Network Simulation

Network Simulation

Richard M. Fujimoto, Kalyan S. Perumalla and George F. Riley

www.morganclaypool.com

ISBN: 1598291106	paperback
ISBN: 9781598291100	paperback
ISBN: 1598291114	ebook
ISBN: 9781598291117	ebook

DOI 10.2200/S00046ED1V01Y200609CNT001

A Publication in the Morgan & Claypool Publishers Series
SYNTHESIS LECTURES ON COMMUNICATION NETWORKS
Lecture #1
Series Editor: Jean Walrand, University of California - Berkeley

First Edition

10 9 8 7 6 5 4 3 2 1

Network Simulation

Richard M. Fujimoto
Georgia Institute of Technology

Kalyan S. Perumalla
Oak Ridge National Laboratory

George F. Riley
Georgia Institute of Technology

SYNTHESIS LECTURES ON COMMUNICATION NETWORKS #1

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

A detailed introduction to the design, implementation, and use of network simulation tools is presented. The requirements and issues faced in the design of simulators for wired and wireless networks are discussed. Abstractions such as packet- and fluid-level network models are covered. Several existing simulations are given as examples, with details and rationales regarding design decisions presented. Issues regarding performance and scalability are discussed in detail, describing how one can utilize distributed simulation methods to increase the scale and performance of a simulation environment. Finally, a case study of two simulation tools is presented that have been developed using distributed simulation techniques. This text is essential to any student, researcher, or network architect desiring a detailed understanding of how network simulation tools are designed, implemented, and used.

KEYWORDS

Computer networks, Network design, Network emulation,
Network performance and scalability, Network simulation, Network simulation tools,
Parallel and distributed simulation (PADS), Wireless networks

Contents

CHAPTER 1

Introduction

Computer networks have become central to almost all computing and information processing applications. The arrival of the Internet into daily life has greatly increased the relevance and importance of computer networks. More and more applications rely on efficient and reliable network operation, such as the world wide web, email, instant messaging, Voice Over Internet Protocol (VOIP), and network-based audio and video. Networks such as the Internet and individual corporate networks have become as essential as to qualify as critical infrastructures for home, work, and government. Ensuring proper operation of the networks is an extremely important task, as also the task of properly designing and improving the network infrastructure. Simulation is an indispensable method in designing and improving networks. Network simulations are now routinely used in all aspects of network design, configuration, and analysis.

1.1 WHY SIMULATE?

Over the past few decades, computer networks have evolved into complex systems, making their analysis very challenging. While analytical techniques (e.g., using queuing network theory) have been adequate in early stages, the complexity and speed of network hardware and the complexity of protocols have made simulation necessary for analyzing realistic scenarios of modern network operation. Network simulation is now used for various aspects of research and development of networks, including protocol analysis, verification, and understanding; evaluating futuristic network technologies/architectures; software-based surrogate networks; online optimization; vendor distinctions in acquisition testing/evaluation; and scenario analysis of deployment configurations. Some of the uses of network simulation can be categorized as follows:

- *Surrogates for real networks*: Real networks are difficult to instantiate (purchase, install, and configure) in order to experiment with scenarios, especially for large network configurations. Moreover, it is hard to create desired network conditions in real networks for controlled experimentation, e.g., with network traffic loads and congestion patterns of interest.
- *Early prototypes of nonexistent or futuristic network technologies*: Often, new network protocols are proposed, such as new Transmission Control Protocol (TCP) variants or

new multicast protocols. The designers of the protocols not only need to analyze the protocols' strengths and weaknesses but also need to convince themselves and the others that the proposed protocols would work as well or better than other existing protocols before venturing into developing real deployments. Similarly, new networking hardware capabilities (such as faster links or interconnects) need to be tested in configurations of interest. Simulation affords a way to explore different loading conditions to better design and optimize the hardware.

- *Duplicates of existing networks*: In certain situations, simulation is useful for re-creating scenarios in order to verify certain theories/models or to better understand certain phenomena. For example, simulation is useful to reconstruct Internet worm attacks in simulation in order to learn more about their operation and their sensitivities to network topology and traffic conditions. Since it is not possible to re-enact such malicious traffic on the real network, network simulation serves as a software-based duplicate of existing networks.

1.2 NETWORK SIMULATOR CAPABILITIES

A range of network simulators have been developed over time, and a variety of simulators exist, each specializing in certain features. Some of the differentiating capabilities among these simulators are as follows:

- *Usability*: availability of graphical user interface (GUI), the richness of model repositories, and animation capabilities.
- *Level of extensibility*: ability to create new models or extend existing models and analysis tools.
- *Customization mechanisms*: availability of scripting features or graphical configuration methods.
- *Implementation approaches*: the way the simulator is realized and used, e.g., as network simulation languages or simulator object class libraries.
- *Speed of execution*: whether the execution is sequential (using one processor) or if parallel/distributed execution is supported (using multiple processors for a single simulation run).
- *Scalability with size*: how scalable the simulator is with respect to the number of network nodes and the number of simultaneously active flows in the simulated network.
- *Emulation support*: whether the simulator can also be used to act as a surrogate for actual networks.

- *Wired/wireless*: the ability to simulate wired networks, wireless networks, or both.
- *Model diversity*: the availability of predefined models for commercial networking products (e.g., specific Cisco router products).
- *Fidelity ranges*: the level of detail supported in the models, such as detailed packet models or aggregate fluid models.
- *Level of support*: whether the simulator is a commercial product or relies on an open source base.

The choice of features depends on the intended use of the network simulator. For example, network administrators who are only interested in assembling models of network products in order to evaluate installation alternatives might be interested in simulators with established repositories of models. Network researchers interested in developing new protocols, for example, would be more interested in the ease with which new functionality could be added to the models. Advanced researchers intending to evaluate large-scale effects, such as peer-to-peer networks or distributed denial of service attacks, would benefit from simulators that can sustain very large configurations of the simulated networks (e.g., with hundreds of thousands of nodes). A novice user would benefit from powerful graphical interfaces to help assemble and execute configurations by simple mouse clicks.

Some of the popular simulators include OPNET, Qualnet, *ns*-2, GTNetS, SSFNet, and NCTU*ns*.

1.3 WHAT IS SIMULATED?

Depending upon the way in which the network simulation is used, there are various aspects of interest that are simulated. Consider questions such as: "Is a TCP variant called A better than another TCP variant B?" "What is the expected latency of object discovery with a given peer-to-peer file sharing protocol?" "How well does a given multicast protocol scale in complex network topologies with many senders and receivers?" "What is the level of jitter experienced by a video stream over a certain wide area network in a distributed video conferencing application?" Typically, simulation can be used to answer such questions with a carefully designed set of one or more simulation-based experiments. Network simulators facilitate this task by providing a framework in which the desired network configurations can be assembled virtually (in software) and virtual traffic loads can be introduced over the network (e.g., TCP and User Datagram Protocol (UDP) traffic flows across the networks), and measurements can be taken without perturbing the system (e.g., achieved TCP throughput measures for a flow, or the level of utilization of a network link, or delay statistics experienced by an application). Clearly, the level of detail at which the network must be simulated depends entirely on the nature and accuracy of results that are desired as output from the simulation. For example, if a very large portion of end-to-end latency for an application depends on the wide area network delays (e.g., dozens of milliseconds), it is safe to skip simulating the small latency effects introduced by the Ethernet protocol of the local area networks to which the end hosts are attached and instead focus on simulating the wide area routers and links. However, in wireless networks, it is important to accurately capture the effects of medium access protocols as they have significant bearing on delay and loss characteristics. In situations that demand an understanding of strengths or limitations of certain models, it is important to understand the implications of fidelity and how best to interpret the results from simulation of models of one's choice.

CHAPTER 2

Network Simulators

In this chapter, we describe the basic software requirements for a typical network simulation tool and discuss design tradeoffs. We will use an existing simulation tool, the Georgia Tech Network Simulator (GTNetS) [1], and present several of the design decisions given in that tool.

2.1 SIMULATION ARCHITECTURE

Nearly all current network simulation environments use *discrete-event* simulation methods to simulate the behavior of a network. This simply means that we are only interested in the state of the system being simulated (the network) at discrete points in time and can safely ignore the state of the system at times in between those discrete points. As an example, consider how to model the act of a router sending a network packet on a wired point-to-point link to another router connected to the same link. In the actual physical system, this seemingly simple act is in fact quite complicated, including encoding the individual bits into electrical or optical waveforms, calculating error detection and correction information, transmitting these waveforms on the actual communications link, detecting the waveforms at the receiving end, verifying the correctness of the received signal, and reconstructing the packet at the receiving router. However, for our simulator, we can model the overall behavior of this activity by simply noting that the transmitting router starts sending the packet at some time T and the packet is completely received by the receiving router at time $T + \Delta T$. ΔT in this case simply represents the time it takes the sender to transmit all of the individual bits in the packet, plus the signal propagation delay (which is the speed-of-light delay in optical networks) along the transmission link to the receiver. The state of the system at any time during the interval ΔT of course varies in the physical system, but can be safely ignored in our simulator. It is easy to see that, by using this approach, we only need to change the state of the system twice in this example: once when the transmitter has completed sending the packet and once when the receiver has received the entire packet. Each state change is realized as an event in discrete-event simulation.

Using discrete-event simulation methods as described previously, the basic design of the "main loop" of a network simulator becomes nearly trivial. The needed actions and data structures are as follows:

1. A variable containing the current value of the time T, nominally called the *simulation time*. This variable represents the time at which the current state of the system is known and represented in the simulation environment. It is initialized to an arbitrary value, usually zero, and advanced in a nondecreasing fashion as the state of the system progresses.
2. A list of *pending future events*. This list contains information about state changes that have not yet occurred, but are known to have been scheduled to occur in the future. An example is the packet transmission activity discussed earlier. When a transmitting router starts a packet transmission action at time T, it schedules two future events in the pending event list. First is an event at time $T + \Delta T_1$ to indicate that the transmission has completed, and all bits have been sent on the link. This is an indication that the link is then free to send another packet if one is available. The time in the future when an event will occur ($T + \Delta T_1$ in this case) is known as the *timestamp* of the event. The second future event at time $T + \Delta T_2$ indicates that the packet will be received at the receiving router, which necessarily occurs after the packet transmission has completed, due to signal propagation delays on the link. It is important to note that the state changes in the network due to these two future events are not necessarily known, nor can they be determined at time T, when the packet transmission starts. The state of the system at time T is of course different than it will be at time $T + \Delta T_1$ when the transmission is complete, or $T + \Delta T_2$ when the packet is received. For efficiency, the list of pending future events is maintained in sorted order of ascending time. The reason for this will become apparent.
3. Once we have a simulation time variable and a sorted list of pending future events, the main loop of the network simulator becomes:
   a) If there are no more pending future events, terminate.
   b) Remove the earliest pending future event from the list.
   c) Set the current simulation time to the timestamp of the just removed event.
   d) Process the state changes that occur due to the event action. These state changes might include the creation of one or more additional pending future events. For example, if the event is the receipt of a packet at a router, that router might immediately transmit the packet on a different link and schedule the corresponding packet receipt event at the next hop router.
   e) Return to step (a).

It is now apparent why the pending event list is maintained in ascending timestamp order. The action in step (b) above is simply removing the first entry in the sorted list. If we did not maintain this list in sorted order, we would have to examine every event in the list to find the earliest. Of course, keeping the list in sorted order results in extra overhead when adding events to the list, but there are several ways to efficiently insert items into a sorted list.

2.2 SIMULATOR STATE OBJECTS

The preceding discussion describing the management of simulation time and the pending event list is applicable to any discrete-event simulator. In this section, we discuss in more detail the design of a network simulator and the types of simulation state objects needed to model the state and actions of computer networks.

Ideally, the design of any simulation environment should match as closely as possible the design and implementation of the corresponding real-world system being simulated. For example, a computer network consists of a set of nodes, which are the end systems and routers, network interface cards (NICs), links, queues, protocols, data packets, and applications. Designing a simulation environment with a state object for each of these corresponding real-world objects leads to a good design that is easy to understand and extend as needed.

2.2.1 Node Objects

For this discussion, the term "node" refers to any network element that either receives or transmits a packet. This might be an end system, such as a desktop personal computer, a high end web server, a router in the interior of the network, an optical switch, or a simple Ethernet hub. All of these devices can either create and transmit new packets as in the case of web servers and desktop PCs, or receive and forward packets as in the case of routers and switches. The state needed to represent a node in a simulation environment will usually include the following.

A protocol graph: When packets are received by a node (usually through a node "interface," as described next), action must be taken to determine what is to be done with the packet. The well-known open systems interconnection (OSI) protocol stack model is a generally accepted method for determining how to process packets that arrive at a node. A protocol graph is a logical representation of each layer in the OSI protocol stack, with an indication at each layer describing a process or subroutine that will handle the packet at that layer. For example, at layer 2 in the stack (called the media access control layer or MAC layer), there might be processors for several different types of transmission media, such as wired point-to-point links using IEEE 802.3 and wireless links using IEEE 802.11. The appropriate processor addresses will be stored in the protocol graph, and the correct one will be selected based on the type of packet being received (wired or wireless in our example). Similarly, once the MAC layer completes processing, the appropriate layer 3 (called the network layer) processor will be called. Again, there may be several possibilities, such as IP version 4 and IP version 6.

A port map: At some point in the processing of a received packet, the protocol stack will determine whether the packet is addressed to this node (such as a connection request to a web server) or is to be forwarded on an output link toward some other node (in the case of packets received by routers and switches). In the first case, further processing of the packet is needed to determine which end application is to ultimately receive the packet. This is generally accomplished by the use of a port map. The port map is a data structure that lists every end application at a node (for example, a web server application) and which port numbers are associated with that application. There may be separate port maps for the various layer 4 (transport layer) protocols, such as TCP and UDP. In real networks, as well as in simulated networks, there are often several applications at any given end node sending and receiving packets simultaneously. The port map allows for de-multiplexing of the received packets to the appropriate process or application for further processing.

Resource usage: During protocol processing and other operations, various resource usage metrics can be tracked in the simulator. For example, in the case of hand-held or low-power sensor devices, the energy consumption of various actions and the network lifetime of the devices is often a concern, which makes it important to model the battery power consumption accurately. There are a number of ways for a simulation environment to monitor the energy used by the devices as packets are received and transmitted. Usually a single state variable that is used to track the remaining energy in a battery is sufficient for this activity. Other resources of interest might include memory consumption levels and CPU time usage.

Physical location: In the case of wireless ad hoc networks, the physical location of a node, either in two-dimensional space or three-dimensional space, plays an important role in the behavior of the node and the overall network.

A list of "network interfaces": Network nodes have one or more "network interface cards" (NICs) that are responsible for the low-level transmission and reception of packets on a transmission medium. These might be Ethernet controllers, fiber optic interface cards, or wireless devices. The detailed state for each interface card should be maintained in a separate simulator object, described next.

2.2.2 Interface Objects

As mentioned in the previous section, network nodes are concerned with receiving and transmitting packets, and therefore simulation objects representing nodes are designed with that in mind. However, the details of how a packet is sent from one node to another are generally transparent to a network node. Activities such as controlling access to a communications medium, creating electrical or optical signals to represent a 1 or a 0 on the medium, determining the start or end of a packet, and error detection are relegated to a "network interface." Similarly, a simulation environment should clearly separate the actions of processing packets and transmitting or receiving packets. To that end, a simulation object representing an "interface" needs to maintain several state variables, as follows.

A "busy" indication: At any point in time, an output interface is either in the process of sending a packet or is idle, waiting for a packet to send. In the former case, the medium is "busy" and cannot start another transmission. Usually, a simple "busy" flag is sufficient to model this.

A packet queue: Since packet processing routines in a node are unaware or unconcerned with actually managing a communications medium, they will naturally be unaware of whether a packet can be transmitted at any point in time. Thus, an interface object must be able to accept requests from node objects to transmit a packet, even when the medium is busy transmitting a previous packet. Each interface should have a mechanism for storing packets in a queue and retrieving them from the queue at a later time. The state needed by queue objects is discussed later.

A communications link: In actual networks, each network interface has a single corresponding communications link. This might be a simple point-to-point link, a broadcast-based local area network (such as the ubiquitous Ethernet), an optical fiber, or the wireless spectrum. Similarly, a simulated network interface object must have some notion of the associated link. Details of the link object are given next.

2.2.3 Link Objects

The primary function of a link object in simulation environments is to specify the physical connectivity of the network elements. Conceptually, one can think of a simulated network topology as a directed graph, with network node objects representing the vertices in the graph and the links representing the edges. Therefore, simulator link objects need state to represent the connectivity of the network.

