology problem is one of modeling large-scale information flow through LSIS systems, and maintaining model consistency through different levels of model abstraction.

1.3 ACCOMPLISHMENTS

Our work showed that reduced-fidelity models of a significant portion of an actual LSIS (and one actual model) could be combined in a coordinated simulation environment with human-in-the-loop control. We constructed a simple, unclassified simulation of a TCP network. Simulating this communication system was done in parallel in Dartmouth SSF and IDES. This was done to test the flexibility with which the workload in the various parts of a network could be varied to allow comparison of low- and high-fidelity models.

"Extend external information system interfaces to include iThink, Microsoft Excel, and Oracle databases."

Java interfaces were written for Microsoft Excel. External model interface improvements were made to enhance interaction with the COTS iThink application. Interfaces to allow communication between IDES, written in the Java programming language and external information systems written in Extend, Microsoft Access, and Microsoft Excel have been written. These wrappers allow IDES to drive models written in these COTS software packages during a simulation. Methods for converting data formats from each software package to Java were also written.

"Augment IDES simulation library to support LSIS fluid information flows, integrating advanced modeling techniques for LSIS systems."

Work in modeling information as a fluid was undertaken in conjunction with Dartmouth College. Dartmouth is taking part under a DARPA grant in DARPA's Scalable Self-organizing Simulation (S3) project, which has as its aim the achievement of breakthroughs in the simulation of very large-scale communications networks. Although Sandia's interest may not be in simulating networks per se, the techniques developed for large networks will also be applicable to massive simulations in other application domains. A pair of simulations of interconnected communications switches was written, one using a fluid model and the other a packet model, in order to explore fluid techniques and compare a non-trivial fluid simulation to its more detailed counterpart.

"Integration of actual LSIS system and faster-than-real-time simulation-based management of that LSIS system for an actual combined simulation data model with the NWC."

The CoMPASS (Confederation of Models to Perform Assessments in Stockpile Stewardship) system was used as a platform for integrating LSIS components. Reduced-fidelity models (RFMs) of the Transportation System Division's Transportation Planning Model (TPM), the Pantex Planning Model (PPM) and the Y-12 CSA planning model were integrated. The real Tritium Production Model was released to the CoMPASS team and was integrated with CoMPASS. Use of the RFMs provides an example of faster-than-real-time simulation; for instance, the real PPM can require eight hours to produce a schedule, whereas the RFM PPM produces one in seconds. Various Defense Program Requirements Planning Model (DPRPM) models were integrated into CoMPASS in FY00.

"Simulate selected external sub-models and data sources and investigate the effect of substituting the simulated entities for real ones. Make simulations of different fidelities and investigate their interactions with the other system components."

Traffic source and sink devices common to SecureNet were modeled and simulated in a simplified, unclassified form. The simulation was parameterized to run at different resolutions. Intermixing of submodels of different fidelities was used to investigate communication mechanisms between high- and low-level simulation entities. In order to compare full-detail and reduced-fidelity models on a large scale, it was necessary to simulate very large traffic flows at the packet level. The ability of IDES to handle larger simulations was accomplished through the application of recently developed techniques (Fujimoto et al, to be presented at the Winter Simulation Conference). These techniques dramatically improved IDES' memory usage, enabling it to handle correspondingly larger models on given hardware.

"Applications of fluid-based simulation to wireless telemetry applications."

Using year-end, additional LDRD budget, fluid-based communication system simulation techniques were applied to wireless telemetry systems. Prototype hardware and software was developed to demonstrate intercommunication between wireless components and investigate communication between remote sensor model simulations. Due to the late start of this additional phase of the project, work is on-going into FY01 using separate funding.

2 FLUID-BASED SIMULATION OF COMMUNICATION NETWORKS USING SSF

Fluid-based models of communication network traffic consider traffic behavior in terms of packet *rates*, rather than packet *instances*. While fluid modeling is commonly used in mathematical approaches, these often require model simplification. Simulation of fluid models may be used in order to retain necessary model detail; it is beginning to receive some attention in the literature [1, 3, 4, 6, 5]. We provide here a cost-benefit study of such techniques, *within a common framework*.

We describe a simulation tool for communication systems, and its use to simulate fluid-based models. Within the same tool we are able to develop and compare packet-based, and fluid-based models, assess any performance benefits, and assess inaccuracies introduced. The specific study made here is of networks where traffic is generated as a Modulated Markov Process and is shaped using a Leaky Bucket flow-control mechanism. Switches model FCFS service among all competing flows at output ports. The principle contributions of this paper are to demonstrate fluid simulation in the context of an existing packet-oriented simulator, and to study the performance/accuracy trade-offs between packet-based and fluid-based models.

2.1 FLUID SIMULATION IN SSF

The Scalable Simulation Framework (SSF)[2] is a Java and C++ based API for the description of network models (see <http://www.ssfnet.org>). Three implementations of this API exist, one in Java and two in C++. The objectives of SSF are to provide a public domain standard of discrete-event simulation of large complex systems. SSF models are compact, flexible, portable, and transparently parallelizable. At the base level the SSF API defines only five base classes: `Entity`, `Process`, `Event`, `InChannel`, and `OutChannel`. An `Entity` is a container of state, of `Processes`, and of communication endpoints. SSF is process-oriented; a `Process` is a bit of C++ code that executes when triggered by a time-out, or the arrival of an `Event` on an `InChannel` it has specified. The process may modify `Entity` state variables and create `Events` that are written to `OutChannels`.

A simple SSF topology might consist of traffic sources, each of which is connected to traffic shapers. Traffic leaving a shaper enters a switch, and is routed to a Host. Hosts in turn might be interconnected through Routers. In an SSF model, each of the fore-mentioned objects would be modeled as an `Entity`, with their interconnections modeled using `InChannels` and `OutChannels`. `Entity` state variables record statistical information, and maintain a description of traffic as it flows through the system. A `Process` at the traffic source will use time-outs to govern when it wakes up to write more traffic to the shaper; the shaper will have one `Process` to respond to new traffic from a source, and another to pass traffic through to a switch. The two processes may coordinate with each other through state variables in the `Entity` they share. Likewise, a switch `Entity` will have a `Process` to respond to new inputs, and another to route outgoing traffic.

