IRLSim: A General Purpose Packet Level Network Simulator

Andreas Terzis, Konstantinos Nikoloudakis, Lan Wang , Lixia Zhang
University of California, Los Angeles
terzis@cs.ucla.edu, nikolud@cs.ucla.edu, lanw@cs.ucla.edu, lixia@cs.ucla.edu

Abstract

Simulation is the main tool for studying networking protocols before deploying them in a wide scale, or for understanding how they are expected to behave under various conditions. IRLSim is a new packet level network simulator that we developed in the hope to study several Internet protocols. From its modest inception as a simulator for the RSVP signaling protocol, IRLSim has evolved into a more general purpose, easy to use, scalable simulator that can be used as a guide for studying existing network protocols as well as a research tool for developing new protocols. This paper describes the architecture of IRLSim in detail, presents its use in the study of a few specific networking problems and argues about its usefulness amongst the variety of other simulators.

Keywords: Packet level network simulator, RSVP, QoS, transport layer, PARSEC

1 Introduction

IRLSim is a general purpose packet level network simulator implemented at UCLA's Internet Research Lab. While IRLSim started as a stand-alone simulator for the RSVP [5] signaling protocol, over the past year it has been significantly enhanced to include a complete network layer as well as transport layer protocols such as TCP and UDP. To ease the creation of simulation scenarios and the analysis of simulation results, users can create simulation topologies using a Java Graphical User Interface we have developed and view animations of their results using the *nam* animator ([10]).

A question that may naturally arise is *why another packet level network simulator?* We believe that IRLSim has three strong points:

1. **Scalability.** IRLSim is written in Parsec, a simulation language that supports a wide range of sequential and parallel architectures. Using this feature the simulator code written for a uniprocessor can run on parallel machines with trivial changes, exploiting the speedup that these parallel machines can provide.
2. **Code Portability.** Since Parsec is very similar to C, simulator code written in Parsec can be easily moved to production code and vice versa. Thereby, existing network code can be effortlessly added to the simulator and conversely code developed in the simulator as a research prototype can be moved to production faster and with fewer porting errors.
3. **Ease of use.** IRLSim is useful to novice users as well as experts. Novice users can start creating their simulation scenarios using the simulator's graphical user interface. In fact, novice users can create simulations and get meaningful results that they can further use, without having to write a single line of code. This way, our simulator can be used as a learning tool as well as a research instrument.

The rest of the paper is structured as follows: In Section 1.1 we present the guiding principles that shaped the simulator's design. Section 1.2 introduces the Parsec simulation language, which is the language the simulation engine is written with. We continue in Section 2, where the simulator's internal architecture is revealed. Sections 3, 4, 5 and 6 present the network, routing, RSVP and transport layers respectively. The simulator's Graphical User Interface is discussed in Section 7 while in Section 8 we show how IRLSim was used for various network studies we have done. We conclude with a presentation of related work in Section 9 and our future work items in Section 10.

1.1 Design Principles

Before we present the simulator's architecture, we would like to present the design principles we have followed in developing IRLSim. In this way some of the decisions we have made will be clear. As we have already mentioned *scalability* is our prime concern. For simulation results to be applicable in the Internet these days, topologies simulated have to be of large scale and involve complex interactions among the various network elements. As we will see later on, the choice of Parsec and some associated techniques provide us with essentially unlimited capacity to scale.

Power and scalability are of little use when the simulator is difficult to use. We want our simulator to be useful to beginners and to expert users alike, and so we require an accessible interface that users can easily create the scenarios they want to simulate. For this reason, we have included a *Graphical User Interface* by which even first-time users can interactively create the requested simulation scenario. In comparison, other simulators, such as *ns* [3], present potential users with a steep *learning curve* since they require learning some specialized *scripting language* to describe the simulation topology.

Following the same principle of *usability*, we also pose the requirement that users can easily extend the simulator and moreover, the code that they write for the simulator can be easily ported to real protocol implementations. In order to achieve this goal, our simulator architecture is much similar to the architecture of a real networking protocol stack. Experienced users can apply their prior knowledge of network protocols stacks, such as the Unix BSD stack, [17] which have been widely studied [29], to understand the structure of our simulator and write new modules for it. On the other hand, other simulators have a much different architecture from real implementations. Two are the major consequences of this difference: first, users have to familiarize themselves with the simulator architecture and second, they have to *port* the code written in the simulator back to the architecture of a real implementation if they want to implement their ideas in real networks. In addition to the extra effort and time required in the porting of code from one architecture to the other, another problem is that of *minor inconsistencies* (e.g. timers) in the protocol implementations in the two architectures that might negative effects in protocol performance and behavior.