Neighbor list: Of primary importance is an indication of which other network nodes are directly connected neighbors of the node associated with a link. For simple point-to-point links, there is a one-to-one correspondence. For Ethernet LAN networks, each node has a (almost static) set of neighbors. For wireless devices with mobility, there is a nonstatic set of neighbors that changes over time. In any case, the link object must maintain a list of directly reachable neighbors.

Link delay: Actual communications links are constrained by speed-of-light delays on the medium. A bit leaving one end of a link will not arrive at the other end until some nonzero time has elapsed while the electrical or optical signals propagate on the medium. Therefore, link objects in a simulation environment need some method to determine this propagation delay value. In simple point-to-point links, a single constant value can be used. For multiple access LANs, a set of delays may be needed. In some cases, particularly in the case of wireless networks, delay can be calculated based on node location.

Link bandwidth: The rate at which data can be sent on links affects the overall performance of a network. Typical values for link bandwidth are either 10 Mbps or 100 Mbps for Ethernet LANs and 2 Mbps, 5 Mbps, or 11 Mbps for wireless networks. Although link bandwidth is often a function of interface connecting to a link (for example, the selection of 10 or 100 Mbps for an Ethernet is determined by the interface card), typically a simulation environment will associate data rates with a link.

Also note that almost always, wired communication links in actual networks are full-duplex, meaning that data can be send in both directions (from A to B and from B to A) simultaneously. This is commonly implemented in a simulator by two half-duplex links, one in each direction.

2.2.4 Queue Objects

Any packet-based network needs some way to buffer packets temporarily in response to transient congestion at output links. In fact, a large body of research exists studying effects of queuing on network performance, with focus on various methods for "fair" queuing and achieving quality of service (QoS) guarantees for data connections in the presence of queuing. Similarly, simulation environments must represent this buffering action in a realistic way. The implementation of a simulation object representing a packet queue is particularly well suited for object-oriented programming languages such as C++ or Java. In essence, any queue simply needs to be able to insert a packet into the queue and retrieve it at a later time. The method by which this is accomplished varies widely depending on the type of queue being used. A simple drop-tail queue will implement a first-in, first-out algorithm and unconditionally drop packets being inserted when the size of the queue reaches a predetermined limit. More elaborate queues can enforce loss or delay limits for certain classes of packets, ensure that certain amount of bandwidth is reserved for higher priority packets, and can preferentially drop packets for connections using an unfair proportion of the link capacity. Regardless of the design of how a queue manages packets, the basic interface for adding and removing packets remains the same. Typically, a simulator will provide some basic design for queues (for example, an abstract base class in C++) that describes the en-queuing and de-queuing actions, and will provide various implementations (in the form of C++ subclasses) that enforce the desired queue behavior.

In general, the state needed for queue objects is as follows:

Queue size limit: Virtually all queuing methods provide some limit on the total number of packets (or bytes) that can be held in the queue at any point in time. If this were not the case, a queue for an oversubscribed link would grow to infinite size and exhaust all available resources in the router or the simulation environment.

Packet list: The queue must be able to retain packets (often in a linked list or other dynamically managed container) and retrieve them for later de-queuing actions. The actual implementation of the packet list is dependent on the type of queue being modeled.

2.2.5 Protocol Objects

The design and deployment of protocols is fundamental to virtually all modern communications networks. The OSI model defines seven unique layers of protocols, depending on the functionality being implemented by a given protocol. For example, "link layer" protocols (layer 2 in the OSI model) are used to provide error detection and correction, identification of the intended next-hop recipient for a packet, and to identify the sender of a packet. "Network layer" protocols (layer 3 in the OSI model) are designed to identify the ultimate destination for a packet and to allow ways to forward the packet along the path (using layer 2 protocols) to the desired end system.

As in the case of packet queues, the design of protocol objects in a simulation environment is particularly well suited for object-oriented design and implementation. There are clear and well-defined interfaces between protocol stack layers that must be defined and implemented to be a given protocol. However, the actions taken by the protocol in response to calls to these interfaces can of course vary depending on the design of the protocol. In general, a given protocol must be able to respond to requests from a higher layer to process a packet, which is canonically called a "data request." Also, protocols must act on requests from lower layers to process packets, which is called a "data indication." For example, the link layer protocol for an IEEE 802.3 Ethernet network must be able to accept packets from the corresponding layer 3 protocol (IP V4 for example) via a data request call and forward those packets on a particular link to the next-hop recipient. When doing this, the layer 2 protocol will append addressing information to the packet, compute cyclical redundancy check (CRC) information, and transmit the packet on the network. Similarly, when receiving a packet, the layer 2 protocol will verify that the next-hop destination address is correct, validate and correct any bit errors detected, and forward the packet to the next higher layer protocol via a data indication call.

Clearly, the state information needed by protocol objects in a simulation environment varies widely depending on the type of protocol being implemented. Complex layer 4 (transport layer) protocols such as TCP have a substantial amount of state including sequence number received, ack number sent, round-trip time estimation, source and destination port numbers, and many others. A layer 2 protocol implementing the IEEE 802.11 wireless access protocol also has substantial state information, including retransmission information, timeout values, base station associations, and more.

2.2.6 Packet Objects

Also fundamental in the design and implementation of modern computer networks is the notion of packets. When sending a large volume of data from any given end system to another, the data is broken down into a number of smaller, often fixed-sized, units (packets), and the packets are sent on the network one at a time. On any given communications link, packets from differing end systems are interleaved, leading to effective utilization and sharing of fixed-bandwidth links.

Fundamentally, a packet contains the data being sent on the network. For example, the data might be the contents of a web page or a music file being downloaded. However, the design specified by the OSI protocol stack requires each layer include additional information specific to that layer in the packet. For example, a layer 2 protocol implementing the IEEE 802.3 standard for Ethernet LANs will add 16 additional bytes to a packet before transmitting it. There are two 6-byte fields specifying the source and destination "media access control" (MAC) addresses and a 2-byte field indicating the total length of the packet. Similarly, a layer 3 protocol implementing the IP V4 standard will append a 20-byte header containing the various information items required by IP. The information added to packets at each protocol stack layer is canonically known as a "protocol data unit" (PDU).

Therefore, the representation of a packet in a simulation environment is logically a collection of protocol data units, plus the actual data being transmitted. Several implementations for this collection of PDUs are possible. A simple first-in, first-out queue, such as the functionality provided by a standard template library (STL) vector, can be used. Alternately, a union or structure can be defined that includes data fields for all possible PDUs. In either case, the information in the packet can be added or removed as needed, as a packet is processed in the simulated network.

In a network simulator, the actual contents of the data being sent are often abstracted away. For example, if one is concerned with the behavior of the TCP protocol in the presence of congestion, then the actual contents of the data in a TCP packet makes no difference in the performance of the protocol. As long as a packet has 1000 bytes of data (for example), and as long as the simulator treats the packet as if it contained 1000 bytes of data, the actual data can be ignored. This abstraction of data can lead to substantial saving in memory and resources in the simulator.

2.2.7 Application Objects

In a network simulation, as well as in an actual network, the end applications provide a data demand for the network. In the current Internet, the types and data demands by applications are widely varied. Peer-to-peer file sharing applications move large volumes of data between a pair of end points, but with minimal time constraints. Web browsing applications transfer a number of small objects such as hypertext markup language (HTML) scripts and images. Voice-over-IP (VOIP) applications transfer time-critical but small and low data rate packets. Interactive time-sharing sessions, such as Telnet and Secure Shell (SSH), transfer sporadic bursts of information in response to keystrokes. It is easy to see that the types and mix of applications in a simulation environment will have a large effect on overall network performance.

There are several ways to construct realistic application models in a simulated environment. One way is to define the behavior of the application in analytical terms. For example, consider the modeling of a VOIP application with data compression and silence suppression. Logically, such an application would generate data at some random rate (depending on how much compression is possible instantaneously) for a random amount of time (depending on how long a person talks before listening). The application would then remain silent for a random amount of time, depending on how long the user listens before speaking again. A simulation model for this application could easily define random variables for the "on" and "off" times, as well as a random variable for the data rate during the on interval. This approach would result in a good model of the actual data demand for a VOIP application, assuming that reasonable values are chosen for the random variables used.

Another approach is to monitor an operational network and construct empirical distributions for various application metrics based on the observed behavior of actual applications. For example, the metrics for a web browser might be the average size of a web page request, the average size of a response, the average number of response, the amount of "think time" between requests, and the number of times a given server is accessed consecutively. Once the operational network has been monitored for a sufficient length of time, empirical distributions can be constructed to match the observed metrics. The simulation model can then simply sample the empirical when requesting a web page, and sample the distribution when constructing a reply. This approach leads to a realistic model of application performance, albeit limited to the performance of that application on the network used for monitoring.

It is sometimes possible to include the actual application software itself in the simulation software. This approach is most useful when the application does not depend on human users for inputs and responses. An example might be a routing protocol, such as the open shortest path first (OSPF) protocol. This protocol sends periodic information regarding network topology to peers and updates this information to others upon receiving it. In this case, the software implementing the protocol could be embedded in the simulation environment, and would therefore produce the sequence of data exchanges in a fashion identical to the actual protocol.

2.2.8 Support Objects

An important design feature of any network simulation environment is the ability to collect statistics about the performance of the network being modeled. A common approach is to provide the ability to create a "log file" for every packet transmission and receipt in the simulated network. While this approach is flexible and provides virtually all information that might be needed, it is not without problems. First is the fact that these log files might become excessively large, since network simulations may handle billions of packets throughout the life of the simulation. Secondly is the need for postanalysis of the log file, since individual packet actions are rarely of interest. Rather, end-to-end performance metrics such as throughput or delay are more commonly needed. Finally, it is often the case that only a subset of all packets and flows in simulation are of interest, with the remaining packets used only to provide competing flows.

Another approach to statistics collection is to provide instrumentation within the various simulation protocols and applications to collect the metrics of interest. For example, a web browser object might monitor the average response time for all flows used by that browser and produce a probability density function (PDF) or cumulative distribution function (CDF) at the completion of the simulation. A VOIP application could track and report the number of missed voice packets or the number of packets that arrived too late to be of use for the play-out buffer.

Finally, generic histogram or averaging objects could be connected to any application or protocol object to track the performance of the particular object. For example, a TCP protocol object could track the number of retransmissions for each data segment and produce a histogram of retransmission counts.

2.3 AN EXAMPLE SIMULATOR

There are a number of public domain and open source network simulation environments available. Undoubtedly, the most popular and widely used tool is the *ns*-2 [2] simulator developed and maintained by the Information Sciences Institute (ISI) at the University of Southern California. The *ns*-2 tool uses a combination of the C++ language along with a scripting language called "tool command language" (TCL) to specify the simulation topology and applications. The "Georgia Tech Network Simulator," (GTNetS) [1] developed and maintained by the Department of Electrical and Computer Engineering at Georgia Tech is an object-oriented design written completely in C++. The design of GTNetS matches closely the design of actual network protocol stacks and other network elements. Further, GTNetS was designed from the beginning to run a distributed environment, leading to better scalability. GTNetS will be used below to illustrate the construction and execution of a network simulation. The SSFNet simulator was developed jointly by Rutgers University and Dartmouth University. It is written primarily in Java, with a C++ variant available. The specification of the topology and applications in SSFNet is accomplished using an XML-like text-based language called "graph modeling language" (GML). SSFNet was designed primarily for multiprocessing environments, utilizing multiple threads to give efficient execution of small-scale topologies.

The GTNetS simulator will be used as an example of the steps needed to create a network simulation and monitor the results of the simulation. As mentioned, GTNetS is written completely in C++ and provides models in the form of C++ objects for many popular network elements. To construct a simulation using GTNetS, a user simply creates a C++ main program that instantiates the necessary objects to describe and execute the simulation. The program is compiled with any standards-compliant C++ compiler and linked with our libraries containing the simulation models. The resultant binary is executed to run the simulation. The details of the compilation and link processes are omitted here.

We will give a simple simulation scenario in GTNetS as an example of the steps that must be performed to construct and run a simulation. While we specifically use GTNetS as our example, the steps are conceptually similar to those needed for other simulation tools. The example scenario is a simple four-node topology, with one server, one TCP sending application, and two routers. The scenario sends 5 million bytes of data and reports the overall throughput of the TCP connection. Default values for link delays and link bandwidth are used for simplicity.

2.3.1 Create a Simulator Object

In GTNetS, the overall control of the simulator is implemented in an object called simulator. All GTNetS must have exactly one simulator object, defined as follows:

```
// Create the simulator object
Simulator s;
```

This declaration can either be a global variable or a local variable in the scope of the C++ main function.

2.3.2 Create Nodes

Node objects in GTNetS are represented by objects of class Node. Node objects can either be created as local variables in the scope of the C++ main function or as dynamic objects created with the C++ new operator. The example below uses the latter approach.

```
// Create the nodes
Node* sender = new Node(); // TCP Sending application
Node* server = new Node(); // TCP Server application
Node* r1     = new Node(); // Router 1
Node* r2     = new Node(); // Router 2
```

2.3.3 Create Links

In GTNetS, there are several different types of link objects depending upon the type of link being modeled. In this scenario, we want a simple point-to-point link, representing perhaps a single wavelength on an optical link. There is a convenient shortcut function in Node objects for creating point-to-point links called AddDuplexLink. The AddDuplexLink method specifies a node object for a remote end point and an optional second parameter specifying an IP address for the local end point.

```
// Create the links
sender->AddDuplexLink(r1, IPAddr("192.168.1.1"));
r1->AddDuplexLink(r2);
server->AddDuplexLink(r2, IPAddr("192.168.2.1"));
```

2.3.4 Create Applications

Since applications are always associated with Node objects, applications are created and managed by nodes. The Node method "AddApplication" is used to create a new application running on a specific node. This method returns a pointer to the newly created application object, which can be used to further define or refine the application behavior.

```
// Create the applications
TCPServer* ts =(TCPServer*)server>AddApplication(TCPServer());
ts->BindAndListen(1000); // Accept connections on port 1000
ts->Start(0);            // Start the application at time 0
TCPSend* tsv = (TCPSend*)sender->AddApplication(
   TCPSend(server->GetIPAddr(), 1000, Constant(5000000)));
tsv->Start(0);           // Start the application at time 0
```

The Constant() parameter specifies that the application should send exactly 5,000,000 to specify random amounts of data using the various random variables, such as Uniform or Exponential.

2.3.5 Run the Simulation

At this point, the topology and the data demand for the network are created, and we are ready to begin scheduling and processing events. This is accomplished by the Run method in object class simulator.

```
// Run the simulation
s.Run();
```

2.3.6 Print the Results

For this extremely simple scenario we are only interested in the overall TCP throughput (sometimes called the "goodput") for the single TCP data flow. In GTNetS, all TCP flows track the measured throughput, which can be accessed using member function GoodPut of object class TCP.

```
// Print goodput statistic
std::cout << "The TCP Throughput is "
          << ((TCP*)(tsv->GetL4()))->GoodPut()
          << " bytes per second" << std::endl;
```

2.4 SUMMARY

The scenario presented above is obviously extremely simple, but is instructive nonetheless. Actual scenarios are likely to have much more complicated topologies, with more different link types and capacities, different applications, link failures, statistics, and measurements. However, the basic steps are the same, as laid out above.

CHAPTER 3

Wire-Line Network Simulation

We will now focus on some of the simulation issues and challenges specific to wire-line networks (see Fig. 3.1).

3.1 MODELING ALTERNATIVES

In general, there is a range of modeling alternatives for simulating wired networks. The approaches include hardware testbeds, emulation systems, packet-level models, mixed abstraction models, and fluid models. Each approach has its own strengths and weaknesses, affording different combinations of scalability of execution and fidelity of models.

Scalability of a method is defined as a limit on the number of network nodes modeled by that method. Network nodes include end hosts and routers. Fidelity in general is hard to define, but it is possible to compare two methods with respect to their relative fidelity. Fidelity could be based on the amount of detail accounted for in the network (e.g., routing, network congestion, etc.), or in end hosts (e.g., stack processing, operating system overheads, etc.). The range of modeling alternatives for network simulation is depicted in Fig. 3.1, ordered according to their levels of fidelity and scalability.

Hardware testbeds have improved in scale with recent advancements, with network testbeds scaling to hundreds or more nodes (e.g., EmuLab). However, hardware testbeds cannot by themselves sustain fidelity with increasing scale. In fact, hardware testbeds resort to some form of network simulation underneath to improve fidelity when virtual configurations exceed physical resources in size (e.g., to emulate link delays or losses).