The description above applies whether the traffic is a stream of packets or a fluid flow. This isomorphism is important, as the simulator infrastructure is the same for both cases; indeed, much of the code is identical. For instance, the traffic generator is a Modulated Markov Process (MMP). This means that the traffic generator moves around in “states” of a continuous time Markov chain (CTMC); in state i of this CTMC traffic is considered to be generated at (constant) rate $\lambda(i)$. Our two implementations use the same MMP code, and same “Fluid Leaky Bucket” traffic shaping code. In both cases the output of the Fluid Leaky Bucket is a sequence of time-stamped “Fluid Events”; each such event describes the new output rate of packets. In the fluid-based model these events are fed directly into a fluid-based switch model, whereas in our packet-based model they are fed into a “Fluid-to-Packet” converter that emits packets at the specified rate. These packets are fed to a packet-based switch model. In both models the output of a traffic source defines an identifiable “flow” through the network. At startup each such flow is mapped through the switching network, where at each switch the input port and output port are specified. Routing is therefore static; given a flow id, a switch knows how to route that flow (the packet simulator knows where to route a new packet and the fluid simulator knows which output queue is affected by a change in the incoming flow rate for the identified flow).

The key difference between our fluid-based and packet-based models is the implementation of the switch. FCFS ordering of packets at output ports is easily implemented in the packet-based model. The fluid model is more delicate. If flow i changes its input rate to a FCFS queue (which receives input from multiple flows) at time t and there are the equivalent of p packets already in queue or service at time t , then the effects of that change on allocation

of service to flow i packets should not be observed until all p packets in the queue at time t have been served. In effect, the rate change that occurs at time t is scheduled to change the mix of fluid entering service at time $t + p/\mu$, where μ is the queue service rate. Therefore, at any instant s each flow i receiving service has an associated input flow rate $\lambda_i(s)$; service rate μ at time s is allocated to the different flows in proportion to their input flow rates at s .

The key difference in computational effort between package-based and fluid-based models is that the number of events executed at a switch in the former model is proportional to the number of all packets that enter the switch, whereas in the fluid model it is proportional to the total number of MMP state changes. Thus we can expect computational savings as the number of packets emitted during a MMP burst period is large. However, fluid models are approximations to packet models, and some care must be taken when measuring statistics. We examine that issue next.

2.2 STATISTICAL MEASURES

We consider traffic through a switch to be comprised of a number of individual flows, and presume that for each flow one desires to estimate the percentage of lost packets and the mean packet delay through the switch. We understand a flow to be mapped from one input port to a queue serving an output port. Losses occur when a flow's packets arrive to find the queue full. The delay a packet receives is the sum of service times of all packets ahead of it in the queue, plus its own service time. The service delay is the reciprocal of the port's output bandwidth.

The queue is in one of 4 states:

1. empty, with no active flows;
2. non-empty, not full, with flows entering at an aggregate rate different from the output bandwidth;
3. full, with flows entering at an aggregate rate greater than the output bandwidth;
4. flows entering at a positive aggregate rate that is equal to the output bandwidth.

In the fluid simulation, at any point statistics are gathered for a switch model, it will be for a period of simulation time $[s, t]$ over which the arrival rate and the departure rates are constant. This means that any time statistics are gathered, the queue has been in exactly one of the above states, over the entire interval of interest. Now in our fluid model losses are suffered only when the queue is in state (3); the estimated number of packets lost is the product of the length of time the queue remains in state (3) times the difference between the aggregate input rate and the output bandwidth. Each flow is ascribed a fraction of this loss in proportion to the rate of the input it contributes over this interval. This estimate is not exact; in a packet-based model, even if the average arrival rate is equal to the average departure rate, variation in arrival times can deliver a packet to a full queue.

Over period $[s, t]$ we compute the sum of waiting delays for all packets that are modeled by the fluid to have arrived in $[s, t]$. If the constant arrival rate is λ over that period and the constant service rate (i.e. bandwidth) is μ , then we define N as the integer part of $(t - s)\lambda$, define Q_s to be the fluid level at time s , estimate the number of packets ahead of an arrival at $u \in [s, t]$ to be $Q_s + (u - s)(\lambda - \mu)$ and so estimate the response time of an arrival at time u as $(Q_s + (u - s)(\lambda - \mu) + 1)/\mu$. Our estimate of the sum of all such waiting times for all arrivals in $[s, t]$ is

$$\sum_{i=1}^N (Q_s + (i/\lambda)(\lambda - \mu) + 1)/\mu$$

This approximation reflects simplifying assumptions that interarrival times are deterministic and equal to the reciprocal of the average arrival rate, and that a full interarrival period goes by before the first arrival.

At the point we measure packet loss and average packet delay over an interval $[s, t]$, the arrival and service rates we use are aggregate rates, taken over all individual flows mapped to the switch. We accumulate loss and delay statistics on a per-flow basis; after computing the aggregate loss and delay, we partition these statistical values to the individual flows in proportion to the flows' arrival rates over $[s, t]$. For example, if the arrival rate of one flow is fully half of the aggregate arrival rate of all flows, then that flow receives credit for one-half of the measured loss packets, and one-half of the aggregate sum of packet delays.

2.3 EXPERIMENTS

Next we turn to some experiments that seek to estimate the error introduced by fluid approximations, seek out sensitivities in that error, and concurrently look at the potential speedups that might be achieved using the approach.

Intuition suggests to us that at least five factors will be important in our study. Buffer size in a switch is one, for that has direct impact on loss rates. Traffic load is another factor, for that largely determines the fraction of time a switch spends in each of the four states outlined above. A third factor is the size of the network being studied, and a fourth is the number of flows concurrently mapped to a switch. The last factor is the average number of packets generated between transitions of the MMP.

We have compared the performance of fluid-based and packet-based SSF models that share a great deal of common code and common infrastructure *except* for the generation and handling of events. We first report on studies at one switch, and follow-up looking at networks of up to hundreds of switches and thousands of flows. In these studies we look at relative error in the number of lost packets, and in the mean packet response time, measured as $(f_m - p_m)/p_m$, where f_m and p_m are the fluid and packet measures of interest, respectively. Figure 1 illustrates the relative error in response time and in packet loss for a single switch, where there is a single traffic source the arrival and service rates are set to create approximately 10% packet loss. The MMP is an on / off process. The independent variable is the switch buffer size, measurements are taken at 4 16 64 and 128 packets. Curves for four different “time scales” are presented. In time scale X (for $X = 1\ 10\ 100\ 1000$) the average number of packets generated by one “on” burst is close to $10X$. We see that response time errors are largely independent of time-scale, are quite small in the worst case (less than 10%), and quite good for large buffers. Time scale matters more with respect to lost packets, but even here the worst relative error observed is less than 1.5%.