1.2 The Parsec simulation language

The simulation engine of IRLSim was written in PARSEC [2], a simulation language developed at the UCLA Parallel Computing Lab. The structure of the simulation engine is heavily influenced by the language's character and capabilities. For this reason, before we delve into the structure of the simulation engine in the next section, we will present Parsec's features in the rest of this section.

The Parsec language is based on C, but introduces several new features. A Parsec program consists of a set of *entities* and C functions. Each entity is a logical process that models a corresponding physical process; entities can be created and destroyed dynamically. Entities communicate by exchanging *messages*. Each message carries a logical time stamp matching the time at which the corresponding event happens in the physical system. Every Parsec program must include an entity called driver. This entity serves a purpose similar to the main function of a C program. Execution of a Parsec program is initiated by executing the first statement in the body of entity driver. The purpose of the driver entity is usually to create and dispatch duties to the other entities.

Simulating a physical system in PARSEC requires two steps: 1. finding the physical processes that communicate with each other and 2. enumerating the different messages exchanged among the communicating processes. Once this is done, physical processes can be mapped to PARSEC entities while the actual information exchange between physical processes can be modeled by the exchange of PARSEC messages between the corresponding PARSEC entities.

One important feature of Parsec is its ability to execute a discrete-event simulation model using several different asynchronous parallel simulation protocols on a variety of parallel architectures. Parsec is designed to cleanly separate the description of a simulation model from the underlying simulation protocol, sequential or parallel, used to execute it. Thus, with few modifications, a Parsec program may be executed using the traditional sequential simulation protocol or one of many parallel optimistic or conservative protocols.

One of PARSEC's major design goals is to facilitate migration of simulation models into operational software. PARSEC is built around a thread-based message-passing programming kernel called MPC (message-passing C), which can be used to develop general-purpose parallel programs. The only difference between PARSEC and MPC is that a PARSEC model executes in logical time, with all messages in the system being processed in the global order of their time stamps. In contrast, each entity in an MPC program can autonomously process messages in the physical order of their arrival. Because of the common set of message-passing primitives used in both environments, it is relatively easy to transform a Parsec simulation model into operational parallel software in MPC or even multiple instances of the same programs running on multiple machines communicating over an actual network (the later case is true when developing network simulations).

2 Simulator Architecture

IRLSim is divided into two major components: the *Graphical User Interface* (GUI) and the *simulation engine*. Users interact with the GUI to create simulation topologies and to modify existing ones. These two components communicate with each other via a TCP socket. An advantage of this communication method is that the two components do not have to be on the same machine. The simulation engine can run on a powerful parallel supercomputer while the User Interface runs on the user's workstation. The architecture of the User Interface is explained in detail in Section 7, while the rest of this section is devoted to exposing the inner workings of the simulation engine.

As Figure 1 shows, the simulation engine follows a mod-

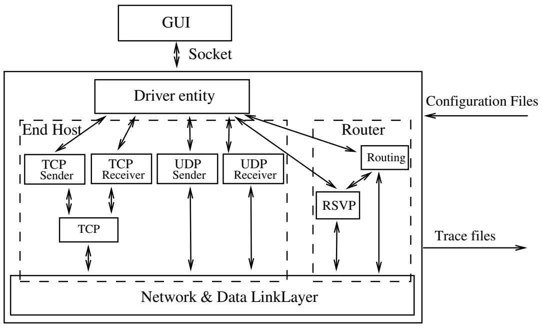

Figure 1. The IRLSim Simulator Architecture

ular design. An important benefit of this design is that is relatively easy to replace any of the modules (such as the routing protocol entities or the entity simulating the physical transmission channel) without affecting the rest. A driver entity is responsible for creating the rest of the entities and for communicating with the User Interface. The remaining components of the simulator engine are structured in a way that reflects the TCP/IP network protocol stack. At the bottom of the simulator stack, we have network and data link layers. The data link layer simulates physical links and their associated MAC (Media Access Control) protocols. The network layer entities simulate the IP layer in real networks which is responsible for packet delivery. These entities learn how to forward packets to their destinations by querying the routing entities. Routing entities receive the network topology from the driver entity and compute the routing table for each network node by running unicast and multicast routing protocols. Transport layer entities receive data from application layer and encapsulate the data in packets before handling it to the network layer. When the packets reach their final destinations, they are delivered back to the peering transport entities.