The next level of scalability is achieved via network emulation. Emulation systems for network analysis have also scaled in size, with recent emulators capable of sustaining a few thousand nodes (e.g., NetLab and ModelNet). Both hardware testbeds and emulation systems are by definition executed in real time.

The next logical alternative is packet-level simulation. Historically, network simulation experiments have always been done on a small scale and the results of such experiments have been extrapolated to derive conclusion on large-scale simulations. However, results on a smallscale are hard to extrapolate to larger configurations and hence can be misleading. Large-scale

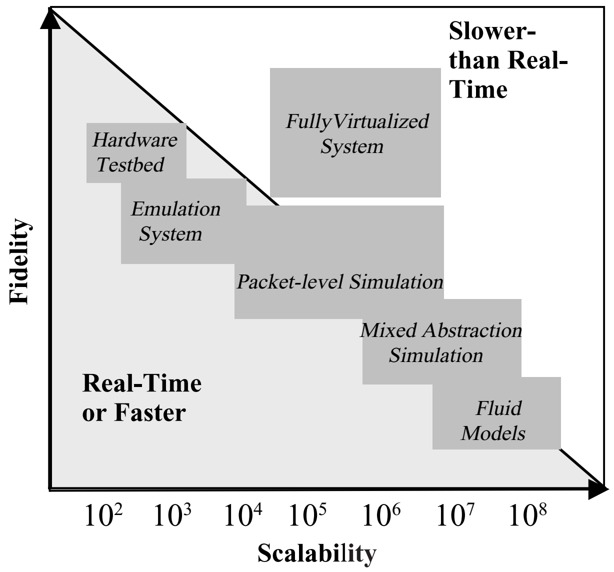


FIGURE 3.1: Variety of alternatives for wire-line network simulation. Scalability ranges are based on present day capabilities of testbeds and tools reported in the literature

network simulations are thus required for detailed and realistic simulations, where individual network parameters might produce a significant difference in the behavior of experiments. A key impediment in running large-scale simulations is the scalability of the network simulators. Packet-level simulation has lately seen great advances, especially due to parallel/distributed execution capabilities of network simulators (e.g., PDNS [3] and GTNetS, and Genesis [4]). Scales of up to a few million nodes have been reported for simulation of large TCP/IP networks [5].

Hybrid simulations, using a combination of fluid and packet-level models, have been used to scale network simulations by at least another order of magnitude [6], but they have been largely constrained in generality (e.g., limited accounting for feedback effects). Further, they are restricted to core network segments (backbone links and routers) and are difficult to extend to general application traffic at end hosts.

As can be seen, packet-level simulation exhibits the best tradeoff between scalability and fidelity, and holds potential to sustain Internet-scale experiments without great loss of flexibility or accuracy. Complex applications can be easily modeled in terms of their TCP/IP packet exchange behavior, and the simulation can be enhanced as needed if/when new protocol types or application-level flows are to be explored.

At sufficiently large levels of scale (e.g., hundreds of thousands of nodes or larger), most high-fidelity systems break down because of their inability to keep up with real time for their "real-system" components. For such large scales, a *fully virtualized system* can help overcome their inherent real-time constraint, without sacrificing fidelity. The fully virtualized system has the potential for achieving the highest fidelity among all approaches, even higher than that of a general hardware testbed. Theoretically it is not limited in scalability, but it is only limited practically by the amount of computation power available.

Imagine an Internet that entirely executes not by real-time clocks, but on virtual (simulation) clocks. Such a network not only retains the highest fidelity level, but also is delinked from real-time completely. Such a virtual network can be achieved as follows. Network links and routers are modeled using traditional packet-level (parallel/distributed) network simulators. End hosts are modeled as real systems themselves, with full blown operating systems, file systems, etc. However, unlike emulation systems that have end hosts running on real hardware, the end hosts are executed in virtualized environments. Since the end hosts are now under the control of a virtual host, they are not free running anymore, and hence can be controlled at will. Since the network is executed as a (packet-level) simulation, its execution is also already controllable. All components of the entire system are lifted away from real time and placed on a controllable virtual timeline.

3.2 PACKET-LEVEL MODELS

Packet-level models of network operation are some of the most general-purpose models that can be developed relatively easily. In this approach, the basic operation of the network is represented as exchanges of data packets among network nodes over network links. In this view, the protocol processing resides at end hosts, and the intermediate router nodes forward packets, subject to queuing at the links (and discarding packets upon experiencing resource constraints).

Packet-level modeling affords several advantages. Models are easier to develop as they roughly mimic actual network operation. Protocol processing can be modeled in great detail; in fact, some simulators incorporate actual kernel-level network stack source code into network simulator models to perform detailed processing (e.g., TCP/IP processing code was reused with a few modifications from Berkeley Standard Distribution (BSD) UNIX implementation in GloMoSim [7]). Packet models are easier to interface with actual operational networks (for network emulation) due to the fact that many packet-level details are preserved (e.g., TCP header fields such as sequence numbers and checksums). Another advantage is that since packet-level models capture the network operation effects at a fine resolution, any emergent behavior evident in actual networks can implicitly and naturally emerge in the simulated network. In other models, emergent behavior (e.g., self-similarity) needs to be explicitly realized in the model. A disadvantage of using packet-level models is that the computation cost can become significant when dealing with high-speed links or large number of nodes.

3.2.1 Event List Management

The choice of the data structure used to maintain the sorted list of future events is important to ensure efficient simulation. In network configurations involving large number of nodes and links, the number of pending events at any given moment during simulation can be quite large. The number of events included in this is proportional to the number of flows active and the number of packets in flight on links at any given time. A large portion of execution time spent by the simulator can be consumed in maintaining the future event list. In order to keep the list of maintenance overheads low, several data structures have been proposed. The calendar queue [8] data structure is generally well suited for events whose timestamps are distributed exponentially, whereas it can perform quite poorly in certain cases containing many timestamps that are close to each other. The in-place heap priority queue data structure is observed to give more uniformly consistent performance on a wide range of scenarios, delivering logarithmic complexity for insertions and deletions. Other data structures exist, such as red–black trees, which can be useful in other scenarios. It is important to note that the choice of the data structure for event list management can be critical when it comes to simulating large network configurations, and hence careful attention needs to be given when runtime efficiency is important. When large network configurations are used, the number of events in each processor becomes extremely large, roughly on the order of the sum of the number of links and the number of nodes in the network. Since this can reach hundreds of thousands, use of the right data structure becomes critical for performance. Simulation of a single packet processing typically takes on the order of just a few microseconds of CPU time, while the event processing time can become comparable to or more than this packet processing time. This relationship of processing time and priority queue insert/delete time thus becomes pronounced when the event list becomes large.

3.2.2 Modeling Packet Routing

One of the biggest challenges in packet-level modeling of large-scale networks is in the representation of routing tables. Consider the case when a packet arrives at a router over an incoming link. The router must forward that packet via another (outgoing) link in order to help the packet reach its destination. The next hop in the packet's path to its destination is determined in different ways, depending upon the type of routing method used in the network. One of the most common routing protocols is the open shortest path first (OSPF [9]) routing. A simplified model of this routing method is often used in network simulators, in which the topology is assumed to be static with no failures, and transmission delay (latency) is used as the link weight in computing the shortest path between any pair of nodes. A general representation of the routing table at router r contains entries of the form $\langle s, d, i, j \rangle$, where s represents the source address of the packet, d is the destination address of the packet, i is the incoming link identifier at router r, and j is the outgoing link identifier at router r. A common simplification of this is to ignore the source address *s* and incoming link identifier *i* (i.e., routing is only based on destination address, irrespective of source address). This table is maintained at every router, such that routing entries for all destinations exist at every router. Even with the simplification of not supporting source-based routing (i.e., only one entry exists per destination), the memory requirement for representing this model scales in a quadratic fashion. If n is the number of routers, the memory needed to store the routing table is proportional to n^2. While this does not pose a problem for simulating small networks, it can be quite significant when n is on the order of thousands of routers, making the routing table size exceed the amount of memory available in the computer.

This problem is addressed using a memory versus speed tradeoff approach. Instead of computing a priori the shortest path information among all pairs of nodes and storing them in the routing table, a dynamic approach is used to compute the routes on the fly. An incremental shortest-path algorithm can be used to compute the shortest path between the source and the destination of a packet when the packet reaches a router. Frequently used routes are cached, thus minimizing recomputation for significant routes, while avoiding the storage overhead of full routing table. If all possible paths are not used in the network scenario (which is commonly the case), this scheme has the added advantage of saving computation time by only computing routes that are actually needed for a specific simulation scenario. A scheme based on this approach, known as NIx-Vector routing [10], is used in the GTNetS and PDNS network simulators. This approach is a major factor in enabling the simulation of large networks containing one million nodes or more.

While the shortest path routing model is generally adequate for medium-sized networks (with thousands of nodes), additional hierarchical models are needed in order to capture the complexity of routing across autonomous systems (AS), such as using models of the border gateway protocol (BGP). Addition of BGP protocol models greatly increases the execution complexity of the models but has the advantage of simulating routing phenomena in a highly realistic fashion. Simulators such as SSFNet and GTNetS support BGP routing as an option for the user. In fact, the BGP models in SSFNet have been used to analyze, validate, and explain the occurrence of certain routing update storms that were an indirect result of computer worm attacks (in this case, the attack of Code Red II).

3.2.3 Interfacing Simulated Networks with Real Networks

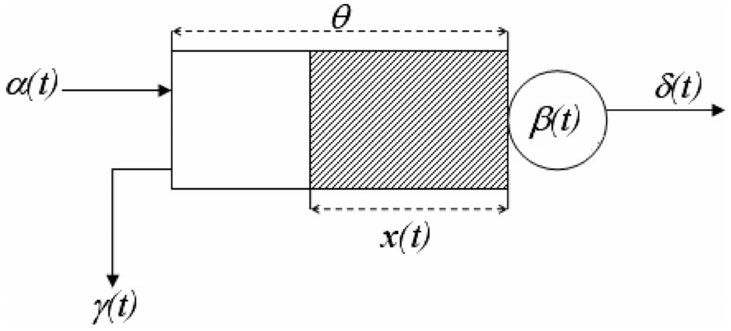
This is also called network emulation. Some of the systems supporting such integration include *ns*-2 (using the "*ns* emulator" called *nse*), Maya (which is based on the GloMoSim simulator), and EmuLab (which is based on *nse*). In a network emulation environment, the simulator contains interfaces to send and receive packets from actual deployed systems. The deployed systems act as if they are interfacing with an existing wide-area or local-area network. The packets created by the actual systems are imported into the simulation environment and processed normally along side other simulated packets. The advantage of this approach is the ability to test and validate actual systems in a more controlled environment than can be found in actual wide-area networks.

When operating in an emulation environment, the simulator must run in real time. This means that the simulation time value must remain approximately equal to the time measured by the actual deployed systems. It is easy to see why this is true. End systems running the TCP protocol (for example) will send a data packet at some time T and schedule a retransmission timer at some time $T + \Delta t$. If the simulation environment is running much slower than real time, the timeout period will elapse (from the point of view of the actual system), causing a packet retransmission. The original acknowledgement packet will eventually arrive from the simulator, but much too late to be of any value to the actual system. Similarly, if the simulation environment is running much faster than real time, the actual system will receive acknowledgements much too soon and will have an unrealistic view of the round trip time between the source and destination. This requirement for real-time execution in the simulation environment places severe constraints on the scalability of simulations when used in an emulation environment.

3.3 FLUID MODELS

Another approach to constructing models of network behavior is to ignore individual packets in the network, but rather focus on the behavior of a large number of flows taken in an aggregate. In this approach, the flows can be thought of as a fluid flowing through fixed capacity pipes, with containers (buffers) that hold excess fluid during periods of excess demand on a given pipe. This approach is described in detail by Wardi and Riley [11]. A simple model of a single queue is shown below.

In this simple example, data is arriving at a single queue at the time-varying rate $\alpha(t)$. The output pipe has the ability to send data at the rate $\beta(t)$, and the data is leaving the queue at the rate $\delta(t)$. The queue has a fixed capacity Θ and contains an amount of fluid defined by



$x(t)$. Finally, fluid is lost from the queue at the rate $\gamma(t)$. It is easy to see that the behavior of these processes can be described as follows:

$$\delta(t) = \begin{vmatrix} \alpha(t) & x(t) \neq 0, \\ \beta(t) & \text{otherwise.} \end{vmatrix}$$

$$\gamma(t) = \begin{vmatrix} 0 & x(t) < \Theta, \\ \alpha(t) - \beta(t) & x(t) = \Theta. \end{vmatrix}$$

$$\frac{\mathrm{d}x}{\mathrm{d}t^{+}} = \begin{vmatrix} 0 & x(t) = 0, \alpha(t) < \beta(t), \\ 0 & x(t) = \Theta, \alpha(t) \geq \beta(t), \\ \alpha(t) - \beta(t) & \text{otherwise.} \end{vmatrix}$$

Once a model of the simple single-stage queue described above has been constructed, a network of these can be created. For example, the output of one queue can be connected to the input of a second queue. Optionally, the input to a queue can be the sum of the outputs from several other queues. Further, the output of a queue might be split, with some fraction feeding one queue and the remaining feeding some other queue. Using this approach, complicated networks can be constructed and models of actual deployed networks can be created.

Using this approach, the simulator no longer models the behavior of individual packets. Instead, the simulator event models *rate changes* at each queue. For example, an inflow rate $\alpha(t)$ might reduce by a factor of 2 due to congestion control by a TCP end point. Alternately, an inflow rate at a queue might increase or decrease due to a change in the outflow rate of an upstream queue. Thus, the simulator uses rate change events as the mechanism to recompute the state of the network, as opposed to packet receipt event in the packet-based approach. Since we have aggregated a large number of flows, each rate change event represents a large number of packets, resulting in a more computationally efficient simulation.

However, this approach is not devoid of problems. It can be shown that a single rate change event at some point in the network could cause a large number of rate changes at potentially every other queue in the system. This so-called ripple effect can have a significant adverse effect on the overall simulator performance. Further, this approach generally ignores delays in the system. In other words, a change in outflow rate at one queue is instantaneously observed at downstream queues. This makes models that depend on network delays (such as TCP's round-trip time estimation) difficult to handle. Abstracting away individual packets makes it impossible to observe the performance and behavior of any individual data flow. Finally, this approach is unable to deal with flows that take different routes through a network, which reduces the number of flows that can be aggregated in this way.

CHAPTER 4

Wireless Network Simulation

The previous section discussed a number of simulation techniques that have been used or proposed to model computer networks using *wired* links. As mentioned, an accurate model of the behavior of such a link is quite easy to create, and is computationally efficient. In contrast, models for *wireless* networks are difficult to construct, difficult to validate, and computationally intensive. This difficulty stems from several factors.

1. A wireless link is inherently a *broadcast* medium. This means that a packet transmission from any single node must result in the generation and processing of a packet receipt event at every neighbor node within the transmission power range of the transmitter. While this is also the case in a wired multiaccess LAN (such as an Ethernet), the Ethernet model can often safely ignore packet receipt event at all nodes other than the addressed recipient. In the wireless domain, commonly used collision avoidance methods require the receipt and processing of all packets, regardless of the addressee.
2. Channel fading, path loss, and interference play a significant role in the behavior and performance of a wireless medium. Correct and efficient computation of the received signal strength of a transmission is extremely complex and is highly dependent on the environment and other factors that are difficult to model. Further, collisions and interchannel interference also contribute to the overall performance of the medium. Again, a wired Ethernet network in principle has similar issues. However, there is very little signal degradation on a wired LAN, and well-known collision detection mechanisms nearly eliminate the effects of collisions on moderately loaded networks. In fact, many existing models of Ethernet networks ignore collisions completely with little loss of accuracy. Unfortunately, the same is not true for wireless media, since the effects of signal loss and collisions can be significant.
3. The existence of end-system mobility introduces a number of computationally demanding issues in a wireless simulation environment. The broadcast nature of wireless requires that every neighbor of a transmitting element be informed of the transmission with a simulation event. However, the set of potential neighbors for any given

transmitter is not static due to node mobility. In the simplest case, every node in the entire system must be examined for every transmission, to determine if the node is a neighbor of the transmitting node.

4. Limited transmission range of a typical wireless transmitter leads to the use of multihop wireless routing protocols, the behavior of which must be modeled in the simulation environment. In contrast, a typical wired network will often completely ignore the routing protocol, and assume an ideal shortest path between pairs of nodes with little loss of accuracy. In wireless networks, the behavior and performance of the routing protocol itself plays a major role in the overall network performance.