Interestingly, the magnitude of these errors does not seem to depend on the traffic intensity *except* when it is so low so as to nearly eliminate packet loss. In this case, while the absolute error in packet loss is considerably less than 1%, the relative error could be much larger – you see a 50% relative error if two packets in 1000 is lost rather than an estimated one packet. This shows up the (not terribly surprising) limitation of fluid estimation of very small packet losses. It should be noted that the estimate of response time remains very good, even when the traffic rate is so large as to saturate the system.

The performance benefit of fluid simulation should clearly depend on the time scale, for changing the time scale does not much change the number of events simulated in the fluid model, but very much changes the number of events simulated in the packet model. For the runs described in Figure 1, with a buffer length of 64, the accelerations of fluid based simulation versus packet-based simulation are 4.6 for a time scale of 1, 33 for a time scale of 10, 112 for a time scale of 100, and 444 for a time scale of 1000. Clearly *very* significant performance benefits are possible with relatively small error.

Next we consider how increasing the number of flows into the switch affects the per-flow errors. Here we fix the size of the switch queue at 64, and adjust the relationship between traffic generation rates and service rates to create approximately 5% loss rate. Figure 2 shows the relative errors found as a function of varying the number of flows among 1, 2, 4, 8, 16. Relative error in response time is taken as the per-flow relative error. It is again relatively small – in these experiments less than 4% in magnitude. The data does suggest some sensitivity in the number of flows, however with error increasing with the number of flows. Average relative error in packet loss is again small, and again seems to grow in number of flows.

Figure 1. Relative Errors for Single Source Queue

To illustrate the performance benefits in this case we consider speedup for the 16-source case as a function of time scale. Here we observe a speedup of 8 for time scale 1, 34 for time scale 10, 219 for time-scale 100, and 2116 for time-scale 1000. Clearly increasing the number of sources had no detrimental effect on the performance potential of fluid simulation.

To conclude this preliminary study we examine the behavior of networks. The switches of each network are interconnected as a binary hypercube of dimension d , and a number s of flow-shaped MMP sources are attached to each switch, for a total of $s2^d$ flows. Each switch has one output port for each hypercube neighbor (with capacity 64 packets), and one sink port. The flows are routed so that the j^{th} flow originating at switch r is mapped through the network to be received and terminated at switch $(r \cdot s + j) \bmod 2^d$. Standard hypercube routing is used (a flow arriving at switch r , destined for switch f , is routed to the hypercube neighbor of r with least order bit that differs from r 's and is identical to f 's). This selection of flow mappings exercises all parts of the network, and creates a variety of congestion conditions flows may experience.

Our experiments cover hypercubes of sizes 4, 8, 16, 32, 64, 128 and 256 switches. We vary the number of sources assigned to a switch between 2 and 16, and we vary the switch output port service rates to affect flow losses. In the "low loss" case it is set so that on ports where there is loss, it is limited to about 2%; in the "high loss" case overloaded links lose about 40% of the flow. All these experiments use a MMP timescale of 10 (so that approximately 10 packets are generated per flow burst). As before, we compare the average per-flow response time and per-flow loss statistics from fluid-based and packet-based implementations, and compute the relative error. Each flow's response time statistic is computed by adding the relevant components from each switch on its path through the network; its fraction of delivered packets is computed as the product of this fraction at each switch on its path.

Figure 3 shows the relative errors found as a function of network size. The main and important point is that the errors are small. The promise of accuracy observed in the single switch case is delivered for networks, at least for the cases studied here. The errors when there are 2 sources per switch and low loss are miniscule, less than 2% at all network sizes. The speedup of the fluid implementation increases from 34 at the smallest network size to 46 at the largest. When the low loss case runs with 16 sources per switch, the response time errors increase (but the loss errors do not). The response time errors increase with network size, but even so the errors are less than 13%. The fluid implementation's speedup in this case hovers around 20 for all network sizes, with no discernible trend.

Moving to the high loss cases, with 2 sources per switch we see very low response time errors ($< 1\%$), low delivered traffic errors ($< 5\%$), and significant speedups (increasing from 38 with the smallest network to 66 with the largest). Good error behavior is also observed in the case of 16 sources per switch; response time error remains less than 5%, and the delivered traffic error is less than 13%. The most distinguishing facet of this case is speedup behavior. A speedup of 7.5 is obtained by the 4 switch network, but with increasing network size the speedup drops so much that on the 256 switch network the fluid implementation actually runs twice as slowly! The explanation for the performance decline lies in the accumulating effects of flow modulation. A heavily loaded output queue simultaneously carries a multiplicity of flow descriptors, of flows sharing the output port. Whenever one of the input flow descriptors changes on *any* of these, the output flow descriptors change on *all* of the flows. These changes in turn propagate as changes to the inputs of a multiplicity of queues at the switch connected to the other end of this port, which further propagate. This is the very phenomena analyzed and modeled by [5]. In the 256-switch case the fluid simulation executes nearly as many events as does the packet simulation, whereas in the 4-switch case it

Figure 2. Relative Errors for Multiple Source Queue

executes only 1/5 as many; each fluid event is significantly more computationally expensive, resulting in the slowdown.

This performance degradation is a high priority concern, and we are working on ways of “smoothing” fluid behavior to reduce the propagation effects. The results of those efforts are beyond the scope of this paper.

2.4 SUMMARY

Fluid-based simulation of networks offers great potential for accelerated performance, but questions must be asked about the accuracy of such simulations, and whether the comparisons made to packet-based networks are fair. We have addressed the second issue by developing fluid modules that plug into a high-performance simulator of communication networks, making a performance comparison absolutely fair by using as much common framework code as possible. We are beginning to address the question of accuracy, which is essentially an issue of how we gather statistics from a fluid simulation. We describe here methods of gathering per-flow estimates of packet loss and of packet response time, and examine the accuracy of these methods as a function of timescale, number of sources, and switch buffer length. We find acceptably small errors, and the potential of tremendously large speedups, depending on the nature of the traffic. Performance problems remain with ripple effects, an area we are currently addressing.

Figure 3. Relative Errors for Hypercube Networks

3 FLUID-BASED SIMULATION MODELING OF TCP

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 chapter 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 and how fluid models give rise to specific challenges that must be addressed by modeler and simulation kernel.

3.1 OVERVIEW

It is impossible to underestimate 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 timeouts 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!

In this chapter we describe a fluid-based simulation model of TCP. This model describes 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 session 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 might be different from that of a segment-oriented model. Nevertheless, intuition tells us that potential exists; we are working to exploit and assess it.

We first show how a rich variant of TCP can be modeled in a fluid framework – 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.