While routers forward packets, it is application hosts that generate and consume those data packets. We have written entities that simulate applications such as FTP or Telnet as well as streaming media applications. While streaming media applications run directly on top of the network layer, reliable applications run on top of a transport layer protocol called TCP that provides reliability and congestion control. Applications that require higher level network services invoke RSVP to reserve resources for their data flows. The RSVP entities simulate the RSVP protocol [5] running on network routers.

In the section that follows we talk about the association between the network nodes being simulated and the entities that implement the various node functionalities.

2.1 Scalability

As we have already explained in Section 1.2 the first approach that comes to mind, when simulating a physical network is to map each network node to a simulation entity. This way the exchange of packets between physical network nodes is simulated by the exchange of PARSEC messages between simulation entities. While this approach is both intuitive and simple it has one major disadvantage: as network topologies increase in size, so does the context switching overhead that results from having to switch among all the threads that implement the Parsec entities. The result of this is increased simulation time. Furthermore, since each entity has a fixed amount of *memory overhead* required for internal PARSEC book-keeping, as the number of entities increases so does the amount of memory required to run the simulation.

Our approach in solving this issue is what is called *entity aggregation*. Using this technique, a single Parsec entity simulates multiple physical nodes. When physical nodes mapped to the same Parsec entity exchange packets, the entity sends messages to itself and no context switch has to occur. Context switches occur only when physical nodes mapped to different simulation entities exchange messages. By appropriately allocating physical nodes to simulation entities we can exploit traffic localities and reduce the number of context switches between the simulation entities.

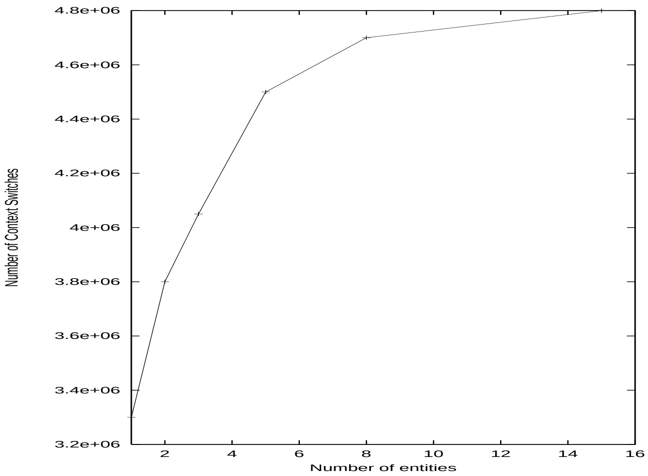

Figure 2. The effect of entity aggregation on context switches

Figures 2 illustrates the effects of entity aggregation. We have used as an example a topology that contains 15 nodes and we measured the number of context switches among the various simulation entities for different number of entities per physical entity. In Figure 2 we can see that the number of context switches is lowest when there is only one entity for all the physical nodes (that is one entity per layer) while the number of context increases as the number of entities increases. The knee of the curve occurs at five entities (that is when each entity simulates three physical nodes) because in the particular topology we have used for this experiment and for the current algorithm of distributing nodes among entities, most messages are exchanged between nodes belonging to different entities and so higher aggregation does not provide any additional improvements.

3 Network and Data Link Layers

We begin our description of the simulator architecture with the lowest layers seen in Figure 1, namely the Network and Data Link layers. These two layers simulate the IP layer and data link layers of the Internet architecture and are responsible for transferring data packets among the various simulated network entities such as end-hosts and routers.

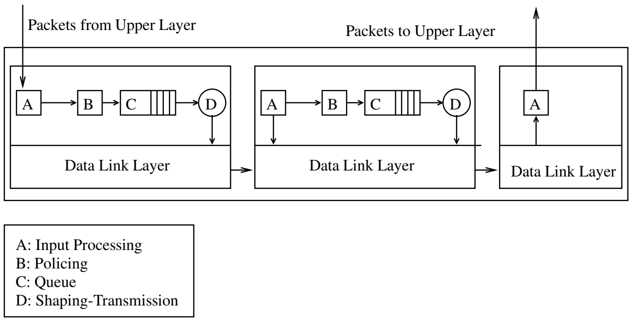

Figure 3. The Network Layer Architecture

As Figure 3 shows, packets from upper layers are submitted to the network layer and are then forwarded hop-by-hop until they reach their final destination(s). The network layer functionality in each host is divided in four main components: 1. *Input Processing*, 2. *Policing and Shaping*, 3. *Queuing* and 4. *Transmission*. We are going to talk about Policing and Shaping in Section 3.1, where we talk about the Quality of Service enhancements to the network layer. Steps 1, 3 and 4 are discussed in the following paragraphs.