In the following sections, we will discuss each of these issues in detail.

4.1 BROADCAST TRANSMISSIONS

Fig. 4.1 shows a graphical representation of a typical wireless simulation. In this scenario, there are 500 nodes placed randomly in a square geographic region of 1000 m by 1000 m. The random placement is based on a polar coordinate system, with the angle chosen uniformly in the range of $[0 \ldots 2\pi]$. The range is chosen from an exponential distribution with a mean of 250 m and a maximum value of 500 m. This type of random distribution accounts for the clustering of nodes near the center of the region, by design.

The figure shows that we are modeling two nodes transmitting packets at the same time. Each transmitter is shown as a cyan square, and has several neighbor nodes that are receiving the transmission correctly (shown as green squares). In this example, the packet is actually addressed to a single recipient, which is indicated by a blue square in the figure. There are a small number of receiving nodes shown in yellow, indicating that a node is aware that a packet is being transmitted, but the signal strength is too weak to properly detect the signal. Finally, there are several receiving nodes shown in red, indicating a collision. Here, a collision means that a given node is receiving transmissions from more than one node simultaneously, and thereby is unable to correctly receive either transmission.

It is easy to see from this simple scenario that the simulation of a simple and relatively small wireless network such as the one shown in the figure can result in a large number of simulation events. Every packet transmission event can result in several hundred receive events as is shown in our example. The proper modeling of the ubiquitous IEEE 802.11 wireless MAC protocol requires each node to be aware of any transmission from any neighbor, regardless of to whom the packet is addressed.

One way to ease this computational burden is discussed by Lee in [12]. While it is true that each receiver must be aware of neighboring transmissions, this knowledge is only useful if

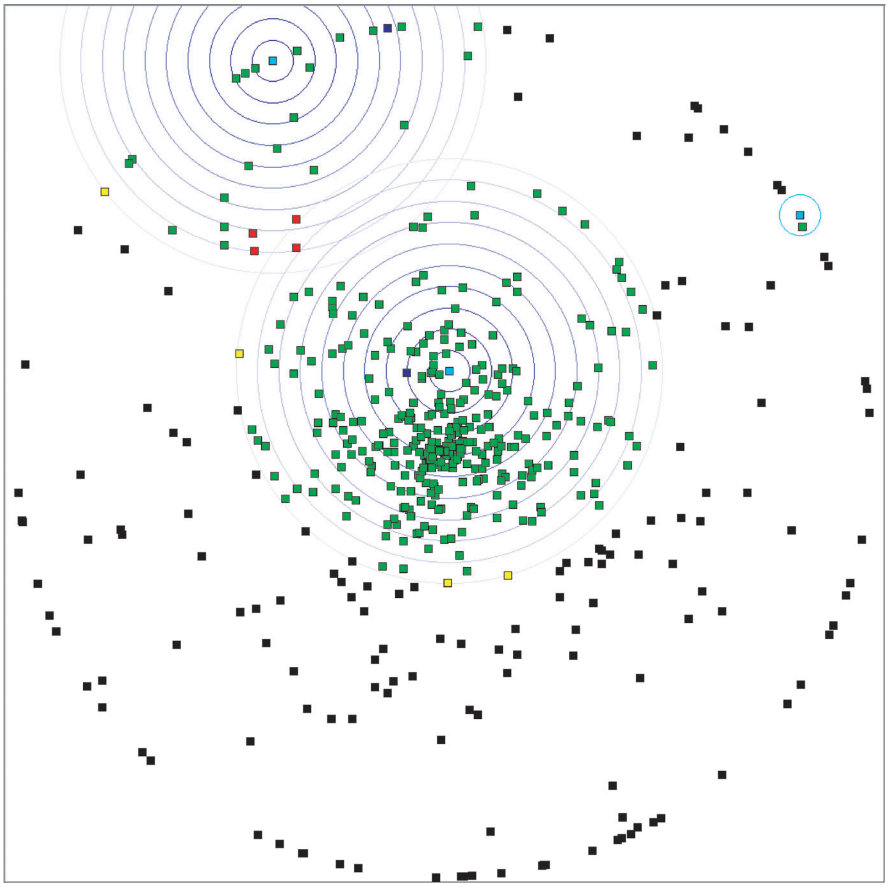


FIGURE 4.1: A visualization of a typical wireless simulation

a given node itself becomes a transmitter in the near future. In other words, each of the nodes shown in green in our example has processed a receive event informing it that a neighbor is transmitting. However, these green nodes will only use this information when they decide to transmit. Given this, the approach discussed by Lee does not inform *nondesignated* receivers (those receivers that are not the addressee of a packet) of packet transmissions. Rather, any node initiating a transmission action must first examine a transmission history buffer, to see if any of its neighbors has transmitted a packet recently. If not, the transmission proceeds normally. If so, the node sets the back off timers and other 802.11 specific variables as if it had received that previous transmission. This approach has been shown to lead to a significant improvement in simulation execution time in some cases.

4.2 PATH LOSS CALCULATIONS

Perhaps the single most difficult and computationally challenging aspect to wireless network simulation is the calculation of the electromagnetic signal strength at a receiver. This signal strength is a function of many variables, including the transmitter power, transmitter antenna gain factor, receiver antenna gain factor, antenna orientations, terrain obstructions, weather, and ambient electromagnetic interference, just to name a few.

As is often the case in modeling and simulation, there are several accuracy versus efficiency tradeoffs that can be made in the computation of the signal strength. An often-used abstraction is the simple *free-space* model for signal strength, shown as follows:

$$P_r = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2\, d^n L},$$

where P_t is the transmission power, G_r is the gain of the receiver antenna, G_t is the gain of the transmitter antenna, λ is the wavelength of the radio carrier signal, d is the distance between the transmitter and receiver, n is the *path-loss exponent*, L is an independent system loss factor (L greater than or equal to 1), and P_r is the computed receiver signal strength. The path-loss exponent is dependent on the environment and other factors and is typically in the range of 2 to 4. This model is clearly computationally efficient, but is known to be highly inaccurate; it may however be satisfactory in some cases, depending on the requirements of the overall simulation.

Another slightly more complex model is given below:

$$P_r = P_t + G_r + G_t - 10\log_{10}\left(\frac{4\pi f}{C}\right) - n10\log_{10} R - \sum_{j=1}^{j=k} X_j,$$

where P_t is the transmission power, G_r is the gain of the receiver antenna, G_t is the gain of the transmitter antenna, f is the frequency of the radio carrier signal, C is the speed of light, n is the path-loss exponent, R is the distance between the transmitter and receiver, k is the number of obstructions between the transmitter and receiver, and X_n is the signal loss through obstruction n. This formula is similar to the free-space model above, except that the signal strengths are defined and computed in milliwatt-decibels, or dBm, and the inclusion of the obstructions set $\mathbf{X}$. This calculation is slightly more complex than the simple free-space model, but may be more accurate. Unfortunately, determining the set $\mathbf{X}$ requires detailed knowledge of the terrain environment of the network, and reasonable values for each X_n can be hard to determine.

Another commonly used model for path loss is the *two-ray* model. In this model, constructive or destructive interference of the signal due to ground reflections is taken into account. The formula for the two-ray model is

$$P_r = \frac{P_t G_t G_r H_t^2 H_r^2}{d^4 L},$$

where all parameters are as above, with the addition of the transmitting antenna height H_t and the receiver antenna height H_r. However, this approach is known to be inaccurate for small distances. Therefore, a common approach is to compute a crossover distance d_c as follows:

$$d_c = \frac{(4\pi H_t H_r)}{\lambda}.$$

For separation distances less than d_c the free space model is more accurate, and for distances greater than d_c the two-ray model is more accurate.

Finally, common models based on received signal strength use a *shadowing* effect. In essence, the shadowing accounts for signal variations due to buildings and other obstacles along the path. While the set **X** discussed in the second free-space model above is designed to account for these obstructions, a random model generally produces better results. The commonly used log-normal shadowing model computes the signal strength in dBm as a function of a reference strength at a short distance d_0. A typical value of d_0 is 1 m, and the simple free-space model is used to compute $P_r(d_0)$. Given that, the signal strength at distance d is given by

$$\left[\frac{P_r(d)}{P_r(d_0)}\right]_{\text{dBm}} = -n10\log_{10}\left(\frac{d}{d_0}\right) + S_{\text{dB}}.$$

Here S_{dB} is a Gaussian (normal) random variable with zero mean and a standard deviation of σ. Typical values for σ are in the range of 3 to 12, depending on the environment. Since S_{dB} is a normal distribution in units of decibels, the actual variation in signal strength due to S_{dB} is a log-normal distribution.

4.3 MOBILITY

In a wired network environment, the set of directly connected neighbors for a given node is typically constant, excepting when modeling link failures and repairs. However, in wireless environments, nodes almost always have a *mobility* pattern. This might be due to soldiers moving across a battlefield, students walking across a campus, or first-response personnel in a natural disaster area. Regardless of the cause of the node mobility actions, the simulation environment must take them into account and ensure that the simulation produces traffic in a realistic way based on the position of all nodes at any time.

A common model for mobility is the simple *random waypoint* model. In this model, each node randomly chooses an alternate location (called a *waypoint*) on the geographic region being modeled, and a random velocity. The node is then assumed to move in a straight line from the current location to the new alternate location at the chosen constant speed. When arriving at the new location, the node chooses a random pause time and remains stationary for that amount of time. After the pause time has elapsed, the node again chooses a new location and velocity and repeats the process. It is easy to see that this model is simple and easy to construct and understand.

A variation on the random waypoint model is called the *specific waypoint* model. This is identical to the random waypoint model, excepting that the waypoints are deterministic and selected in advance. This allows for known mobility patterns, such as students moving from one building to another between classes.

A third variation is called the *Manhattan* model. In this model, motion is only allowed on predefined paths (streets). Thus when moving from one waypoint to another, the node may not be able to proceed in a straight line, but rather must stay on the streets and make turns at intersections.

It is easy to see that there is little computational overhead as a direct effect of node mobility. In any of the above models, the computations are simple laws of motion and physics and easy to compute. However, the indirect effects of mobility, specifically the changing neighbor set for packet transmissions, can be significant. Recall that for any wireless transmission, any other node in range of the transmitter must be informed of the transmission. In order to be sure that every possible neighbor is informed, the path loss calculations must be performed for *every other node* in the simulation, to determine if that node is able to receive the signal. Even though at time t a given node k may be unable to detect a transmission from node n, there is no guarantee that this is also true at time $t + \Delta t$, since both nodes k and n may have moved during the Δt interval. Thus the overall computational complexity of each packet transmission can be as high as $O(N^2)$, where N is the total number of nodes in the simulation.

One approach to easing this computational burden is to compute a neighbor set for a given transmitting node, and to retain this set for use with later transmissions. In addition, an *expiration time* for the neighbor set is calculated. This expiration time is defined as the earliest time in the future when the calculated neighbor set might possibly be different. For example, if a given transmission is calculated to be detectable within 100 m of the transmitting unit, clearly any node at a distance of 200 m away will not be a neighbor. If the maximum mobility speed is 10 m s^{-1}, then the second node is guaranteed to be out of range for at least 5 s in the future (if both the transmitter and receiver are moving directly toward each other at maximum speed). Once the expiration time is known, any future transmission by the selected transmitter can use the previously calculated neighbor set until the expiration time is reached, which can result in significant computational savings.

4.4 AD HOC ROUTING

Two different issues related to wireless data communications combine to lead to the necessity of multihop, ad hoc routing in wireless networks. First is the node mobility as discussed in the previous section. When end systems are mobile, the neighbor set is not fixed and therefore routes between pairs of nodes change frequently. Second is the limited transmission range of typical wireless devices. The IEEE standard 802.11 specifies a maximum range of 2000 m, but in practice a hundred meters is common. Thus, for any reasonable-sized geographic region, allowing any node to communicate with any other node will by necessity result in multihop packet forwarding, known as ad hoc routing.

There is considerable research in methods for efficient ways to determine routes, forward packets around congested areas, repair broken routes, and maintain routing state, just to name a few. Two popular ad hoc routing protocols are "ad-hoc on-demand distance vector" (AODV) and "dynamic source routing" (DSR). It is well known that the specific routing protocol used and the parameters specified for the protocol can have a significant effect on the overall performance of the network.

The routing protocol may also have a significant effect on the performance of the simulator as well, both in terms of execution time and memory footprint. Depending on how routing information is stored in the simulator, the memory requirements can be O(N^2), where N is the number of nodes in the network. In addition, the overhead of generating, transmitting, and receiving the control packets for the routing protocol can be considerable. A recent study by Zhang [13] showed that in very large networks (50,000 nodes) there can be as many as 500 control packets for every data packet. Clearly, such overhead is a significant computational burden on both the simulator and the actual deployed network.

CHAPTER 5

Performance, Scalability, and Parallel Model Execution

We now turn our attention to issues concerning the performance and scalability of network simulation programs. By *performance* we mean the execution speed of the network simulator. By *scalability* we mean the size of the network that can be modeled. Performance and scalability are important issues in network simulation because it is often the case that one would like to model networks that are beyond the capabilities of the available simulation tools. One approach to addressing these concerns is to utilize parallel computation, so a major emphasis of our discussion considers the exploitation of parallel computers to execute the simulation program.

In practice, two factors that often limit the scale of packet-level simulations that can be performed are the amount of memory required and the amount of computation time needed to complete each simulation run. Because the simulator requires a certain amount of memory to represent the state of each node, memory requirements increase at least linearly with the number of nodes. Memory requirements may increase faster than linearly in some simulators, e.g., to represent routing tables [10]. Thus, the amount of memory on the computer performing the simulation limits the number of nodes that can be represented.

At the same time, the amount of time required to complete a packet-level simulation usually increases in proportion with the amount of packet traffic that must be simulated. This is because packet-level simulators are usually based on a discrete-event paradigm where the computation consists of a sequence of event computations, so the execution time is proportional to the number of events that must be processed, assuming that the average amount of time to process each event remains constant. Events in a packet-level simulation represent actions associated with processing a packet such as transmitting the packet over a link. Thus the execution time in a packet-level simulation is proportional to the number of packets that must be processed. Assuming that the modeler is only willing to wait a certain amount of time for the simulation to complete, this places a limit on the amount of traffic that can be simulated.

These observations place specific limits on the scale (size and amount of traffic) that can practically be modeled by a given simulator using a specific hardware platform. A notional

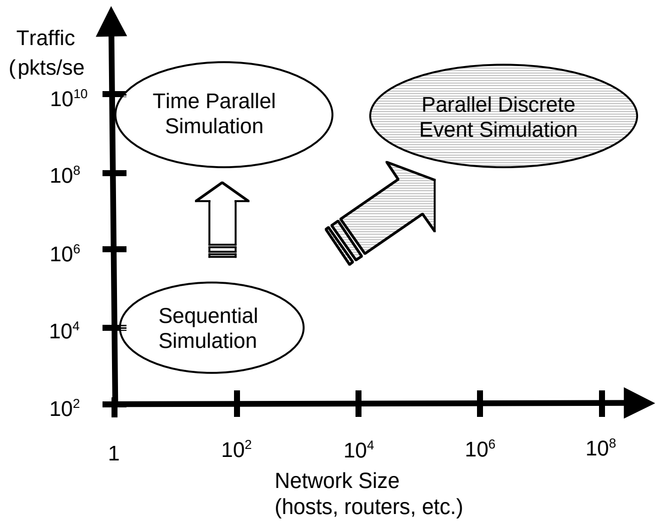


FIGURE 5.1: Notional diagram depicting network size (nodes and traffic) that can be simulated in real time

diagram illustrating these scalability limitations is shown in Fig. 5.1. In this figure we normalize the execution time constraint by requiring that the simulation *execute in real time*, i.e., to simulate S seconds of network operation requires no more than S seconds of wallclock time. Each shaded area in Fig. 5.1 represents a class of network simulators. Sequential simulation methods provide a baseline. As will be discussed later, contemporary network simulators such as *ns*-2 with suitable optimizations to reduce memory consumption are able to simulate on the order of tens to hundreds of thousands of packet transmissions per second of wallclock time for networks containing thousands to tens of thousands of nodes on a modern workstation or personal computer.