3.2 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 segments would 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 interlocked with TCP with respect to data buffers (e.g., it does not 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 for an agent that sends data in bulk, an agent that merely sends TCP headers containing acknowledgement information in response to that data, and an agent that sometimes sends segments containing either data or headers.

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 interarrival 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 (possibly 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.

Following standard practice we model routers with “fluid switches.” We provide only an informal description for a precise mathematical treatment. A flow entering a switch is mapped to an output port; service at each output

port is governed with a fluid version of a work-conserving FCFS queue. Under congestion conditions different flows may affect each other through these queues. Let μ be a port's service rate (we normally define μ to be the bandwidth of the communication channel fed by the port). If at time t there are n flows mapped to a port with raw byte arrival rates $\lambda_1(t), \dots, \lambda_n(t)$, and $\sum_{i=1}^n \lambda_i(t) \leq \mu$, then we assume there is no queuing and that every flow's rate out of the port is identical to its rate into it. However, the instant that the aggregate arrival rate exceeds μ a queue begins to build. Because of queuing, variations in arrival rates are not immediately reflected at the output port. When a queue forms, all flows' output rates will be strictly smaller than their input rates (unless of course a flow's input rate at that point is zero). Thus if a queue began to form at time t , and a rate change occurs at time $t + d > t$, then in the queue there will be a density of fluid related to the flow that must be served before fluid that follows the rate change is served. To capture this dependency we describe a queue in terms of epochs during which all arrival rates are constant. A change in any flow's input rate defines the beginning of a new epoch. Because of FCFS assumptions, all of the fluid that arrives within an epoch is served before any fluid from a subsequent epoch is served. Under congestion then, a port is allocating service to all fluid that arrived within some epoch during which arrival rates were constant. It allocates its service to a flow in proportion to the flow's arrival rate during that epoch. That is, if s is some instant within the epoch being served at time t , the departure rate of flow i is

$$\lambda'_i(t) = \mu \times \frac{\lambda_i(t)}{\sum_{i=1}^n \lambda_i(t)}.$$

Note that if the sum of arrival rates during an epoch is less than μ , then $\lambda'_i(t) > \lambda_i(t)$, a consequence of work-conserving assumptions. Another important point to observe is that when a queue has formed, a change in *any* flow's raw byte rate eventually affects the raw byte rate of *every* flow mapped to that port. We will later see how this consequence creates problems that the simulation framework must explicitly address. A final note is that FCFS is by no means the only queuing mechanism a router might employ. A different queuing policy will of course require a different fluid formulation.

A fluid queue with finite capacity suffers losses when the queue is full and the aggregate arrival rate continues to exceed the service rate. Fluid is lost at a rate given as the difference of the two, and ascribed to different flows in proportion to their arrival rates. Whenever the loss rate of a flow changes, we compute the proportion of the flow (p) that is *not* lost, and create a new rate change whose raw byte rate reflects the diminished flow, and whose delivered fraction component is the product of p with the delivered fraction of the flow at the point of loss. Maintenance of a flow's delivered fraction allows an agent receiving a flow to determine the volume of traffic sent by the source.

3.3 QUICK PRIMER ON TCP

TCP is a full duplex protocol. Traffic flows in both directions between agents, even if one of the directions contains only acknowledgements. TCP views *data* flow in terms of a contiguous sequence of enumerated bytes – bytes in the TCP header are not enumerated. It refers to a segment by the index of its first data byte, called the segment's *sequence number* (the first sequence number is actually chosen randomly when the session is established and the segment's byte offset is added to this; this aspect has no bearing on our discussion and will be ignored). 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 *acknowledgement number* field – the sequence number of the next segment the agent expects to receive. The header also contains its sending agent's *advertised window*, which is the number of bytes available to buffer incoming traffic. This header information places a crucial role in governing TCP's behavior.

TCP is a sliding window flow control protocol, where the window is related to sent segments and received acknowledgements. 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 acknowledgement number seen by this agent in any segment from its partner. TCP dynamically computes the maximum value that 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. A stalled agent may be released on receiving a segment whose acknowledgement number

increases LBA. We defer discussion on rules for governing maximum window size, to talk about acknowledgements and retransmissions.

An agent expects to receive segments in contiguous order, and can always compute the sequence number of the next segment it should receive. Under normal circumstances an agent receives the next segment it expects, and either explicitly sends a header (with no data segment) with an increased acknowledgement number, or soon sends another data segment whose header contains that acknowledgement 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 setting a timer on each segment it sends. The segment is considered to be lost if the timer expires before the segment is acknowledged. However detected, an agent retransmits a segment it considers to have been lost. Too many timeouts are taken as indication that the partner has been lost, and triggers a shutdown.

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, and additional rules provide congestion control. The idea is that the maximum window size should be the amount of data that can be carried through the network in one round-trip delay time, from sender to receiver and back. In the ideal case, just before the application presents a new segment for transmission to a full window, an acknowledgement 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 should be smaller. Furthermore, TCP agents do not know the bandwidth of the network links between them nor the latencies. The congestion control rules try to identify what the window size should be, and react quickly when the traffic congests.

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. Rather than open a large window and immediately pump many segments into the network, slow-start grows the window in a simple fashion. 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 acknowledgement for the first segment comes in. *cwnd* 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 timeout, or triple-duplicate acknowledgement). 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 mode.

TCP must detect and retransmit lost segments. One detection mechanism is for an agent to run a timer for every segment sent. The segment is considered lost if the timer fires before the segment is acknowledged. The segment is immediately retransmitted. An agent also uses the “fast retransmit” mechanism. This works because a acknowledgement segment is sent for (nearly) every segment received. The acknowledgement number is the highest numbered *consecutive* byte received to date. Thus, if a segment is lost, subsequent segments received each trigger an acknowledgement segment that has the same acknowledgement number as the previous acknowledgement segment. Under fast retransmit, if an agent receives three consecutive segments with the same acknowledgement 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.

3.4 FLUID MODELING OF TCP

The heart and soul of TCP is its control of the window. Whereas the TCP specification discretely advances LBA and LBS, 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 $(\lambda_{byte}(t), p(t), \tau_{data}(t), \tau_{ack}(t))$ 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 $\lambda_{byte}(t) = 0$. $p(t)$ is the delivered fraction. Informally, the flow being received at time t corresponds to flow transmitted by the source at some s ; $t - s$ 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 seen to be $\lambda_{data}(t) = \tau_{data}(t) * \lambda_{byte}(t)$ and $\lambda_{ack}(t) = \tau_{ack}(t) * \lambda_{byte}(t)$, respectively.