Input Processing contains all the steps that are performed when a packet arrives at a network node. For example for multicast packets, a *reverse path* check must be performed, that is, the router reads the packet's source address and checks if the packet arrives from the interface that the router would use to *send* packets destined for this address. If the packet arrived from a different interface the packet is dropped. If the packet is destined for the current node, or if it is an RSVP packet (see Section 5 for further discussion about RSVP) then the packet is delivered to the appropriate upper layer, otherwise the packet has to be forwarded to the next hop towards it's destination(s).

If a received packet has to be forwarded then the network entity sends a routing query to the routing entity asking for the next hop(s) that the packet should be forwarded to. The query message contains the packet's destination and for multicast packets it also contains the packet's source address, since for multicast routing protocols such as DVMRP [8] source address is also important in making routing decisions. Once the routing protocol entity replies with a list of output interfaces, the packet is ready to be enqueued.

Packets ready for transmission through a particular interface are placed on that interface's output queue. Whether the packet is placed on the queue depends on the type of the output queue. At this point, two types of output queues are supported:

1. FIFO queues. Packets are placed at the end of this queue and are transmitted on a first-come, first-served basis. If the queue is full the packet is dropped. For this reason these queues are also called *Tail Drop* queues.

2. RED queues. These queues implement the RED [12] active queuing mechanism. According to this mechanism, upon arrival a packet could be randomly dropped even if the queue is not completely full. The reason to do so, is to try to control the size of the queue by signaling co-operating end protocols (e.g. TCP) to reduce their sending rate.

Users can select which kind of queue is used for each link using the GUI or the configuration file. Once the packet is enqueued, it stays in the queue awaiting its transmission. The packet at the queue's head is handed to the *data link* protocol for transmission. While at this point we have only implemented a simple data-link protocol that simulates point-to-point links with fixed propagation delay and capacity, more complex data link protocols (such as CSMA/CD or even wireless data link protocols such as MACA) can also be easily incorporated. Once the packet at the head of the queue is transmitted, the queue size is reduced by one and if there is another packet ready to be transmitted, it is passed to the data link functions.

One important feature of the network layer entities is that they support extensive logging facilities similar to the *Berkeley Packet Filter* (BPF) [16]. The logging facilities record various events such as packet arrivals and departures from a certain network node and users can easily obtain plots that show queue size variation with scripts we have written. Users can also select the level of logging they need.

3.1 QoS Capabilities

To support our work in *Differentiated Services*, which we present later in Section 8.2, we have enhanced the network layer of our simulator with capabilities to provide different Qualities of Service (QoS). Providing QoS in the network layer fundamentally involves two tasks:

1. Classifying packets to their corresponding service classes.

2. Implementing methods to provide the various different treatments to packets according to the service class they belong to. Examples of such methods include scheduling and buffer management techniques.

The first step taken once a packet is received at a network node is finding the service class the packet belongs to. This task is part of the Input Processing phase shown in Figure 3. Packets are categorized according to the value of the DS field [19] in the packet's IP header. Once the packets have been categorized they can be further treated according to the service class they belong to.

So far we have implemented only one service in addition to the normal *best effort* service. This service is the *Expedited Forwarding* (EF) service [13] that provides a low loss, low latency, low jitter and assured bandwidth network service. In order to provide a low latency and low jitter network service, EF packets must experience little if any, queuing delay inside network nodes. To achieve this effect, two things have to be done: 1. The packet incoming rate must always be smaller that the packet departure rate. If this condition isn't met, as queuing theory shows us queues will build inside the node and queuing delay will be created, 2. EF packets should not be preempted by best effort packets for more than a single packet transmission time.