Effective use of simulation tools requires the design of an experimentation plan. Preceding the development of the plan, one must define the specific goals of the experiments and the metrics that will be used. For example, one goal might be to compare a particular routing algorithm against competing algorithms in order to compare end-to-end delay and delay variance. Simulation runs utilizing different routing algorithms are obviously needed. In addition, for each routing algorithm, multiple runs will be required to test the sensitivity of the output metrics to specifics of the test scenario, e.g., the type and amount of traffic that is generated, or the network configuration (e.g., topology). Further, even for a single configuration and test scenario, multiple runs are required to ensure the simulation results are statistically significant. Multiple runs with different random number streams are needed to ensure that random variations, e.g., in the interarrival times of messages in the traffic stream, do not have a large impact on output metrics. Such variances are captured by *confidence intervals* that characterize that the results are within a certain range (termed the confidence interval), with a certain likelihood (e.g., 95%). A related aspect that must be considered in the experimentation plan is the length of time that the network must be simulated. A typical network simulation will start with an idle network (message queues are assumed to be empty) and go through a "warm up" period before the network reaches steady state behavior. Often, collection of statistics by the simulator is only begun after a sufficient warm up period has been completed. Statistical aspects of simulation are beyond the scope of the discussion here, but are described in references such as [14].

Concurrent execution of the simulation computation can improve the scalability of the simulator both in terms of network size and execution speed, depending on the method that is used. Time-parallel simulation methods such as that described in [15–21] parallelize a fixed-sized problem by partitioning the simulation time axis into intervals, and assigning a processor to simulate the system over its assigned time interval. These methods increase the execution speed of the simulator enabling more network traffic to be simulated in real time, but do not increase the size of the network being simulated (see Fig. 5.1). Parallel discrete-event methods utilize a space-parallel approach where a large network is partitioned and the nodes of each partition are mapped to a different processor, offering the potential for both network size and execution speed to scale in proportion to the number of processors. Several parallel simulators using this approach have been developed. Finally, in addition to these "pure" time and space-parallel approaches, some approaches combine both time and space parallelism, e.g., see [22].

Finally, another trivial approach to parallelism involves simply executing different simulation runs on different machines. While this approach does not address the issue of simulation scalability because each machine must have the capacity to complete a simulation run, this is a widely used technique to reduce the time to complete sets of simulation experiments.

5.1 SIMULATOR PERFORMANCE

Just as computer hardware designers utilize metrics such as the number of instructions of floating point operations a machine can process per second of wallclock time, it is useful to have a quantitative metric to characterize the performance of packet-level network simulators. Since the bulk of the simulation computation in a packet-level simulator involves simulating the transmission and processing of packets as they travel from the source, through intermediate nodes (routers), to its destination, it is convenient to use the number of *Packet Transmissions that can be simulated per Second of wallclock time* (or PTS) as the metric to specify simulator speed. This metric is useful because given the PTS rate of a simulator, one can estimate the amount of time that will be required to complete a simulation run if one knows the amount of traffic that must be simulated, and the average number of hops required to transmit a packet from source to destination.

Specifically, the execution time T for a simulation run can be estimated by the equation

$$T = N_T/\mathrm{PTS}_S$$

where N_T is the number of packet transmissions that must be simulated and PTS_S is the number of simulated packet transmissions that can be simulated per second of wallclock time by simulator S. N_T depends on several factors. A first-order estimate is

$$N_T = N_F{}^* P_F{}^* H_F$$

where N_F is the number of packet flows that must be simulated, P_F is the average number of packet transmissions per flow, and H_F is the average number of hops a packet must traverse in traveling from source to destination. It is important to note, however, that this is an approximation because it does not consider packet losses, or other nonuser traffic packets that must be sent by the protocol being used, e.g., acknowledgement packets using TCP. Nevertheless, when these considerations are accounted for, these equations provide a reasonable means to estimate execution time.

When the objective is to achieve real-time performance, the execution time constraint PTS_S must be at least as large as $N_F{}^* R_F{}^* H_F$ where R_F is the average rate of packets transmitted (packets per second) per flow, again with the same caveats concerning losses and protocol-generated packets. For example, consider a simulation of 500,000 active UDP flows in real time where each flow produces traffic at a rate of 1 Mbps. Assuming 1 KB packets, this translates to 125 packets/second. If the average path length is 8 hops, the simulator must execute at a rate of 500 million PTS to achieve real-time performance.

5.2 TIME-PARALLEL SIMULATION

As mentioned earlier, time-parallel algorithms divide the simulated time axis into intervals, and assign each interval to a different processor. This allows for massively parallel execution because simulations often span long periods of simulated time. A central question that must be addressed by time-parallel simulators is ensuring that the states computed at the "boundaries" of the time intervals match. Specifically, it is clear that the state computed at the end of the interval $[T_{i-1}, T_i]$ must match the state at the beginning of the interval $[T_i, T_{i+1}]$. Thus, this approach relies on being able to perform the simulation corresponding to the ith interval without first completing the simulations of the preceding ($i-1$, $i-2, \ldots, 1$) intervals.

Because of the "state-matching" problem, time-parallel simulation is really a methodology for developing massively parallel algorithms for specific simulation problems than a general approach for executing arbitrary discrete-event simulation models on parallel computers. Time-parallel algorithms are currently not as robust as space-parallel approaches because they rely on specific properties of the system being modeled, e.g., specification of the system's behavior as recurrence equations and/or a relatively simple state descriptor. This approach is currently limited to a handful of applications, e.g., queuing networks, Petri nets, cache memories, and multiplexers in communication networks. Space-parallel simulations offer greater flexibility and wider applicability, but concurrency is limited to the number of logical processes. In some cases, both time and space parallelism can be used.

One approach to solving the state matching problem is to have each processor guess the initial state of its simulation, and then simulate the system based on this guessed initial state [15]. In general, the initial state will not match the final state of the previous interval. After the interval simulators have completed, a "fix-up" computation is performed to account for the fact that the wrong initial state was used. This might be performed, for instance, by simply repeating the simulation, using the final state computed in the previous interval as the new initial state. This "fix-up" process is repeated until the initial state of each interval matches the final state of the previous interval. In the worst case, N such iterations are required when there are N simulators. However, if the final state of each interval simulator is seldom dependent on the initial state, far fewer iterations will be needed.

A variation in this approach has been devised in the context of simulating statistical multiplexers for asynchronous transfer mode (ATM) switches. ATM switching networks transmit fixed-sized data packets as the atomic unit of data that is transported through the network. A multiplexer combines several incoming traffic streams into a single output stream. Because the bandwidth of the outgoing line is usually smaller than the sum of the bandwidths of the incoming lines, cells may accumulate if the total incoming traffic flow exceeds the capacity of the output link. Buffer memory within the multiplexer is used to hold cells waiting to be sent on the outgoing link. If the buffer memory overflows the cell is simply discarded. Therefore the multiplexer must provide sufficient buffer memory to reduce the number of lost cells to an acceptable level. A typical design objective is on the order of only one in 10^9 transmitted cells that are lost due to overflows. Due to the infrequent nature of cell losses, long simulation runs are required to capture a statistically significant number of losses.

The input to the multiplexer can be described as a sequence of tuples $\langle A_i, \delta_i \rangle$ ($i = 1, 2, 3, \ldots$) with one tuple corresponding to a time period when the number of input lines that are active remains constant. Let A_i denote the number active or "on" (transmitting cells) sources, and δ_i denote the length of time during which the A_i input lines are active. For example, the tuples $\langle 2,3 \rangle$, $\langle 1,2 \rangle$, and $\langle 2,4 \rangle$ meaning initially two input lines are active for three units of time, then one is active for two units of time, then two for four units of time, and so on. Then the simulation problem can be formulated as follows: for a multiplexer with N input links of unit capacity, an output link with capacity C and a FIFO queue containing K buffers determine the average utilization and the number of lost cells for the incoming traffic characterized by the sequence of tuples $\langle A_i, \delta_i \rangle$.

Assuming that the simulation starts at time zero, let T_i denote the simulation time corresponding to the end of the ith tuple. Let $Q(T_i)$ denote the number of cells stored in the queue at time T_i. Let $S(T_i)$ denote the total number of cells serviced (transmitted on the output link) and $L(T_i)$ denote the number of cells lost up to time T_i.

During each tuple, the multiplexer will be in one of the two states. In the *overload* state $A_i > C$ and the queue is being filled. Conversely, in the *underload* state, $A_i \leq C$ and the queue is either remaining at the same capacity, or is being drained. All overflows occur during the overload state.

It is easy to see that

$$\begin{aligned} Q(T_i) &= \min(Q(T_{i-1}) + (A_i - C)\delta_i, K) \text{ if } A_i > C(\text{overload}),\\ &= \max(Q(T_{i-1}) - (C - A_i)\delta_i, 0) \text{ if } A_i \leq C(\text{underload}).\end{aligned}$$

$$\begin{aligned} S(T_i) &= S(T_{i-1}) + C\delta_i \text{ if } A_i > C(\text{overload}),\\ &= S(T_{i-1}) + ((Q(T_{i-1}) + A_i\delta_i) - Q(T_i)) \text{ if } A_i \leq C(\text{underload}).\end{aligned}$$

$$\begin{aligned} L(T_i) &= L(T_{i-1}) + (Q(T_{i-1}) + A_i\delta_i) - Q(T_i) - C\delta_i \text{ if } A_i > C(\text{overload}),\\ &= L(T_{i-1}) \text{ if } A_i \leq C(\text{underload}).\end{aligned}$$

These equations enable one to simulate the behavior of the multiplexer over time.

The above computation seems inherently sequential because Q, S, and L for the ith tuple depends on Q for the previous tuple. This problem can be solved by observing that certain tuples are guaranteed to cause either an underflow or an overflow. Specifically, a tuple $\langle A_i, \delta_i\rangle$ is guaranteed to cause an overflow if $(A_i - C)\delta_i \geq K$, because even if the queue were empty at the start of the tuple, the overload condition would last sufficiently long that the buffer must overflow. In this case, we know $Q(T_i) = K$. Similarly, if $(C - A_i)\,\delta_i \geq K$, then we know the queue will underflow during this tuple, meaning that $Q(T_i) = 0$. Therefore, we need only use a preprocessing step to scan the tuples, and identify those guaranteed to cause an underflow or an overflow. These tuples can be used to partition the time axis for the time-parallel simulation [21]. Because the state of the system, namely, the number of occupied buffers in the queue, is known at these points, independent simulations can be begun at these points in simulated time, thereby eliminating the need for a fix-up computation.

Of course, one must be able to identify a sufficient number of guaranteed overflow or underflow tuples for this algorithm to succeed. In practice there will usually be an abundance of guaranteed underflow tuples because cell losses are very rare. One implementation of this algorithm reported performance on the order of 10^{10} PTS (simulated packet transmissions per second) [21].

Another approach to time-parallel simulation is described in [17]. Here, a queuing network simulation is expressed as a set of recurrence equations that are then solved using well-known parallel prefix algorithms. The parallel prefix computation enables the state of the system at various points in simulated time to be computed concurrently. Another approach also based on recurrence equations is described in [23] for simulating timed Petri nets.

5.3 SPACE-PARALLEL SIMULATION

We now turn our attention to a more general approach to parallelization: space-parallel simulation. As stated earlier, the idea in space-parallel simulation is to partition the network into a collection of subnetworks, and assign a separate simulation process (called a *logical process* or LP) to simulate each subnetwork. Each LP can be viewed as a sequential simulator that simulates the subnetwork assigned to it. LPs interact by scheduling events for each other. For example, consider a packet being transmitted across a link that spans two subnetworks (LPs). The LP simulating the subnetwork sending the packet will schedule a packet arrival event at the LP simulating the subnetwork on the receiving side of the link. The timestamp of this event will typically be set equal to the time when the packet has been fully received by the receiving node, i.e., the simulation time when transmission began plus the delay for the first bit of the packet to traverse the link plus the time required to transmit the remainder of the packet, i.e., the size of the packet divided by the bandwidth of the link. If the LP scheduling the event is at the simulation time when the transmission begins, it will schedule the event with a simulation time in the future equal to the delay and packet transmission times. This amount of time into the future that the packet is scheduled is referred to as the lookahead associated with the communication link, and as will be seen later, is very important for achieving efficient parallel execution.

Scheduling an event between LPs (which may reside on different processors) is achieved by sending a simulation message between the corresponding LPs. For this reason, "scheduling an event for another LP" is synonymous with "sending a message" in the discussion that follows. LPs can *only* interact by sending messages to each other. Specifically, two LPs cannot interact by referencing shared memory.

This message-passing approach to parallelization raises several important issues:

- *Partitioning*. How should the network be partitioned into subnetworks/LPs? It is usually advantageous to partition the network so that there are more LPs than processors to allow some flexibility in mapping LPs to processors for load balancing purposes (described next). On the other hand, too many LPs reduce the execution efficiency of the simulation because of the overheads to manage many LPs, and communication between LPs is generally less efficient than communications within an LP. Another very important question concerns which communication links are "broken" when partitioning the network. For reasons to be explained shortly, it is much more advantageous to "break" communication links that have a high latency, i.e., it is better to partition low bandwidth links that span long distances than high bandwidth links connecting nodes that are close together.

- *Load balancing*. A related question concerns the distribution of logical processes among processors. The load distribution algorithm attempts to simultaneously ensure that each processor has approximately the same amount of computation to perform, while also trying to minimize the communication that takes place between processors. These two goals are sometimes conflicting. Static load distribution strategies map the logical processes to processors prior to the execution of the simulation, and do not attempt to redistribute workload during the course of the execution. Dynamic load distribution strategies do redistribute the mapping of processes to processors after the execution begins in order to try to maintain an effective load balance.
- *Synchronization*: A synchronization algorithm is needed to ensure that the parallel execution of the simulator yields the same results as a sequential execution. In particular, a synchronization algorithm is needed to ensure that events are processed in a correct order. Further, the synchronization algorithm helps us ensure that repeated executions of a simulation with the same inputs produce exactly the same results.

The need for an efficient synchronization algorithm is one aspect that distinguishes parallel network simulations from other forms of parallel computation. Synchronization plays a central role; the discussion that follows focuses on describing different approaches to synchronization.

5.3.1 Synchronization Algorithms

The goal of the synchronization algorithm is to ensure that each LP processes events in timestamp order. This requirement is referred to as the *local causality constraint*. Ignoring events containing exactly the same timestamp, it can be shown that if each LP adheres to the local causality constraint, execution of the simulation program on a parallel computer will produce exactly the same results as an execution on a sequential computer where all the events are processed in timestamp order. This property also helps us ensure that the execution of the simulation is repeatable; one need only ensure that the computation associated with each event is repeatable.

Each LP maintains local state information corresponding to the objects it is simulating and a list of timestamped events that have been scheduled for this LP, but have not yet been processed. This *pending event list* is like that described earlier for sequential simulations in that it includes local events that the LP has scheduled for itself as well as events that have been scheduled for it by other LPs. As before, the main processing loop of the LP repeatedly removes the smallest timestamped event from the pending event list and processes it. Processing an event means zero or more state variables within the LP may be modified, and the LP may schedule additional events for itself or other LPs. Each LP maintains a simulation time clock that indicates the timestamp of the most recent event processed by the LP. Any event scheduled by an LP must have a timestamp at least as large as the LP's simulation time clock when the event was scheduled.

As noted earlier, the synchronization algorithm must ensure that each LP processes events in timestamp order. This is nontrivial because each LP does not know what events will later be received from other LPs. For example, suppose the next unprocessed event stored in the pending event list has timestamp 100. Can the LP process this event? How does the LP know it will not later receive an event from another LP with timestamp less than 100? This question captures the essence of the synchronization problem.

Synchronization algorithms can be broadly classified as either *conservative* or *optimistic*. Conservative algorithms avoid the possibility of processing events out of timestamp order, i.e., the execution mechanism avoids synchronization errors. In the aforementioned example where the next unprocessed event has a timestamp of 100, the LP must first ensure that it will not later receive any additional events with timestamp less than 100 before it can process this event. On the other hand, optimistic algorithms use a detection and recovery approach. If events are processed out of timestamp order, a mechanism is provided to detect and recover from such errors. Each of these approaches is described next.

5.3.2 Conservative Synchronization

Conservative synchronization algorithms must determine when it is "safe" to process an event. An event is "safe" when one can guarantee no event containing a smaller timestamp will be later received by this LP. Thus, conservative synchronization algorithms must determine a lower bound on the timestamp (LBTS) of future messages that may later be received by an LP; events with a timestamp less than the LP's LBTS value are safe to process. For example, if the synchronization algorithm has determined that the LBTS value for an LP is 90, then all events with timestamp less than 90 may be processed. Conversely, all events with timestamp larger than 90 cannot be safely processed. Whether or not events with timestamp equal to 90 can be safely processed depends on specifics of the algorithm and the rules concerning the order that events with the same timestamp (called simultaneous events) are processed. Processing of simultaneous events is a complex subject matter that is beyond the scope of the current discussion, but is discussed in detail in [24]. The discussion here assumes that each event has a unique timestamp. It is straightforward to introduce tie breaking fields in the timestamp to ensure uniqueness [25].