Because TCP is a full duplex algorithm, a segment carrying data may, in its header, carry an acknowledgement number for data the sender has 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 acknowledgement 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 acknowledgements. 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 defines 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 acknowledgements 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 the incoming flow's reference point to the time of receipt, and 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

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

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.

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

Table 1 summarizes these definitions, as well as others yet to be introduced.

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.

3.4.1 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 $LBS - LBA$ to the maximum data window size $\min\{adw, cwnd\}$. In the fluid formulation, the data transmission rate at time t depends on the relationship of $LBS(t) - LBA(t)$ to $\min\{adw, cwnd(t)\}$. To illustrate, consider Figure 4. At the beginning of the period depicted, $\lambda_{ack}^{(in)}(t) > \lambda_{data}^{(out)}(t)$ for $t < 2$. In this interval $LBS(t) - LBA(t)$ actually decreases as a function of t . At time 2 however the acknowledged byte rate diminishes so that $LBA(t)$ grows more slowly in t , and at time 4 the acknowledged byte rate drops to zero and $LBA(t)$ stops growing in t entirely. The value of $LBS(4) - LBA(4)$ is presumed here to be less than $\min\{adw, cwnd(4)\}$, and so transmission can continue unabated until the window is full, at time 6, when $LBS(6) - LBA(6) = \min\{adw, cwnd(6)\}$. 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 4 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) $LBS(t) - LBA(t)$ grows linearly in t . This cannot continue ad infinitum for eventually the window limits will be reached. Using this function, the earliest time s such that $LBS(s) - LBA(s)$ 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 cancelled, no longer being valid. $LBS(t) - LBA(t)$ 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 time 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.

$\lambda_{byte}^{(out)}(t)$	outgoing raw byte transmission rate	$\lambda_{byte}^{(in)}(t)$	incoming raw byte transmission rate
$\lambda_{data}^{(out)}(t)$	outgoing data byte transmission rate	$\lambda_{data}^{(in)}(t)$	incoming data byte transmission rate
$\lambda_{ack}^{(out)}(t)$	outgoing acked byte transmission rate	$\lambda_{ack}^{(in)}(t)$	incoming acked byte transmission rate
$p^{(out)}(t)$	delivered fraction, outgoing flow	$p^{(in)}(t)$	delivered fraction, incoming flow
$\tau_{data}^{(out)}(t)$	outgoing data byte ratio	$\tau_{data}^{(in)}(t)$	incoming data byte ratio
$\tau_{ack}^{(out)}(t)$	outgoing acked byte ratio	$\tau_{ack}^{(in)}(t)$	incoming acked byte ratio
$B_{data}^{(out)}$	size of data segment, outgoing flow	$B_{data}^{(in)}$	size of data segment, incoming flow
B_{hdr}	size of TCP header		
$B_{seg}^{(out)}$	size of segment, outgoing flow	$B_{seg}^{(in)}$	size of segment, incoming flow
$LBS(t)$	Last Byte Sent function	$LBA(t)$	Last Byte Acknowledged function
$cwnd(t)$	Congestion control window size function	$\lambda_{cwnd}(t)$	Growth rate of $cwnd$ in time
adw	Advertised receive window		
$mxwnd(t)$	Maximum window size	$\lambda_{mxwnd}(t)$	Growth rate of $mxwnd(t)$ in time
$\lambda_{app}(x, f(t))$	application data offered rate	$\lambda_{src}(t)$	maximum data transmission rate

Table 1: Functions used in fluid model description

3.4.2 MODELING DATA ARRIVAL

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 .

Figure 4. Sender Window Behavior in Fluid Formulation

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 time t .

3.4.3 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 $\lambda_{cwnd}(t) = \lambda_{ack}^{(out)}(t)$. 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 $cwnd$ 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 5 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 acknowledgement 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 $LBS(21) - LBA(21) < cwnd(21)$ and schedules a "window filled" event to occur at 22. The window filled event stops the transmission because the acknowledgement 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, $cwnd(42)$ 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

$$mxwmd(t) = \min\{adrw, cwnd(t)\},$$

and define its rate function as

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

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 $LBS(s) - LBA(s) = adrw$, given the present rates of flow, while the other is the earliest time t when $LBS(t) - LBA(t) = cwnd(t)$, 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 $cwnd$ so we define $\lambda_{cwnd}(t)$ to *approximate* the real non-linear growth with a piece-wise constant function. We schedule events at instants when the constants change as follows.

Figure 5. Behavior of Slow-Start in Fluid Model

Imagine that at a scheduling point, the last segment acknowledged has index A , and that $cwnd$ reflects C_0 segments. We call segments A through $A + C_0 - 1$ the first *window set*, segments $A + C_0$ through $A + C_0 + (C_0 + 1) - 1$ the second window set, segments $A + C_0 + (C_0 + 1)$ through $A + C_0 + (C_0 + 1) + (C_0 + 2) - 1$ the third window set, and so on. In congestion avoidance mode, the acknowledgement of a window set's worth of data increases $cwnd$ by one segment's worth of data. We can approximate the growth behavior over k window sets with a constant growth rate that delivers an increase of k segments after these k window sets are acknowledged. There are $kC_0 + k(k - 1)/2$ segments in k window sets, implying the constant ought to be $k/(C_0 + k(k - 1)/2)$. Since $cwnd$ grows as a function of the acknowledged byte flow, for times t where the acknowledged byte stream refers to these k window sets, we define

$$\lambda_{cwnd}(t) = \lambda_{ack}^{(in)}(t) * k / (C_0 + k(k - 1)/2).$$

We deliberately left variable k unspecified. We wish to present a strategy for modeling congestion avoidance that allows for user management of an inherent tradeoff. For a given k , there will be at least one event for every $C_0 + k(k + 1)/2$ segments. For small k and C_0 , that may cause $cwnd$ updates to be the bottleneck in the simulation. The tradeoff is that using small k is more faithful to TCP's definition, whereas using large k allows for greater execution speed.

There are circumstances that allow us to suspend $cwnd$ update events. As we will see in the subsection to follow, the reason for explicitly modeling $cwnd$ growth rates is to capture the effect that $cwnd$ growth has on the transmission rate in situations where the transmission rate is permitted to grow. The transmission rate is unaffected by changes in $cwnd$ if it is already at application or bandwidth speeds. Likewise, the transmission rate is insensitive to increases in $cwnd$ when $adrw < cwnd$. The simulation can detect these states and suspend $cwnd$ updates, but keep sufficient information at hand so that the value of $cwnd$ can be computed and the updates resumed, when needed.

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

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 and then present a table that specifies the flow description as a function of agent state.