To satisfy these two requirements, we have added *policing* of incoming EF packets and we also added a second queue for EF packets at the output interface, that has priority over the best effort queue. The rate of incoming EF packets is compared against a *token bucket* profile. The token bucket is filled with token at a configured rate. For each incoming EF packet, the number of tokens is reduced by one. If a packet arrives when the token bucket is empty, the packet is discarded. This way the incoming rate of EF packets is controlled and will always be smaller than the service rate. EF packets are put in a separate output FIFO queue that has priority over the best effort queue. If the EF queue is not empty then the next packet transmitted is picked from the EF queue. To make sure that the best effort queue is not starved there is an upper limit on the amount of resources that can be used by the EF queue (i.e the EF queue can only use a fixed percentage ($\leq 50\%$) of the interface's capacity). The output of the EF queue is *shaped* so that packets exit the queue at regular intervals without creating any traffic bursts. To do so, the service rate of the output queue is set to the sum of the rates of all incoming traffic that is directed towards that specific output link.

4 Routing Layer

Entities at the routing layer are responsible for computing the routing tables used at each network node. The information in the routing tables is used to forward packets from one node to the next as they travel from the source host to their final destination(s). The first step in computing the routing tables is finding each node's neighbors, that is the nodes directly connected to that node. This information can be extracted from the simulated network topology created through the simulator's GUI or loaded through a configuration file. The GUI communicates this information to the simulator's driver entity which in turn forwards the list of neighbors for every node to the routing layer entities. Using this information, each routing layer entity computes the routing tables for the network nodes it serves (see Section 2.1 for node aggregation).

In our current architecture, routing table entries for unicast destinations are computed during simulation initialization time and are static. This is not restricting for now, since there aren't any events such as node or link failures that could change the initial topology. For each node, Dijkstra's algorithm is used to compute the least distance path to each destination. All links in this computation have unit length, while a simple extension would be each link's metric to be proportional to the link's propagation delay. The multicast routing protocol in our simulator[1] is DVMRP [8]. The initial multicast delivery tree is computed by the driver entity. The tree is then distributed to the routing entities at the start of simulation and subsequently maintained by the routing entities. Some branches of the tree may be *pruned* or *grafted* as the multicast membership changes.

During simulation, when a node wants to forward a packet downstream towards the packet's final destination, it sends a query message to the routing entity containing the destination and source addresses. The routing entity then looks up the routing table and returns a list of next hops the packet should be sent out to. For unicast destinations there is only one next hop while packets sent to multicast destinations may have to be sent out through multiple outgoing interfaces.

5 RSVP Layer

RSVP [31],[5] is the signaling protocol for the Integrated Services architecture [7]. RSVP is a *soft-state*, *receiver-oriented*, *two-phase* resource reservation protocol for simplex flows supporting one-to-one and multi-party communications. Senders advertise the characteristics of the traffic

[1]The code was originally written by Rajat Ahuja and Lokesh Bajaj of UCLA's Parallel Computing Lab and it has been modified to fit into IRL-Sim's framework

they generate (i.e. in terms of peak and average transmission rate[2]), by sending PATH messages to the potential receiver(s). Receivers interested in receiving higher QoS for the traffic advertised by these senders, respond by sending RESV messages requesting a specific level of service. RESV messages travel on the reverse path from receivers to senders reserving resources along the way. Routers on the path between senders and receivers must process these RSVP messages, create associated *state* for each one of these messages and allocate network resources according to the requests carried in the RESV messages.

Senders and receivers periodically transmit their PATH and RESV messages respectively. When routers receive these messages, they *refresh* the associated RSVP state they keep. If on the other hand, these routers do not receive regular refreshes they will *tear down* the associated RSVP state and network resources will be released. This approach (Clark in [6] used the term *soft state*) provides an elegant yet powerful way of handling network failures and stale state.

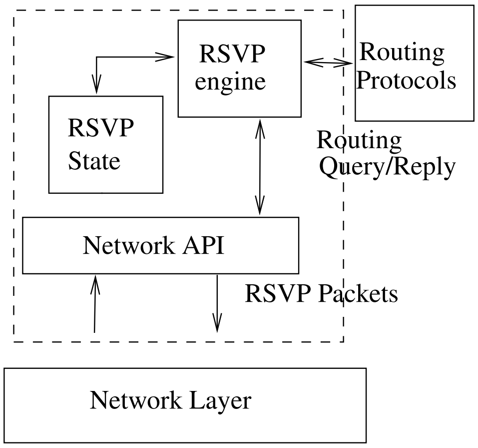

Figure 4. RSVP Daemon Architecture

The architecture of the RSVP daemon is depicted in Figure 4. A node's RSVP daemon receives through its Network API all the RSVP messages exchanged between senders and receivers that pass through that node. Messages are then processed by the RSVP protocol engine and RSVP state is installed or modified inside the RSVP daemon. After processing each RSVP message, the RSVP daemon forwards the message to the next router. To do so, it queries the routing protocol (unicast or multicast depending on the destination) about the (list of) interface(s) that the packet should be sent out from. Each RSVP entity records every message exchanged and the associated state changes in a log file for post-processing.