5.3.2.1 The Null Message Algorithm

The "null message" or deadlock avoidance algorithms were among the first conservative synchronization algorithms that were developed [26, 27]. These assume that the topology is fixed, indicating which LPs send messages to which others, and that the topology is known prior to execution. In a network simulation, the LP topology simply corresponds to the topology of subnetworks. It is assumed that each LP sends messages with nondecreasing timestamps, and the communication network ensures that messages are received in the same order that they were sent in. This guarantees that messages on each incoming link of an LP arrive in timestamp order. This implies that the timestamp of the last message received on a link is a lower bound on the timestamp of any subsequent message that will later be received on that link. Thus, the LBTS value for an LP is simply the minimum among the LBTS values of its incoming links.

A first-in, first-out queue is maintained for each incoming link of the LP. Incoming messages arrive in timestamp order because of the above restrictions. Local events scheduled within the LP can be handled by having a queue within each LP that holds messages sent by an LP to itself. Each link has a clock that is equal to the timestamp of the message at the front of that link's queue if the queue contains a message, or the timestamp of the last received message if the queue is empty. The process repeatedly selects the link with the smallest clock and, if there is a message in that link's queue, processes it; if the selected queue is empty, the process blocks. The LP never blocks on the queue containing messages it schedules for itself, however. This protocol guarantees that each process will only process events in nondecreasing timestamp order, so the local causality constraint is never violated.

Unfortunately, this approach is prone to deadlock. A cycle of empty links with small link clock values (e.g., smaller than any unprocessed message in the simulator) can occur, resulting in each process waiting for the next process in the cycle. *Null* messages are used to avoid deadlock. Unlike the event messages described earlier, null messages do not correspond to any simulated activity such as a packet arrival; null messages are introduced solely for the purpose of preventing deadlocks. A null message with timestamp T_{null} sent from LP_A to LP_B is a promise by LP_A that it will not later send a message to LP_B carrying a timestamp smaller than T_{null}. Processes send null messages on each outgoing link after processing each event if the timestamp of the null message is larger than that of the last null or nonnull message sent on the link. A null message provides the receiver with additional information that may be used to determine that other events are safe to process. Processing a null message advances the simulation clock of the LP to the timestamp of the null message. However, no state variables are modified and no nonnull messages are sent as the result of processing a null message.

The timestamp on each null message is determined by examining the clock value of each incoming link; this provides a lower bound on the timestamp of the next event that will be removed from that link's buffer. When coupled with the delay to transmit a packet over the link, this bound can be used to determine a lower bound on the timestamp of the next *outgoing* message on each output link. For example, if the time to transmit a packet over a link is T, then the timestamp of any future packet arrival event must be at least T units of simulated time larger than any arrival event that will be received in the future. Whenever a process finishes processing a null or a nonnull message, it sends a new null message on each outgoing link. The receiver can then compute new bounds on its outgoing links, send this information on to its neighbors, and so on. It can be shown that this algorithm avoids deadlock [26].

The link transmission time is a key property utilized by virtually all conservative synchronization algorithms. This value is referred to as the *lookahead* of the link. In general, if an LP is at simulation time T, and it can guarantee that any message it will send in the future will have a timestamp of at least $T + L$ regardless of what messages it may later receive, the LP is said to have a lookahead of L. As we just saw, lookahead is used to generate the timestamps of null messages. One constraint of the null message algorithm is that it requires that no cycle among LP's exists containing a zero lookahead, i.e., it is impossible for a sequence of messages to traverse the cycle, with each message scheduling a new message with the same timestamp. This will clearly be the case in any network simulation because the speed of light and the maximum link bandwidth dictates the minimum amount of time that must elapse for one LP to affect another.

5.3.2.2 Second Generation Algorithms

The null message algorithm may generate a large number of null messages, especially if the link lookahead is small. Consider a simulation containing two LPs. Suppose both are blocked, each has reached simulation time 100, and each has a lookahead equal to 1. Suppose the next unprocessed event in the simulation has a timestamp of 200. The null message algorithm will result in null messages exchanged between the LPs with timestamp 101, 102, 103, and so on. This will continue until the LPs advance to simulation time 200, when the event with timestamp 200 can now be processed. A hundred null messages must be sent and processed between the two LPs before the nonnull message can be processed. This is clearly very inefficient. The problem becomes even more severe if there are many LPs.

To solve this problem, we observe that the key piece of information that is required is the timestamp of the next unprocessed event within each LP. If the LPs could collectively recognize that this event has timestamp 200, all of the LPs could immediately advance from simulation time 100 to time 200. Thus, the time of the next event across the entire simulation provides critical information that avoids the "time creeping" problem in the null message algorithm. This idea is exploited in more advanced synchronization algorithms.

A second problem with the null message algorithm arises when simulating networks that are highly connected. In this case, each LP must send a null message to many other LPs after processing each event. This also results in an excessive number of null messages.

The deadlock detection and recovery algorithm provides one solution to this problem. This algorithm allows the computation to deadlock, but then detects and breaks the deadlock [28]. The deadlock can be broken by observing that the message(s) containing the smallest timestamp is (are) always safe to process. Alternatively, one may use a distributed computation to compute lower bound information (not unlike the distributed computation using null messages described above) to enlarge the set of safe messages.

Synchronous algorithms offer another approach. Here, the computation cycles between (i) determining which events are "safe" to process, and (ii) processing those events. It is clear that a key step is determining the events that are safe to process during each cycle. Each LP must determine a lower bound on the timestamp (LBTS) of messages that it might later receive from other LPs. This can be determined from a snapshot of the distributed computation as the minimum among (1) the simulation time of the next event within the LP if it is blocked, or the current time of the LP if it is not blocked, plus the LP's lookahead and (2) the timestamp of any transient message, i.e., any message that has been sent but has not yet been received at its destination.

Barrier synchronization can be used to obtain the snapshot. Transient messages can be "flushed" out of the system in order to account for their timestamps. If first-in, first-out communication channels are used, null messages can be sent through the channels to flush the channels, though as noted earlier, this may result in many null messages. Alternatively, each LP can maintain a counter of the number of messages it has sent, and the number if has received. When the sum of the send and receive counters across all of the LPs are the same, and each LP has reached the barrier point, it is guaranteed that there are no more transient messages in the system. In practice, summing the counters can be combined with the computation for computing the global minimum value [29].

To determine which events are safe, the link lookahead can be exploited [30–32]. The link lookahead notion is extended to determine the minimum "distance" between LPs, i.e., the minimum amount of simulation time that must elapse for an event in one LP to directly or indirectly affect another LP, and can be used by an LP to determine bounds on the timestamp of future events that it might receive from other LPs. Other techniques focus on maximizing exploitation of lookahead, e.g., see [33, 34].

Another thread of research in synchronization algorithms concerns relaxing ordering constraints in order to improve performance. Some approaches amount to simply ignoring out of order event processing [35, 36]. Use of time intervals, rather than precise timestamps, to encode uncertainty of temporal information in order to improve the performance of time management algorithms have also been proposed [37, 38]. Use of causal order rather than timestamp order for distributed simulation applications has also been studied [39].

5.3.3 Optimistic Synchronization

Optimistic event processing offers a different approach to synchronization. These methods allow violations to occur, but are able to detect and recover from them. Optimistic approaches offer two important advantages over conservative techniques. First, they can exploit greater degrees of parallelism. If a packet arrival event on one link *might* be affected by an arrival on another incoming link, but this second event never materializes, conservative algorithms will force the LP to block, while optimistic algorithms will allow the first event to be processed. Second, conservative algorithms rely on application-specific information (e.g., link delays) in order to determine which events are safe to process. While optimistic mechanisms can execute more efficiently if they exploit such information, they are less reliant on such information for correct execution. This allows the synchronization mechanism to be more transparent to the application program than conservative approaches, simplifying software development. On the other hand, optimistic methods may require more overhead computations than conservative approaches, leading to certain performance degradations.

Time warp [40] is the most well known optimistic method. When an LP receives an event with timestamp smaller than one or more events it has already processed, it rolls back and reprocesses those events in timestamp order. Rolling back an event involves restoring the state of the LP to that which existed prior to processing the event (checkpoints or other state saving techniques are used for this purpose), and "unsending" messages sent by the rolled back events using a mechanism called antimessages. An antimessage is a copy of a previously sent message; whenever an antimessage and its matching (positive) message are both stored in the same queue, the two are deleted (annihilated). A process need only send the corresponding antimessage to "unsend" a message. If the matching positive message has already been processed, the receiver process is rolled back, possibly producing additional antimessages. Using this recursive procedure all effects of the erroneous message will eventually be erased.

Rollback introduces two new problems that must be solved. First, certain computations, e.g., I/O operations, cannot be rolled back. Second, the computation will continually consume more and more memory resources because a history (e.g., checkpoints) must be retained; a mechanism is required to reclaim the memory used for this history information. Both problems are solved by *global virtual time* (*GVT*). GVT is a lower bound on the timestamp of any future rollback. GVT is computed by observing that rollbacks are caused by messages arriving "in the past." Therefore, the smallest timestamp among unprocessed and partially processed messages gives a value for GVT. Once GVT has been computed, I/O operations occurring at simulated times older than GVT can be committed, and storage older than GVT (except one state vector for each LP) can be reclaimed.

Computing GVT is essentially the same as computing LBTS values, described earlier. This is because rollbacks result from receiving a message or antimessage in the LP's past. Thus, GVT amounts to computing a lower bound on the timestamp of future messages (or antimessages) that may later be received. Several algorithms for computing GVT (LBTS) have been developed, e.g., see [29, 41], among others. Asynchronous algorithms compute GVT "in background" while the simulation computation is proceeding, introducing the difficulty that different processes must report their local minimum at different points in time. A second problem is that one must account for transient messages in the computation, i.e., messages that have been sent but not yet received. Mattern describes an elegant solution to these problems using consistent cuts of the computation and message counters, discussed earlier [29].

Time warp can suffer from some LPs advancing too far ahead of the others, leading to excessive memory utilization and long rollbacks. Several techniques have been proposed to address these problems by limiting optimistic execution. An early technique involves using a sliding window of simulated time [35]. The window is defined as [GVT, GVT + W] where W is a user-defined parameter. Only events with timestamp within this interval are eligible for processing. Another approach delays every message transmission until it is guaranteed that it will not be later rolled back, i.e., until GVT advances to the simulation time at which the event was scheduled. This eliminates the need for antimessages and avoids cascaded rollbacks, i.e., a rollback resulting in the generation of additional rollbacks [42, 43]. An approach that is also a local rollback mechanism to avoid antimessages using a concept called lookback (somewhat analogous to lookahead in conservative synchronization protocols) is described in [44, 45]. A technique called direct cancellation is sometimes used to rapidly cancel incorrect messages, thereby helping to reduce overly optimistic execution [46, 47]. Another line of research in optimistic synchronization focuses on adaptive approaches where the simulation executive automatically monitors the execution and adjusts control parameters to maximize performance. Examples of such adaptive control mechanisms are described in [48, 49], among others.

Optimistic synchronization usually requires more memory than conservative execution. Several techniques have been developed to address this problem. For example, one can roll back computations to reclaim memory resources [50, 51]. State saving can be performed infrequently rather than after each event [52, 53]. The memory used by some state vectors can be reclaimed even though their timestamp is larger than GVT [54].

Additional considerations are required when utilizing optimistic synchronization. Specifically, one must be able to roll back all operations, or be able to postpone them until GVT advances past the simulation time of the operation. Care must be taken to ensure that operations such as memory allocation and deallocation are handled properly, e.g., one must be able to roll back these operations. Also, one must be able to roll back execution errors. This can be problematic in certain situations, e.g., if an optimistic execution causes portions of the internal state of the time warp executive to be overwritten [55].

Rather than using state saving, another technique called reverse computation can be used to implement rollback [56]. Undoing an event computation is accomplished by executing the inverse computation, e.g., to undo incrementing a state variable, the variable is instead decremented. The advantage of this technique is that it avoids state saving, which may be both time consuming and require a large amount of memory. In [56] a reverse compiler is described to automatically generate inverse computations.

Depending on specifics of the application, either conservative or optimistic execution may be preferable. In general, if the application has good lookahead characteristics and programming the application to exploit this lookahead is not overly burdensome, conservative approaches are the method of choice. Indeed, much research has been devoted to improving the lookahead of simulation applications, e.g., see [57]. Otherwise, optimistic synchronization offers greater promise. Disadvantages of optimistic synchronization include the potentially large amount of memory that may be required, and the complexity of optimistic simulation executives. Techniques to reduce memory utilization further aggravate the complexity issue.

5.4 A CASE STUDY IN PARALLEL NETWORK SIMULATION

We now discuss results of a case study exploring the use of parallel discrete-event simulation techniques to improve the performance and scalability of network simulations. Several parallel discrete-event simulation systems have been developed to improve the scalability of network simulations. The traditional approach to realizing such a system is to create the parallel simulator "from scratch," where all the simulation software is custom designed for a particular parallel simulation engine. The simulation engine provides, at a minimum, services for communication and synchronization. Examples of parallel network simulators using this approach include GloMoSim [7], TeD [58, 59], SSFNet [60], DaSSF [61], TeleSim [62], and the ATM simulator described in [63], among others. One advantage of this approach is that the software can be tailored to execute efficiently in a specific environment. Because new models must be developed and validated, this approach requires a significant amount of time and effort to create a useable system.

Another approach to parallel/distributed simulation involves interconnecting existing simulators. These federated simulations may include multiple copies of the *same* simulator (modeling different portions of the network), or entirely *different* simulators. The individual simulators that are to be linked may be sequential or parallel. This approach has been widely used by the military to create simulation systems for training or analysis, and several standards have been developed in this regard [64–68]. An approach linking multiple copies of the commercial CSIM simulator to create parallel simulations of queues is described in [69]. The federated approach offers the benefits of model and software reuse, and provides the potential of rapid parallelization of existing sequential simulators. It also offers the ability to exploit models and software from different simulators in one system [70].

The case study described here focuses on the latter approach. Specifically, we are concerned with two parallel simulators: parallel/distributed network simulator (PDNS) based on the widely used *ns*2 network simulation tool, and GTNetS, a recently developed tool designed for scalable

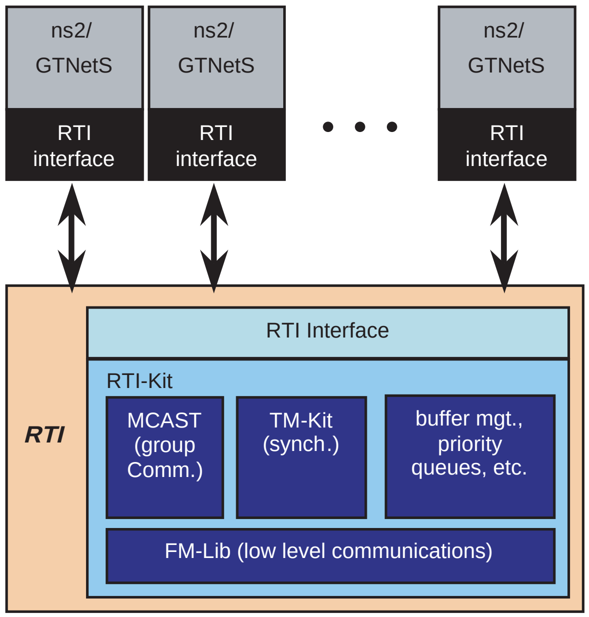


FIGURE 5.2: PDNS/GTNetS architecture

federated network simulation. Both systems use the same underlying runtime infrastructure (RTI) software that provides services for communication and synchronization (see Fig. 5.2). Each of these is described next.

5.4.1 RTI-Kit

RTI-Kit is a component of the federated simulations development kit (FDK). It provides a set of libraries for realizing RTI software. Specifically, the MCAST library provides group communication services for implementing publication/subscription communication among simulators (federates). The current version uses reliable, point-to-point communication to implement group communications. The TM-Kit library includes software to synchronize the computation, as was discussed earlier. The other modules shown in Fig. 5.2 implement mechanisms such as buffer management, priority queues, and low-level communications support.

As discussed earlier, synchronization mechanisms for parallel discrete-event simulation can be broadly classified as conservative or optimistic. The principal responsibility of the synchronization mechanism is to ensure that each simulator (called a *federate*) processes events in timestamp order. Conservative synchronization is better suited when federating existing sequential simulators because one need not add a rollback mechanism to the original simulator.