The first principle is that if state constraints force the data outflow ratio to be zero (i.e. $\tau_{data}^{(out)}(t) = 0$), 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:

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

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

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 acknowledgements, 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 acknowledgement 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, acknowledgement 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 $1.0/\text{cwnd}$. The data transmission rate is designed to retain the equality $\text{LBS}(t) - \text{LBA}(t) = \text{mxwnd}(t)$ 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. $\lambda_{\text{ack}}^{(in)}(t) > 0$), we need

$$\lambda_{\text{data}}^{(out)}(t) = (\lambda_{\text{ack}}^{(in)}(t) + \lambda_{\text{mxwnd}}(t)),$$

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

Raw Transmission Rate $\lambda_{\text{byte}}^{(out)}(t)$	Data Ratio $\tau_{\text{data}}^{(out)}(t)$	(Source Flow, Incoming Acks, Incoming Data, Window State) $\lambda_{\text{src}}^{(out)}(t), \lambda_{\text{ack}}^{(in)}(t), \lambda_{\text{data}}^{(in)}(t), \{\text{open, saturated}\}$
0	0	(off, -, off, -)
$(B_{\text{seg}}^{(out)} / B_{\text{data}}^{(out)}) \lambda_{\text{src}}(t)$	$B_{\text{data}}^{(out)} / B_{\text{seg}}^{(out)}$	(on, -, -, open)
$(B_{\text{hdr}} / B_{\text{data}}^{(in)}) \lambda_{\text{data}}^{(in)}(t)$	$B_{\text{data}}^{(out)} / B_{\text{seg}}^{(out)}$	(off, -, on, -)
$\lambda_{\text{ack}}^{(in)}(t) + \lambda_{\text{mxwnd}}(t)$	$B_{\text{data}}^{(out)} / B_{\text{seg}}^{(out)}$	(on, on, -, saturated)

Table 2: $\lambda_{\text{byte}}^{(out)}(t)$ and $\tau_{\text{data}}^{(out)}(t)$ as a function of agent state

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 $\lambda_{\text{src}}(t)$ to reflect maximum data transfer from the buffer; the raw transmission rate is scaled to account for header information.

Table 2 applies these principles to the set of state vectors. We denote a “don’t care” condition in a component by “-.” Three states are not represented. Each is transient because exactly one of the source flow and acknowledged byte flows is on, and the window is saturated. If the source flow is on, then it must instantaneously stop. If the acknowledged byte flow is on, then the window instantaneously transitions into the open state.

3.4.5 MODELING ACKNOWLEDGMENTS

The role acknowledgements play in TCP is paramount, so we do well to try and capture its key features. An important feature is that acknowledgements move the send window and increase its size. We judge therefore that timeliness of response is critical. Our fluid model acknowledges error-free 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 piggybacking the acknowledgement flow onto the data. If however the receiving agent is not transmitting data, we model transmission of data-free TCP headers carrying acknowledgement information (this is reflected in Table 2).

If the received flow reflects any loss ($p^{(in)}(t) < 1.0$), then the outgoing acknowledged byte flow is zero, to reflect TCP’s requirement that a segment is acked only if all previous segments have already been successfully received. In our model, the acknowledgement flow will not start again before being reset by receiving a “reset cork”

embedded in a rate change whose delivered fraction is 1.0. We say the incoming flow is *sullied* if ever its delivered fraction component was less than 1.0 since last being reset. Thus, $\tau_{ack}^{(out)}(t) = 0$ if the incoming flow is sullied at t .

We have already described how the outgoing acknowledgement 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

$$\tau_{ack}^{(out)}(t) = \begin{cases} (\lambda_{byte}^{(in)}(t)\tau_{data}^{(in)}(t)) / \lambda_{byte}^{(out)}(t) & \text{for an un-sullied flow, when } \lambda_{byte}^{(out)}(t) > 0 \\ 0 & \text{when } \lambda_{byte}^{(out)}(t) = 0 \text{ or the incoming flow is sullied at } t \end{cases}$$

Table 2 and the equation above 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. It remains for us to define the discrete events that force changes in flow descriptions within an agent's state, and the events that force transitions between states. First, we continue our description of modeling TCP functionality.

3.4.6 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 component 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 $(\lambda_{byte}^{(in)}(t) - \lambda_{loss}(t), p^{(in)}(t) * (1.0 - \lambda_{loss}(t) / \lambda_{byte}^{(in)}(t)), \tau_{ack}^{(in)}(t), \tau_{data}^{(in)}(t))$. These modifications allow a device receiving the flow description to infer what the corresponding raw transmission rate was at the source by simply dividing the received raw transmission rate by the received delivered fraction.

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.

3.4.7 MODELING FAST RETRANSMIT

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

In our model, an agent detects "retransmission" of acknowledgement 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.

3.4.8 MODELING TIMEOUTS

One way TCP recognizes that a segment has been lost is if a long period of time elapses after it has been transmitted, without being acknowledged. This being an essential component of TCP, we need to capture it in the fluid formulation.

Real TCP implementations associate a timer with each sent segment. A time-out occurs if the timer fires before the segment is acknowledged, and the segment is retransmitted. In “go-back-N” implementations of TCP, all subsequent segments are also retransmitted, a logic we find convenient to model. A fluid equivalent of “go-back-N” logic follows from the observation that we need be concerned only about identifying the first segment to time-out. Once that occurs all later segments are sent again.

To schedule a potential time-out we need to identify *which byte* in the stream first times out; the time-out event is scheduled one time-out epoch after the time of transmission.

The geometry of the key portion of the scheduling problem is illustrated in Figure 6. Here we focus on the behavior of LBS over an interval $[s, t]$ in the past, plotting both the function and its translation ahead T units of simulation, T being the timeout epoch. Starting with the current value $LBA(now)$, we consider three different possibilities for LBA trajectories assuming that $\lambda_{ack}^{(out)}(x)$ does not change for $x \geq now$. The first trajectory (labeled LBA1) corresponds to an acknowledged byte rate fast enough so that it reaches value $LBS(t)$ before time $s + T$. This means that all the bytes transmitted in $[s, t]$ are acknowledged before timing out. The second trajectory, LBA2 corresponds to a slower acknowledgement rate that pushes $LBA(x)$ past value $LBS(out)$ before time $s + T$, but fails to push it past value $LBS(t)$ before time $t + T$. This means that some byte transmitted in $[s, t]$ times out. Determination of which byte Y , at which time X , is a matter of computing the intersection of two linear functions. The last trajectory (LBA3) corresponds to an acknowledged byte rate so low that LBA doesn't even achieve value $LBS(out)$ by time $t + T$. In this case, a time-out had to have occurred in an interval of time preceding $[s, t]$.