[2]traffic specifications are called *Tspecs* in RSVP jargon

5.1 RSVP Senders and Receivers

While routers process and forward RSVP messages, it is the traffic end-points that initiate RSVP sessions to reserve network resources for the traffic they send and receive. So, to have complete QoS functionality in our simulator, in addition to RSVP processing at network routers we have implemented the needed functionality at the network edges. Specifically we have integrated RSVP functionality at the traffic senders and receivers. While we describe these end-points in more detail in Section 6 we talk about RSVP senders and receivers in the paragraphs that follow.

The role of the RSVP sender is to send PATH messages thus initiating the whole resource reservation procedure. Once initialized, the sender creates an RSVP PATH message using the configured parameters and sends the message towards its destination. After that, it enters an endless loop, periodically refreshing the PATH message while waiting for RESV or ERROR messages.

While the senders advertise existing flows by sending PATH messages, it is the role of the receivers to initiate the reservations. In this respect, the receiver functionality is more complicated than that of the sender. A receiver, if it is part of a multicast group it has to join the group. After that, it waits for PATH messages from senders sending to the session (unicast or multicast) that it is listening to. Each new PATH message is processed and if it comes from a previously unknown sender, it is added to the list of senders sending to the receiver's session. From that list of senders, the receiver chooses a flow to reserve for. The receiver then uses the sender's parameters to send a RESV message towards that sender. Once this process is finished, the receiver periodically refreshes its reservation by sending refresh RESV messages

6 Transport Layer Protocols

As we have seen in Section 2, there are two major types of simulated network nodes: routers that forward packets and end nodes that act as the sources and sinks of network traffic. User applications run at the end hosts as well as the *transport* protocols responsible for tasks such as *demultiplexing*, *reliable delivery* and *congestion control*. In the paragraphs that follow we will present each of the two transport protocols we have implemented and the applications that run on top of them.

6.1 UDP

The first class of applications is what we call *UDP sources/receivers* and contains applications that do not perform any reliability or congestion control mechanisms. We have named them this way after the UDP [21] transport protocol in the TCP/IP protocol stack, since UDP does not perform any of these functions either. We haven't actually implemented UDP in the sense of a different entity that "sits" between the applications and the network layer. What we have done instead, is to fold UDP inside the applications.

There are three types of UDP sources in IRLSim. The difference among them is in the pattern of the traffic they send. The simplest of the three is a *constant* rate source that sends specified size packets at fixed intervals. The next type of source, is what is called an ON/OFF source, meaning that the source alternates between the ON state in which it sends packets at a fixed interval and the OFF state where it does not send any traffic. We have implemented two variations of the ON/OFF source, the difference being the *distribution* of the length of the ON and OFF intervals. In the first type of source, the intervals follow an *exponential* distribution which results in a *Poisson* source. Poisson sources have been widely used in network research as traffic sources, but recent evidence [28] shows that traffic on the Internet shows *self-similar* characteristics which cannot be generated by Poisson sources. The authors of [28] have shown that the superposition of a large number of ON-OFF sources that have a *long-tail* distribution (i.e Pareto) can create self-similar network traffic. For these reasons we have also implemented a *Pareto* ON/OFF source where the ON and OFF intervals follow the Pareto distribution.

As we will see in Section 8.1 we have used UDP sources and receivers to test the effects of mobility on *playback* applications. The term playback applications appeared in [7] to model real-time applications. In these applications the source takes some signal (e.g. voice) packetizes it and then transmits it over the network. The network inevitably introduces some variation in the delay of each delivered packet. This variation has traditionally been called *jitter*. The receiver depacketizes the data and attempts to faithfully play back the signal. This is done by buffering data to remove the network induced jitter and then replaying the signal at some designated *play-back* point. Any data that arrives before its associated playback point can be used to reconstruct the signal; data arriving after the play-back point is useless in reconstructing the real-time signal. The implemented UDP receivers support two *adaptation protocols* for setting the play-back point. These adaptation protocols were originally presented in [23].

6.2 TCP

We first describe the design of the TCP module in Section 6.2.1 and then address some issues in porting real code to IRLSim using TCP as an example. In Section 6.2.3, we describe the TCP applications implemented in our simulator.