The synchronization mechanism used here is based on using a distributed snapshot algorithm to compute the lower bound on timestamp of future messages that might be received.

In addition to computing a global minimum, this algorithm must account for messages that have been sent, but have not yet been received, i.e., *transient messages*. Specifically, a variation of Mattern's algorithm [29] is used for this purpose. At the heart of TM-Kit is the computation of reductions (global minimums) that account for transient messages using message counters. This computation and performance optimizations that have been developed are described in [71].

5.4.2 PDNS

PDNS is based on the *ns*2 simulation tool, and is described in greater detail in [72]. Each PDNS federate differs from *ns*2 in two important respects. First, modifications to *ns*2 were required to support distributed execution. Specifically, a central problem that must be addressed when federating sequential network simulation software in this way is the global state problem. Each PDNS federate no longer has global knowledge of the state of the system. Specifically, one *ns*2 federate cannot directly reference state information for network nodes that are instantiated in a different federate. In general, some provision must be made to deal with both static state information that does not change during the execution (e.g., topology information) and dynamic information that does change (e.g., queue lengths). Fortunately, due to the modular design of the *ns*2 software, one need only address this problem for static information concerning network topology, greatly simplifying the global state problem.

To address this problem, a naming convention is required for an *ns*2 federate to refer to global state information. Two types of remote information must be accessed. The first corresponds to link end points. Consider the case of a link that spans federate boundaries, i.e., the two link end points are mapped to different federates. Such links are referred to in PDNS as *rlinks* (remote links). Some provision is necessary to refer to end points that reside in a different federate. This is handled in PDNS by using an IP address and a network mask to refer to any link end point. When configuring a simulation in PDNS, one creates a TCL script for each federate that instantiates the portion of the network mapped to the federate, and instantiates rlinks to represent the "edge-connections" that span the federates. The second situation where the global state issue arises concerns reference to end points for logical connections; specifically, the final destination for a TCP flow may reside in a different federate. Here, PDNS again borrows well-known networking concepts, and uses a port number and an IP address to identify a logical end point.

The second way that PDNS differs from *ns*2 is that it uses a technique called NIx-Vector routing. An initial study of *ns*2 indicated that routing table size placed severe limitations on the size of networks that could be simulated because the amount of memory required to store routing table information increased O(N^2), where N is the number of network nodes. To address this problem, message routes are computed dynamically, as needed. The path from source to destination is encoded as a sequence of (in effect) output ports numbers call the NIx-Vector, leading to a compact representation. Previously computed routes are cached to avoid repeated recomputation of the same path. This greatly reduces the amount of memory required, and greatly increases the size of network that can be simulated on a single node. The NIx-Vector technique is also applicable to the sequential version of *ns*2 and is used in PDNS.

5.4.3 GTNetS

GTNetS is a tool recently develop at Georgia Tech for scalable network simulation [73]. Network models are written in C++. Like PDNS, the parallel version of GTNetS was realized using a federated simulation approach. Unlike PDNS, the sequential version of GTNetS was designed with parallel processing in mind. As such, it benefits from the experiences derived from creating PDNS, and it overall is more scalable and easier to use than PDNS.

GTNetS uses many of the same techniques used in PDNS to enable parallel execution. In particular, it uses rlinks, IP addresses, and address masks to identify link end points, and an IP address and port number to identify a remote end host. GTNetS also uses NIx-Vector routing, as described earlier, to economize on the memory required to store routing table information.

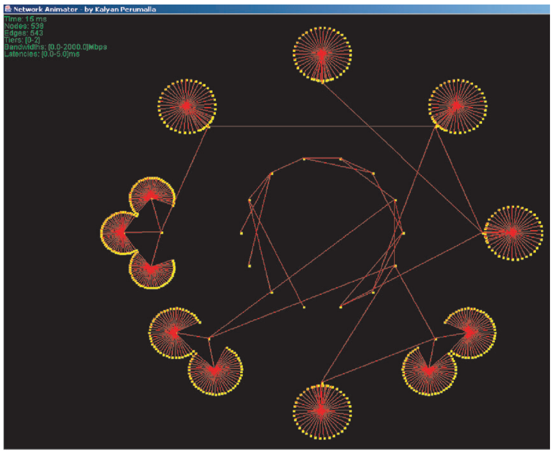
5.4.4 Methodology

The objective of the performance study was to stress test the PDNS and GTNetS simulations to quantitatively characterize the scale and performance of network simulations that can be done today. In the following we describe the methodology used to create and validate the network simulators. Specifically, we describe the networks that were used for this study. We then describe both sequential and parallel performance results on a Sun workstation, a Linux-based cluster, and a large-scale supercomputer.

The network topology, traffic, and parameters were based on a benchmark specification developed by the research group at Dartmouth College [60]. The benchmarks were developed as a set of baseline models for the network modeling and simulation community. The benchmark configurations were developed with the intention of facilitating the demonstration of network simulator scalability.

The basic "building block" used to construct large network configurations is referred to as a campus network (CN). The topology of a typical CN is shown in Fig. 5.3. Each CN consists of 4 servers, 30 routers, and 504 clients for a total of 538 nodes. The CN comprises four separate networks. Net 0 consists of three routers, where one node is the gateway router for the CN. Net 1 is composed of two routers and four servers. Net 2 consists of 7 routers, 7 LAN routers, and 294 clients. Net 3 contains 4 routers, 5 LAN routers, and 210 clients.

Net 0 is connected to Net 2 and Net 3 via standalone routers. Net 1 is connected directly to Net 0 through a single link. All nonclient links have a bandwidth of 2 Gbps and have

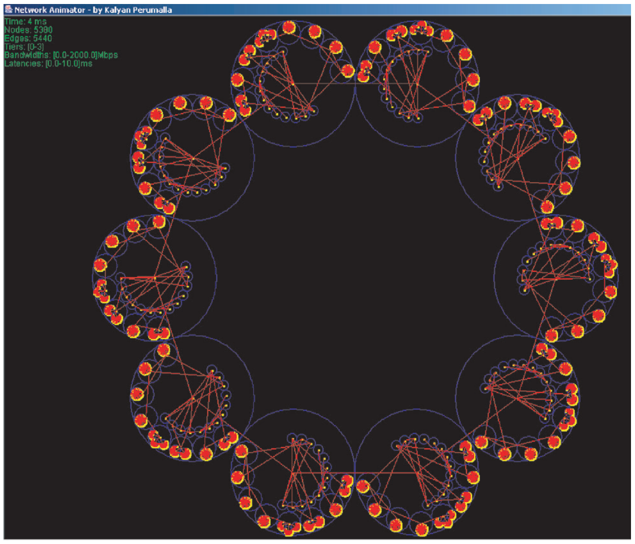


```
Time: 15 ms
Nodes: 538
Edges: 543
Tiers: [0-2]
Bandwidths: [0.0-2000.0]Mbps
Latencies: [0.0-5.0]ms
```

FIGURE 5.3: Topology of a single campus network

a propagation delay of 5 ms with the exception of the Net 0 to Net 1 links, which have a propagation delay of 1 ms. Clients are connected in a point-to-point fashion with their respective LAN routers and have links with 100 Mbps bandwidth and 1 ms delay.

Multiple CNs may be instantiated and connected together to form a ring topology. This aspect of the network allows the baseline model to be easily scaled to arbitrarily large sizes. Multiple CNs are interconnected through a high latency 200 ms 2 Gbps link via their Net 0 gateway router. A network consisting of 10 CNs is shown in Fig. 5.4.



```
Time: 4 ms
Nodes: 5380
Edges: 5440
Tiers: [0-3]
Bandwidths: [0.0-2000.0]Mbps
Latencies: [0.0-10.0]ms
```

FIGURE 5.4: Topology of ten campus networks configured in a ring

In our performance study, we focus on pure TCP traffic requested by clients from server nodes. All TCP traffic is "external" to the requesting CN clients, i.e., all the clients generate TCP traffic to/from servers in an adjacent CN in the ring (CN i communicates with CN $i+1$, etc.). Also, we use the short transfer case of the baseline model, where clients request 500,000 bytes from a random Net 1 server. The TCP sessions start at a time selected from a uniform distribution over the interval from 100 and 110 s of simulation time from the start of the run.

5.4.5 Validation

The simulators were validated by creating three separate implementations using different network simulation packages, and comparing the resulting statistics that were produced. Specifically, the campus network configuration was realized using PDNS, GTNetS, and Opnet; Opnet is a widely used commercial network simulation package.

A single campus network configuration was used for validation purposes. The simulators were run, and processed over 4.5 million packet transmissions. Because the different simulators use different random number generators and the model implementations differed slightly (e.g., the different packages use different means to specify network traffic), statistical results from the different packages were not identical. However, the connections with the minimum and maximum average end-to-end delays yielded average delays that differed by less than 3%, and the overall average delay across all connections also differed by less than 3%.

5.4.6 Hardware Platforms

Experiments were performed across a variety of platforms. Initial sequential experiments to establish baseline performance were conducted on a 450 MHz Sun UltraSPRAC-II machine running the Solaris operating system.

Parallel measurements were performed on both a cluster computing platform at Georgia Tech and a supercomputer at the Pittsburgh Supercomputing Center (PSC). The cluster computing platform is a large Linux cluster consisting of 17 machines, or a total of 136 CPUs. Each machine is a symmetric multiprocessor (SMP) machine with eight 550 MHz Pentium III XEON processors. The eight CPUs of each machine share 4 GB of RAM. Each processor contains 32 KB (16 KB Data, 16 KB Instruction) of nonblocking L1 cache and 2 MB of full-speed, nonblocking, unified L2 cache. An internal core interconnect utilizes a five-way crossbar switch connecting two four-way processor buses, two interleaved memory buses, and one I/O bus. The operating system is Red Hat Linux 7.3 running a customized 2.4.18-10 smp kernel. The 17 SMP machines are connected to each other via a dual gigabit Ethernet switch with EtherChannel aggregation. Our RTI software uses shared memory for communications within an SMP, and TCP/IP for communication across SMPs.

Supercomputer experiments were conducted on the “Lemieux” machine at PSC. This machine includes 750 HP–Alpha ES45 servers. Each server contains 4 GB of memory and four 1.0 GHz CPUs. A high-speed Quadrics switch is used to interconnect the servers.

5.4.7 Performance Measurements

Sequential performance of *ns*2 (version 2.1b9) and GTNetS in simulating a single CN on a Sun/Solaris machine and an Intel/Linux machine are shown in Table 5.1. As can be seen, both simulators execute on the order of 40 K to 44 K PTS on the Sun. A single Intel/Linux processor in the cluster machine executes *ns*2 simulations about 2.2 times faster than the same simulation running on the Sun/Solaris platform.

The parallel simulation experiments described here scale the size of the simulated network in proportion to the number of processors used. This is a widely accepted approach for scalability studies in the high performance computing community. It also circumvents the problem of having a sequential machine with enough memory to execute the entire model (necessary to compute speedup), which would not be possible for the large simulations that were considered here.

Fig. 5.5 shows the performance of PDNS on the Linux cluster as the problem size and hardware configuration are increased in proportion. It can be seen that performance increases linearly as the scale is increased. The largest configuration simulates 645,600 network nodes on 120 processors, yielding a performance of approximately 2 million PTS. Since these experiments were performed, optimizations to PDNS were later added, and found to yield aggregate performance of 5.5 million PTS using 136 processors in simulating a campus network configuration containing 658,512 nodes and 616,896 traffic flows. A “SynFlood” denial of service attack scenario with 25,000 attacking hosts and background traffic was also executed on the simulator, yielding a performance of 1.5 million PTS on 136 processors. This version of PDNS was based on *ns*2 version 2.26, which we observed to be somewhat slower than version 2.1b9 that was used for the previous experiments.

TABLE 5.1: Baseline PerformanceMeasurements

	*ns*2 (SUN)	GTNets (SUN)	*ns*2 (INTEL)
Events	9,107,023	9,143,553	9,117,070
Packet transmissions	4,546,074	4,571,264	4,552,084
Run time (seconds)	104	112.3	48
PTS	43,712	40,706	94,814

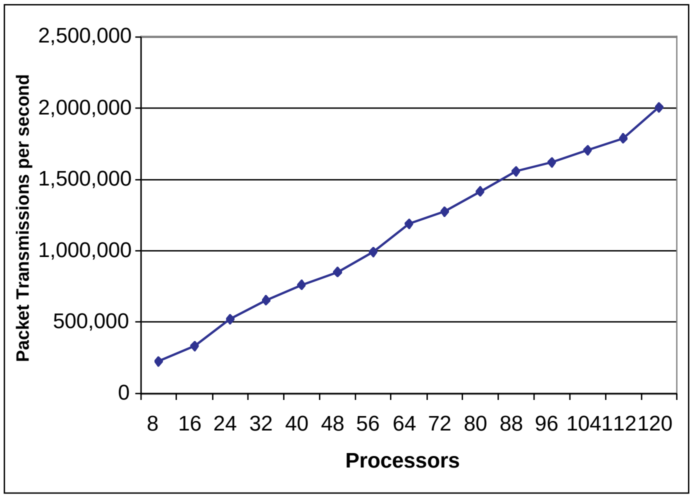


FIGURE 5.5: PDNS performance on a cluster of Intel SMPs; each processor simulates ten campus networks (up to 645,600 network nodes with 120 processors)

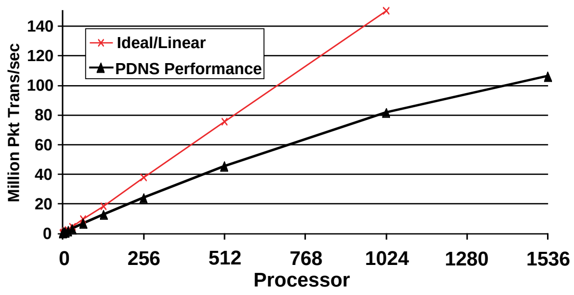


FIGURE 5.6: PDNS performance on Pittsburgh supercomputers

Performance measurements of PDNS executing on the PSC are shown in Fig. 5.6. The largest configuration contains approximately 4 million network nodes, and yields a simulator performance of 106 million PTS on 1536 processors.

Experiments with GTNetS on PSC also demonstrated that scalable executions could be obtained for that simulator. An execution containing 5.5 million nodes and 5.2 million flows on 512 processors yielded a performance of 12.3 million PTS. Another experiment demonstrated 1.0 million web browsers on a network containing 1.1 million nodes simulated in near real time, requiring 541 s of wallclock time to simulate 300 s of network operation. This experiment assumed that the HTTP traffic was generated using a widely used empirical model for web traffic, described in [74]. This experiment demonstrates that real-time packet-level simulation of million node networks is within reach today.

CHAPTER 6
The Future

The goal of this effort is to try to characterize quantitatively the capability of parallel simulation tools to simulate large-scale networks and to point out that the ability now exists to simulate large networks. This is by no means to imply scalable network simulation is a “solved problem!” Much additional research and development is required to effectively exploit these capabilities.

On the modeling side, creating realistic models of the Internet remains an extremely challenging problem. Some of the key issues that must be addressed are described in [75]. For example, the topology, configuration, and traffic of the Internet *today* is not well understood and is constantly changing, let alone the Internet of tomorrow that is targeted by most simulation studies. Techniques to effectively validate and experiment with large-scale network simulations must be developed. Much work is required to make the parallel simulation tools easily accessible and usable by network researchers who are not experts in parallel processing. Robust parallel performance is needed; modest changes to the configuration of the network being simulated may result in severe performance degradations. These and many other issues must be addressed before large-scale network simulation tools can reach their fullest potential.

As an example of the use of simulation, the telecommunication industry uses network simulation routinely to measure and evaluate the performance of voice over Internet protocol (VOIP) implementations, including VOIP gateway devices and VOIP protocols. Almost every new protocol utilizing the Internet protocol suite is first tested via network simulation in order to analyze its performance under a variety of conditions. Various design options of peer-to-peer networks are evaluated and compared via controlled scenarios. Network emulation is a common tool used for testing many commercial applications, including network-based streaming video client/server systems, and distributed gaming applications. Wireless network simulations are now widely used in a variety of civilian and military applications to evaluate the expected performance of network infrastructure or ad hoc wireless network design. This is especially true of testing critical command, control, and communications systems, which is an active research area for military applications.

Bibliography

[1] G. F. Riley, "The Georgia Tech Network Simulator," in *Proc. of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*. Karlsruhe, Germany: ACM Press, 2003.

[2] The Network Simulator – ns2 homepage, http://www.isi.edu/nsnam/ns/.