Figure 6. Geometry of Time-out Calculation

To support time-out calculations, we keep a time-ordered list of maximal epochs during which the data transmission rate was constant and, for each, the associated data transmission rates. The list spans the sequence of

bytes in the sender's window. Each epoch describes an interval like $[s, t]$ in Figure 6. As shown by analysis of Figure 6, for any epoch we can determine whether the first timed-out byte lies before, during, or after the bytes transmitted in the epoch of interest. Thus, the list can be searched efficiently (e.g. binary search if the list is long) to find the first timed-out byte Y in the window. A timeout event at time X can be scheduled to cause retransmission beginning at byte Y .

If it happens that a retransmission event is already scheduled to occur before time X , the timeout event is not scheduled.

3.4.9 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 $LBS(t) = LBA(t)$. 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 as defined in Table 2 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 $LBA(t)$ is embedded in the rate change, and the whole message is sent to the next device in the path.

4 DISTRIBUTED NETWORK OF INTELLIGENT SENSING MODULES

Sandia has expertise in real-time data acquisition, large-scale distributed information systems, and parallel simulation frameworks. The coupling of these technologies into a wireless distributed network of sensing modules has application to nuclear weapons as well as many other defense and industrial uses. Our LDRD year-end work was to create a proof of concept prototype Distributed Network of Intelligent Sensing Modules. We integrated intelligent sensing modules – sensors with integrated microprocessor systems and software – wireless hierarchical communication networks, and a distributed simulation framework into prototype design with immediate applicability to five program development areas at Sandia, including telemetry application, ubiquitous computing, health of weapon applications, wireless instrumentation/process control, and homeland defense/situational understanding. Our approach used a unique spread spectrum communication technique for wireless communication between modules – technologies under development in current LDRDs. Follow-on work will expand use of the network to other applications.

4.1 INTEGRATING TECHNOLOGIES

Our prototype design for intelligent sensing modules (ISMs) relies on an innovative combination of covert, noise immune spread spectrum techniques, wireless networking, real-time telemetry for integrated sensor systems, and a distributed software framework for data aggregation and visualization. ISMs are comprised of a physical sensor, embedded microprocessor, and distributed software component responsible for overall system control. This architecture does not impose restrictions on the type of physical sensor employed by each ISM; first demonstrations will include rudimentary JTA sensors. Since intercommunication is handled through wireless spread spectrum, it requires no physical wires, simplifying installation into a given weapon and eliminating the data loss associated with cable/connector failure in flight (Figure 7). Distributed components of the software framework present on each ISM monitor, interpret, and fuse dispersed sensor reports for centralized dissemination.

Figure 7. A collection of Intelligent Sensing Modules

Correlators fuse information from an interconnected ISM group and other neighboring correlators. This architecture then represents a hierarchy of intelligent sensing modules (Figure 8).

Each ISM and correlator module (i.e. node) is running a simulation of the physical system being sensed. The simulation is derived from an in situ database derived from ground-based high-fidelity models run beforehand on Sandia's supercomputers. A distributed information filter transforms the predicted data and data sensed from the actual environment into information. This information, in turn, is used by one or more intelligent agents residing on each node to make strategic decisions regarding the type and rate of data acquisition and priority of data being sent to archive. In this way, at a given data rate, the quality of information delivered to the analyst is increased by orders of magnitude. Agents may make decisions autonomously, but also communicate with one another to gain situational understanding of the entire sensed environment. Decisions are made collectively among the participating agents. Taken in aggregate, the software running on each of these nodes forms a model that can be used to understand the entire system. The type of sensor and model simulated by the ISM and correlator nodes is specific to the system being instrumented – sample application areas are documented below.

4.2 Creativity and Innovation

The spread spectrum approach to be employed is unique. In the commercial world, there are basically three uses for spread spectrum: cell phones, household cordless phones, and wireless networking. Of these, the cell phone environment is by far the most complex. Here, issues such as cell-to-cell handoff, transmitting power, and an undetermined number of users add a significant amount of overhead to the spread spectrum methodology used. This results in extremely complex chipsets and control techniques. Also, a certain amount of data loss can be tolerated before the user can no longer interpret the voice on the other end of the line. The wireless network environment is similar to a cell phone environment on a smaller scale. The emerging Bluetooth standard, which uses frequency hopping spread spectrum techniques, is designed to allow multiple personal electronics devices to talk to one another over a wireless link. Cell phones could talk to laptops, which could talk to printers, etc., all using the Bluetooth standard. Issues such as an undetermined number of data sources, networking standard compliance, and transmitting power all contribute to the complexity of the system. In a household cordless phone application, the spread spectrum techniques used are extremely simple because it is essentially a one channel system – the handset is the only device transmitting to the base unit.

Generally, the sort of software that we envision bringing to sensor arrays would not be found on embedded sensors, but more often associated with high-performance systems and large-scale distributed computing. Parallel simulations and intelligent agents are normally the province of large-scale distributed systems or supercomputers. However, the morphology and connectivity of these sensor arrays is strikingly similar to the systems to which this type of software is more commonly applied. The challenge here is to accommodate the lower processing power and network bandwidth that the sensor arrays will demand.

The key difference between the ISM architecture and the standardized wireless networking architectures such as Bluetooth is that these systems allow for an undetermined number of active users at any given time. This adds a significant overhead in communications protocol, both in terms of the information exchanged between devices and in the way spread spectrum is used for communication. Our prototype design is geared towards a finite and predetermined number of data sources located within a fairly small area. The number of data sources in a given system is determined by the length of the spreading codes used – the longer the codes, the more channels a particular system can support. The spread spectrum coding technique used eliminates inter-channel interference, which means each data source transmits at full power – no transmitting power adjustment is needed. Since a wireless instrumentation system based on this technology may be used in a JTA environment, data loss on the order of 1 bit per million is a strict requirement. Therefore, the system must be extremely robust.

The general hardware architecture of an ISM will include a sync pulse receiver and a data receiver. Because these two signals are transmitted using spread spectrum, a unique hybrid IC is necessary to decode/encode the data. Sensor data for a given module will be digitized and sent to a microprocessor, analyzed, then encoded and transmitted through proprietary spread spectrum techniques. The microprocessor will be capable of running software algorithms based on a common, commercial embedded operating environment. The glue logic

Figure 8. Hierarchical ISM Structure

synchronizes the microprocessor with asynchronous sensor data and incoming data from other modules in the ISM network.