6.2.1 TCP Module Design

TCP [22] is a transport protocol that provides in-order reliable delivery. To ensure reliability, TCP assigns each byte of data a sequence number and a receiver sends an acknowledgment carrying the sequence number after it receives a packet from the sender. The sender then decides on which piece of data to (re)transmit based on the information in the acknowledgment. What's most interesting in TCP is the congestion control algorithms that open/close TCP's sending window dynamically in response to changes in network condition. The intricate details of these congestion control algorithms have been studied for decades and many improvements have been applied to the original algorithms [1]. In the rest of this section, we describe the design objectives of the TCP module and how we achieved them.

There are at least three possible uses of the TCP module:

1. Experimentation with various features of TCP;
2. Evaluating TCP performance;
3. Implementation of application prototypes on top of TCP.

To support these uses, we have chosen to port the TCP code in 4.4BSD-Lite [17] to IRLSim. The ported code supports many algorithms of TCP such as slow start, congestion control and avoidance, fast retransmission and fast recovery, as well as various TCP header options. In addition, users can (a) enable or disable some optional features including keep-alive, delayed ACK and Nagle algorithm; (b) turn on or off any TCP header options; and (c) specify various protocol parameters, such as maximum sender segment size, sender and receiver's buffer size.

To collect simulation statistics, we embedded logging facilities inside the code. By analyzing the log file using the scripts we provide, one can obtain TCP window plot (window size variation with time) and sequence number plot (sequence number growth with time). From the window plot, one can tell the various stages a TCP connection is in (e.g. slow start or congestion avoidance). TCP sequence number plot shows how fast a TCP connection transmits data and when it retransmits a packet. We also print summary statistics for TCP connections, such as throughput, starting and ending time, at the end of a simulation run.

To support application prototyping, we designed a simple interface between applications and TCP. This interface is a simplified version of the Unix socket API, which enables users to port simulation applications to real Unix system and vice versa. The interface API is comprised of five primitives: *listen*, *open*, *close*, *send* and *receive*. An application can initiate TCP connections using the first three primitives and transfer data using the last two primitives.

6.2.2 Porting Real Code into IRLSim

In general, one needs to address the following issues in order to port real networking code into IRLSim:

1. **Protocol Interaction**

   If the real code uses function calls to invoke the services of another protocol/module, one needs to map the function calls to the corresponding PARSEC messages we have defined. This is because we model each protocol as a PARSEC entity and entities communicate via PARSEC messages. For example, when the network layer entity wants to submit a packet to TCP, it sends a PARSEC message *packet_nw_to_tcp* with the packet as a parameter, as opposed to calling *tcp_input()* directly. Once the TCP entity receives the Parsec message, it calls *tcp_input()* to process the packet.

2. **Timer Management**

   Networking code uses timers extensively and different systems implement timers in different ways. In the BSD code we use, there is a system interrupt every 500 ms and each 500 ms is called a tick. When TCP sets a timer, it actually sets a timer variable to the number of ticks until the timer expires. Whenever it receives a system interrupt, it checks all the timer variables to see if any of them has expired. In the simulation, we use a Parsec message which is periodically sent to the TCP entity to simulate the interrupt. In this way, we minimize the changes to the timer management code. Since other systems may implement timers in a different way, one needs to find a way to simulate timers in Parsec or make use of our periodic timer messages.

3. **Simulation Detail**

   To simplify code and speed up simulation, one may want to eliminate unnecessary features present in the real code. For example, if the simulated network environment is homogeneous, one can take out the code that handles backward compatibility and heterogeneity.

6.2.3 TCP Applications

FTP and Telnet are two representative TCP applications. We simulate FTP traffic by sending data continuously from a TCP sender to a TCP receiver. Telnet traffic is modeled as one-byte data sent at randomly distributed intervals (we currently use exponential distribution). Users can specify starting time, data transfer duration and data size for every TCP sender. As a next step, we plan to integrate tcplib ([14]) into IRLSim to model more applications.

7 Graphical User Interface

A valuable addition to any simulator package is a graphical user interface (GUI). Therefore, we decided to provide one for IRLSim and it turned out to be very useful by simplifying many tasks. The GUI, which was implemented in Java with ideas and the reuse of some classes of the similar interface of [18], allows users to view the protocol actions and interact with various network elements during simulation. A snapshot of how the interface looks like with a simple topology is presented in Fig. 5.