[3] G. Riley, M. Ammar, R. M. Fujimoto, A. Park, K. Perumalla, D. Xu, "A Federated Approach to Distributed Network Simulation," ACM Transactions on Modeling and Computer Simulation, Vol. 14, No. 1, pp. 116–148, April 2004. doi:10.1145/985793.985795

[4] B. K. Szymanski, et al. "Genesis: a system for large-scale parallel network simulation," *Workshop on Parallel and Distributed Simulation*, Washington D.C., USA, 2002.

[5] R. Fujimoto, et al. "Large-scale network simulation: how big? how fast?" in *Proc. 11th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'03)*, 2003.

[6] G. F. Riley, T. Jaafar, and R. M. Fujimoto, "Integrated fluid and packet network simulations," in *Proc. 10th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*, 2002.

[7] X. Zeng, R. Bagrodia, and M. Gerla, "GloMoSim: a library for parallel simulation of large-scale wireless networks," in *Proc. 1998 Workshop on Parallel and Distributed Simulation*, 1998, pp. 154–161.

[8] R. Brown, "Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem," *Commun. ACM*, Vol. 31, No. 10, pp. 1220–1227, 1988. doi:10.1145/63039.63045

[9] IETF, OSPF Version 2, IETF RFC 2328, 1998.

[10] G. F. Riley, R. M. Fujimoto, and M. Ammar, "Stateless routing in network simulations," in *Proc. 10th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'00)*, 2000.

[11] Y. Wardi, and G. F. Riley, "IPA for loss volume and buffer workload in Tandem SFM networks," in *Proc. 6th Workshop on Discrete Event Systems (WODES'02)*, 2002.

[12] Y. J. Lee, and G. F. Riley, "Efficient simulation of wireless networks using lazy MAC state update," in *19th Workshop on Principles of Advanced and Distributed Simulations(PADS'05)*, Monterey, CA, USA, 2005, pp. 131–140.doi:full_text

[13] X. Zhang, and G. F. Riley, "Performance of routing protocols in very large-scale mobile wireless ad hoc networks," in *Symp. on Modeling, Analysis, and Simulation of Computer and Telecommuncation Systems*, Atlanta, GA, USA, 2005.

[14] A. M. Law, and W. D. Kelton, *Simulation Modeling and Analysis*, 3rd ed. Boston, MA: McGraw-Hill, 2000.

[15] Y. B. Lin, and E. D. Lazowska, "A time-division algorithm for parallel simulation," *ACM Trans. Model. Comput. Simul.*, Vol. 1, No. 1, pp. 73–83, 1991.doi:10.1145/102810.214307

[16] S. Andradottir, and T. Ott, "Time-segmentation parallel simulation of networks of queues with loss or communication blocking," *ACM Trans. Model. Comput. Simul.*, Vol. 5, No. 4, pp. 269–305, 1995.doi:10.1145/226275.226276

[17] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani, "Algorithms for unboundedly parallel simulations," *ACM Trans. Comput. Syst.*, Vol. 9, No. 3, pp. 201–221, 1991. doi:10.1145/128738.128739

[18] H. Wu, R. M. Fujimoto, and M. Ammar, "Time-parallel trace-driven simulation of csma/cd," in *Proc. Workshop on Parallel and Distributed Simulation*, 2003.

[19] K. G. Jones, and S. R. Das, "Time-parallel algorithms for simulation of multiple access protocols," in *9th Int. Symp.on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2001.

[20] A. Greenberg, et al., "Efficient massively parallel simulation of dynamic channel assignment schemes for wireless cellular communications," in *Proc. 8th Workshop on Parallel and Distributed Simulation*, 1994, pp. 187–194.

[21] R. M. Fujimoto, I. Nikolaidis, and A. C. Cooper, "Parallel simulation of statistical multiplexers," *J. Discrete Event Dyn. Syst.*, Vol. 5, pp. 115–140, 1995.

[22] B. K. Szymanski, Y. Liu, and R. Gupta, "Parallel network simulation under distributed genesis," in *Proc. 17th Workshop on Parallel and Distributed Simulation*, 2003, pp. 61–68.

[23] F. Baccelli, and M. Canales, "Parallel simulation of stochastic petri nets using recurrence equations," in *ACM Trans. Model. Comput. Simul.*, Vol. 3, No. 1, pp. 20–41, 1993. doi:10.1145/151527.151545

[24] V. Jha, and R. Bagrodia, "Simultaneous events and lookahead in simulation protocols," *ACM Trans. Model. Comput. Simul.*, Vol. 10, No. 3, pp. 241–267, 2000.

[25] H. Mehl, "A deterministic tie-breaking scheme for sequential and distributed simulation," in *Proc. of the Workshop on Parallel and Distributed Simulation*, Society for Computer Simulation, 1992, pp. 199–200.

[26] K. M. Chandy and J. Misra, "Distributed simulation: a case study in design and verification of distributed programs," *IEEE Trans. Softw. Eng.*, Vol. SE-5, No. 5, pp. 440–452, 1978.

[27] R. E. Bryant, "Simulation of packet communications architecture computer systems," MIT-LCS-TR-188, 1977.

[28] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, Vol. 24, No. 4, pp. 198–205, 1981. doi:10.1145/358598.358613

[29] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *J. Parallel Distrib. Comput.*, Vol. 18, No. 4 pp. 423–434, 1993. doi:10.1006/jpdc.1993.1075

[30] B. D. Lubachevsky, "Efficient distributed event-driven simulations of multiple-loop networks," *Commun. ACM*, Vol. 32, No. 1, pp. 111–123, 1989.doi:10.1145/63238.63247

[31] W. Cai, and S. J. Turner, "An algorithm for distributed discrete-event simulation—the "carrier null message" approach," in *Proc. of the SCS Multiconference on Distributed Simulation*, SCS Simulation Series, 1990, pp. 3–8.

[32] R. Ayani, "A parallel simulation scheme based on the distance between objects," in *Proc. of the SCS Multiconference on Distributed Simulation*, Society for Computer Simulation, 1989, pp. 113–118.

[33] R. A. Meyer, and R. L. Bagrodia, "Path lookahead: a data flow view of PDES models," in *Proc. 13th Workshop on Parallel and Distributed Simulation*, 1999, pp. 12–19.

[34] Z. Xiao, et al., "Scheduling critical channels in conservative parallel simulation," in *Proc. 13th Workshop on Parallel and Distributed Simulation*, 1999, pp. 20–28.

[35] L. M. Sokol, and B. K. Stucky, "MTW: experimental results for a constrained optimistic scheduling paradigm," in *Proc. of the SCS Multiconference on Distributed Simulation*, 1990, pp. 169–173.

[36] D. M. Rao, et al., "Unsynchronized parallel discrete event simulation," in *Proc. of the Winter Simulation Conference*, 1998, pp. 1563–1570.

[37] R. M. Fujimoto, "Exploiting temporal uncertainty in parallel and distributed simulations," in *Proc. 13th Workshop on Parallel and Distributed Simulation*, 1999, pp. 46–53.

[38] R. Beraldi and L. Nigro, "Exploiting temporal uncertainty in time warp simulations," in *Proc. 4th Workshop on Distributed Simulation and Real-Time Applications*, 2000, pp. 39–46.

[39] B.-S. Lee, W. Cai, and J. Zhou, "A causality based time management mechanism for federated simulations," in *Proc. 15th Workshop on Parallel and Distributed Simulation*, 2001, pp. 83–90.

[40] D. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, Vol. 7, No. 3, pp. 404–425, 1985.doi:10.1145/3916.3988

[41] B. Samadi, "Distributed simulation, algorithms and performance analysis," Computer Science Department, University of California, Los Angeles, Los Angeles, CA, 1985.

[42] P. M. Dickens, and P. F. J. Reynolds, "SRADS with local rollback," in *Proc. of the SCS Multiconference on Distributed Simulation*, 1990, pp. 161–164.

[43] J. S. Steinman, "SPEEDES: a multiple-synchronization environment for parallel discrete event simulation," *Int. J. Comput. Simul.*, Vol. 2, pp. 251–286, 1992.

[44] G. Chen and B. K. Szymanski, "Lookback: a new way of exploiting parallelism in discrete event simulation," in *Proc. 16th Workshop on Parallel and Distributed Simulation*, 2002, pp. 153–162.

[45] G. Chen and B. K. Szymanski, "Four types of lookback," in *Proc. 17th Workshop on Parallel and Distributed Simulation*, 2003, pp. 3–10.

[46] R. M. Fujimoto, "Time warp on a shared memory multiprocessor," *Trans. Soc. Comput. Simul.*, Vol. 6, No.3, pp. 211–239, 1989.

[47] J. L. Zhang and C. Tropper, "The dependence list in time warp," in *Proc. 15th Workshop on Parallel and Distributed Simulation*, 2001, pp. 35–45.doi:full_text

[48] A. Ferscha, "Probabilistic adaptive direct optimism control in time warp," in *Proc. 9th Workshop on Parallel and Distributed Simulation*, 1995, pp. 120–129.doi:full_text

[49] S. R. Das, and R. M. Fujimoto, "Adaptive memory management and optimism control in time warp," *ACM Trans. Model. Comput. Simul.*, Vol. 7, No. 2, pp. 239–271, 1997. doi:10.1145/249204.249207

[50] D. R. Jefferson, "Virtual time II: storage management in distributed simulation," in *Proc. 9th Annual ACM Symposium on Principles of Distributed Computing*, 1990, pp. 75–89. doi:full_text

[51] Y.-B. Lin and B. R. Preiss, "Optimal memory management for time warp parallel simulation," *ACM Trans. Model. Comput. Simul.*, Vol. 1, No. 4, pp. 283–307, 1991. doi:10.1145/130611.130612

[52] Y.-B. Lin, et al., "Selecting the checkpoint interval in time warp simulations," in *Proc. 7th Workshop on Parallel and Distributed Simulation*, 1993, pp. 3–10.

[53] A. C. Palaniswamy and P. A. Wilsey, "An analytical comparison of periodic checkpointing and incremental state saving," in *Proc. 7th Workshop on Parallel and Distributed Simulation*, 1993, pp. 127–134.

[54] B. R. Preiss and W. M. Loucks, "Memory management techniques for time warp on a distributed memory machine," in *Proc. 9th Workshop on Parallel and Distributed Simulation*, 1995, pp. 30–39.doi:full_text

[55] D. M. Nicol and X. Liu, "The dark side of risk," in *Proc. 11th Workshop on Parallel and Distributed Simulation*, 1997, pp. 188–195.doi:full_text

[56] C. D. Carothers, K. Perumalla, and R. M. Fujimoto, "Efficient optimistic parallel simulation using reverse computation," *ACM Trans. Model. Comput.Simul.*, Vol. 9, No 3, pp. 224–253, 1999.doi:10.1145/347823.347828

[57] E. Deelman, et al., "Improving lookahead in parallel discrete event simulations of

large-scale applications using compiler analysis," in *Proc. 15th Workshop on Parallel and Distributed Simulation*, 2001, pp. 5–13.doi:full_text

[58] K. Perumalla, R. Fujimoto, and A. Ogielski, "TeD—a language for modeling telecommunications networks," *Perform. Eval. Rev.*, Vol. 25, No. 4, 1998.

[59] A. L. Poplawski and D. M. Nicol, "Nops: a conservative parallel simulation engine for TeD," in *12th Workshop on Parallel and Distributed Simulation*, 1998, pp. 180–187.

[60] J. H. Cowie, D. M. Nicol, and A. T. Ogielski, "Modeling the global internet," *Comput. Sci. Eng.*, Vol. 1, No. 1, pp. 42–50, 1999.doi:10.1109/5992.743621

[61] J. Liu, and D. M. Nicol, *DaSSF 3.0 User's Manual*, 2001 (Available from: http://www.cs.dartmouth.edu/research/DaSSF/Papers/dassf-manual.ps).

[62] B. Unger, "The telecom framework: a simulation environment for telecommunications," in *Proc. 1993 Winter Simulation Conference*, 1993.

[63] C. D. Pham, H. Brunst, and S. Fdida, "Conservative simulation of load-balanced routing in a large ATM network model," in *Proc. 12th Workshop on Parallel and Distributed Simulation*, 1998, pp. 142–149.

[64] D. C. Miller and J. A. Thorpe, "SIMNET: the advent of simulator networking," in *Proc. IEEE*, Vol. 83, No. 8, pp. 1114–1123, 1995.doi:10.1109/5.400452

[65] IEEE Std 1278.1-1995, *IEEE Standard for Distributed Interactive Simulation—Application Protocols*. New York, NY: Institute of Electrical and Electronics Engineers, Inc., 1995.

[66] IEEE Std 1278.2-1995, *IEEE Standard for Distributed Interactive Simulation—Communication Services and Profiles*. New York, NY: Institute of Electrical and Electronics Engineers, Inc., 1995.

[67] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture for Simulation*. Englewood Cliffs, NJ: Prentice Hall, 1999.

[68] IEEE Std 1516.3-2000, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Interface Specification*. New York, NY: Institute of Electrical and Electronics Engineers, Inc., 2000.

[69] D. Nicol and P. Heidelberger, "Parallel execution for serial simulators," *ACM Trans. Model. Comput. Simul.*, Vol. 6, No. 3, pp. 210–242, 1996.doi:10.1145/235025.235031

[70] K. Perumalla, et al., "Experiences applying parallel and interoperable network simulation techniques in on-line simulations of military networks," in *Proc. 16th Workshop on Parallel and Distributed Simulation*, 2002, pp. 97–104.

[71] K. S. Perumalla, et al., "Scalable RTI-based parallel simulation of networks," in *Proc. 17th Workshop on Parallel and Distributed Simulation*., 2003, pp. 97–104.

[72] G. Riley, R. M. Fujimoto, and M. Ammar, “A generic framework for parallelization of network simulations,” in *Proc. 7th Int. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 1999, pp. 128–135.

[73] G. F. Riley, “The Georgia Tech Network Simulator,” in *Proc. of the Workshop on Models, Methods, and Tools for Reproducible Network Research (MoMe Tools)*, 2003.

[74] B. A. Mah, “An empirical model of HTTP network traffic,” in *INFOCOM*, 1997, pp. 592–600.doi:full_text

[75] S. Floyd, and V. Paxson, “Difficulties in simulating the internet,” *IEEE/ACM Trans. Netw.*, Vol. 9, No. 4, pp. 392–403, 2001.doi:10.1109/90.944338

Author Biographies

Richard M. Fujimoto is a Professor and Chair of the Computational Science and Engineering Division in the College of Computing at the Georgia Institute of Technology. He received the Ph.D. and M.S. degrees from the University of California (Berkeley) in 1980 and 1983 (Computer Science and Electrical Engineering) and the B.S. degrees from the University of Illinois (Urbana) in 1977 and 1978 (Computer Science and Computer Engineering). He has been an active researcher in the parallel and distributed simulation community since 1985. Among his current activities he is the technical lead concerning time management issues for the DoD high level architecture (HLA) effort. He served as an area editor for ACM Transactions on Modeling and Computer Simulation and Co-Editor-in-Chief of the journal Simulation: Transactions of the SCS, and has also been chair of the steering committee for the Workshop on Parallel and Distributed Simulation (PADS) since 1990. He also served as a member of the Conference Committee for the Simulation Interoperability Workshop.

Kalyan S. Perumalla is a senior researcher in the Computational Sciences and Engineering Division at the Oak Ridge National Laboratory. He also holds an adjunct faculty appointment with the College of Computing, Georgia Tech. Dr. Perumalla has over 10 years of research and development experience in the area of parallel and distributed simulation systems, and has published widely on these topics. He was instrumental in developing widely disseminated tools such as the μsik scalable simulation engine for simulations using 1000 or more processors, the Telecommunications Description Language (TeD) for automatically simulating networks in parallel, and the Federated Simulations Development Kit (FDK), a high-performance run-time infrastructure. He has also built several additional research prototype systems and tools (e.g., for distributed debugging, network modeling and parallel optimization). He received a Ph.D. in Computer Science from Georgia Tech in 1999. Dr. Perumalla has served as an investigator on multiple federally funded projects on scalable parallel/distributed discrete event simulation systems. He can be reached via email at perumallaks@ornl.gov, and his homepage is at www.ornl.gov/~2ip.

George F. Riley is an Assistant Professor of Electrical and Computer Engineering at the Georgia Institute of Technology. He received his Ph.D. in Computer Science from the

Georgia Institute of Technology, College of Computing, in August 2001. His research interests are large-scale simulation using distributed simulation methods. He is the developer of Parallel/Distributed *ns2* (PDNS) and the Georgia Tech Network Simulator (GTNetS). He can be reached via email at riley@ece.gatech.edu.