4.3 PROPOSED APPLICATIONS

Unifying wireless, real-time data acquisition and large-scale, distributed information systems and parallel simulation into a unified, hardware/software system will have immediate benefit to five Sandia program areas (Figure 9).

Figure 9. Leveraging ISM Technologies throughout Sandia

Unifying these advanced research topics in radio frequency, electrical, and computer engineering provide a hardware/software platform for linking and integrating dispersed information assets into a coherent system for addressing national security threats. A distributed network of intelligent sensing modules guarantees Sandia's leadership in design and deployment of integrated threat detection systems. The following suggested application areas are provided as a brief overview of possible future Sandia investment areas.

4.3.1 TELEMETRY

Joint Test Assembly (JTA) flights are becoming both less frequent and more expensive. As a result, analysts want as much data as possible from a given flight. Due to the limited communication bandwidth between a JTA in flight and the receiving station on the ground, it is becoming increasingly difficult to provide data from each and every sensor on board a JTA. However, it is possible to gather much more data on board a JTA than can be transmitted to the ground. A Distributed Network of Intelligent Sensing Modules would gather large amounts of information from each weapon component, perform in flight data analysis and data fusion on this data, and provide analysts on the ground with a dataset containing more information using less communication bandwidth (see Figure 10).

Weapons analysts want more data than can be currently transmitted in telemetry applications. Vast amounts of data can be collected; however, getting that data to the ground in a JTA flight environment is becoming increasingly difficult. The solution is to "build the analyst into the weapon". Build an intelligent data acquisition system that compares sensor data to a high fidelity system model in real time. In this scheme, if a channel is performing out of spec, that channel's data, and possibly other appropriate data, is transmitted to the ground. If a channel is performing within spec, only a status bit is transmitted. It would also be possible for a system such as this to draw conclusions regarding the performance of certain parts of a weapon. For instance, if certain voltages in a fire-set do not reach their expected values, the system might conclude that a particular component is damaged.

Figure 10. Adaptive Telemetry Application

4.3.2 UBIQUITOUS COMPUTING

Distributed or clustered computing in support of high-performance system simulation has been effective within the bounds of hardwired connectivity. With the advent of spread spectrum communication and mobile computing technologies, it is now possible to realize high-performance computing in a mobile, distributed environment. For

example, advances in C-Plant evolution could be based on intelligent, wireless networking technologies developed through this proposal.

Further, traditional ubiquitous computing applications – such as “the information integrated hospital”, or home automation – will benefit from the integrated wireless communication and processing power demonstrated through this proposal.

4.3.3 HEALTH OF WEAPON APPLICATIONS

Already under consideration in Org. 8400 is the Weapon Embedded Instrumentation System (WEIS) (Figure 11), a wireless network of intelligent sensing modules would be built directly into a weapon, providing state of health information during fabrication and over the life of the weapon in addition to providing real-time JTA data.

Weapon Embedded Instrumentation (WEIS)

- When a weapon is in storage, the intelligent network will wakeup once a day (or more) to collect state of health (SOH) data from various parts of the weapon. Periodically, the SOH data can then be downloaded from the weapon and examined.
- When the network is dormant, it consumes microwatts of power. This will extend battery life out to several years.

- If a weapon is assigned to a flight test, the network will automatically detect that a launch has occurred, and switch to a real-time data acquisition mode of operation. In this mode, the necessary telemetry format to score the weapon is generated and transmitted to the ground as in all JTA flights.

Figure 11. Weapon Embedded Instrumentation (WEIS)

4.3.4 INDUSTRIAL APPLICATIONS

Researchers at Sandia's Combustion Research Facility and in the EUVL Program have identified several areas of application for this technology. Steel production, automotive manufacturing, automotive instrumentation, and lithography equipment environments are several examples. Manufacturing environments in particular can benefit from this technology for several reasons. These facilities are often very large, and a traditional (wired) instrumentation system could require miles of cabling to bring the information from the floor to the control room for interpretation. In addition, there are often very high levels of EMI and RF noise present in these environments. An automotive instrumentation environment is similar to the manufacturing one; however, the instrumentation modules themselves must be very small, low power devices. An instrumentation system utilizing spread spectrum is ideal for these types of applications. Because it is wireless, cabling is no longer an issue, and the very nature of spread spectrum would make the system extremely immune to electrically noisy environments.

4.3.5 SITUATIONAL UNDERSTANDING

By integrating sensors, computation, and distributed real-time data acquisition, systems can be developed to support corps level data fusion (see Figure 12). Possible applications within defense and national security are many. For immediate demonstration, the prototype distributed network of intelligent sensing modules could be used for homeland defense, with linkage to prototype chemical/biological sensors. Other possible application areas include NEST searcher training, battlefield threat interpretation, and operational support of emergency response to threats to national security.

Figure 12. Situational Understanding – Homeland Defense Applications

REFERENCES

- [1] J.S. Ahn and P.B. Danzig. Packet network simulation: Speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking*, 4(5):743–757, October 1996.
- [2] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):30–38, January 1999.
- [3] G. Kesdis, A. Singh, D. Cheung, and W.W. Kwok. Feasibility of fluid-driven simulation for ATM networks. In *Proceedings of IEEE GLOBECOM*, volume 3, pages 2013–2017, November 1996.
- [4] K. Kumaran and D. Mitra. Performance and fluid simulations of a novel shared buffer management system. In *Proceedings of IEEE INFOCOM 98*, March 1998.
- [5] B. Liu, Y. Gao, J. Kurose, D. Towsley, and W. Gong. Fluid simulation of large scale networks : Issues and tradeoffs. In *Proceedings of 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999. To appear.
- [6] A. Yan and W. Gong. Fluid simulations for high speed networks. *IEEE Trans. on Information Theory*, June 1999. To appear.

DISTRIBUTION:

1	MS 0188	C. E. Meyers	1030
1	MS 0188	D. L. Chavez	1030
1	MS 9102	D. E. Rhodes	8416
1	MS 9201	H. R. Ammerlan	8112
1	MS 9201	L. D. Brandt	8112
1	MS 9201	P. K. Falcone	8114
1	MS 9201	M. E. Goldsby	8114
1	MS 9201	M. M. Johnson	8114
1	MS 9201	A. S. Yoshimura	8112
3	MS 9018	Central Technical Files, 8940-2	
1	MS 0899	Technical Library, 4916	
1	MS 9021	Classification Office, 8511/Technical Library, MS 0899, 4916	
1	MS 9021	Classification Office, 8511 for DOE/OSTI	
1	MS 0161	Patent and Licensing Office, 11500	