One of the first goals in the development of IRLSim was to have a full RSVP simulation that would be used for various RSVP related experiments. To test and evaluate the behavior of RSVP during such experiments it would be useful to visually observe the exchange of RSVP messages. This was made possible with the GUI and it proved a valuable feature for debugging and testing. The small disks along each edge in Fig. 5 were used for that purpose. When no message is transmitted over the link the disks remain colored grey. When a message is transmitted or received, one of them is briefly illuminated with a color appropriate for the type of message being send or received. However, as the topologies and the scenarios become more complex, this feature becomes less valuable. The multitude of messages that are exchanged can get very large and it becomes hard for the user to visually track them. This feature was not extended for the other protocols that were added in the simulator (e.g. for TCP and UDP) because of the ever increasing number of messages that would have to be animated. Additionally, an option is given to the user to disable this feature.

Nevertheless, for several of the experiments that were conducted with the simulator, it is desirable to have detailed view of what is going on down to the packet level. For example, in the experiments involving the study of mobile RSVP we want to study and observe closely the data flow. Due to the shortcomings of using the GUI for such tasks, we decided to integrate into IRLSim support for the Network Animator - *nam* ([10]). nam allows for the post-mortem animation of a networking scenario at varying playback speeds. This is very appealing since it allows to observe events that might be happening too fast to be observed during the simulation. So, during the course of the simulation, networking events are logged according to the nam log format for later playback.

As a balance between waiting to view the logs after the experiment has concluded and viewing all the networking events as they happen, various logs are kept that can being accessed at the end of a simulation step. This featured has been integrated in the GUI and the logged information that has been recorded on the disk is displayed by clicking on the desired node.

Reviewing fig. 5 we can distinguish several components

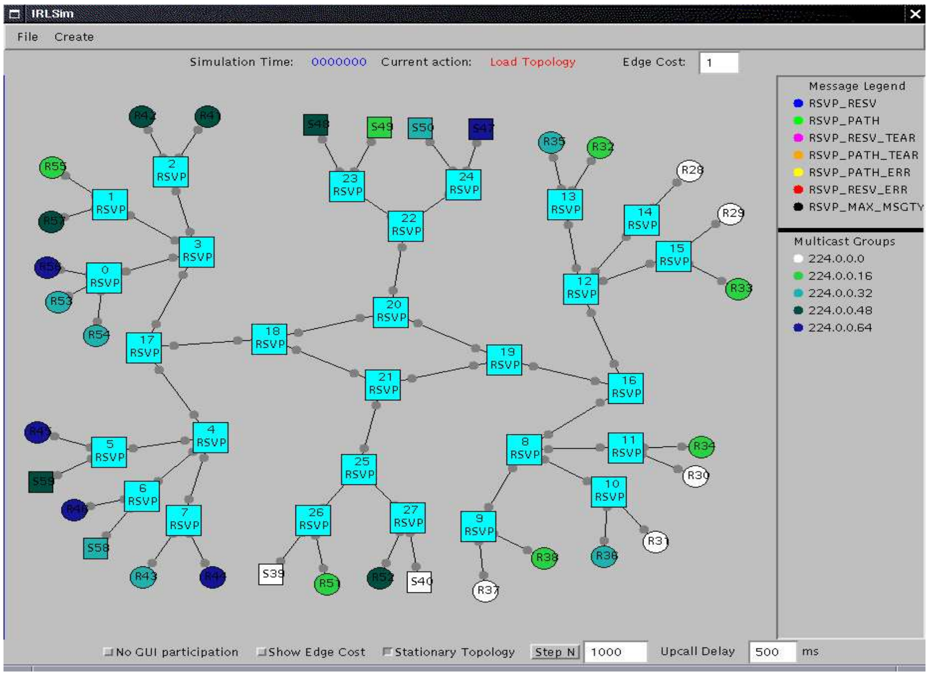

Figure 5. A screen snapshot

of the GUI. The most prominent part is the central component which displays the topology and where all the action takes place. There are three different kinds of nodes: senders identified by a square shape, properly labeled with the sender's id; RSVP capable routers also with square shapes but distinctly colored and labeled; and receivers drawn as circles, also appropriately labeled. The coloring of routers is insignificant, but the coloring of senders and receivers reveals information about their multicast group membership. Receivers with the same color means that they belong to the same multicast group and senders of the same color means that the corresponding sender has data flow for that group.

This brings us to the right portion of the interface which has not any functionality but it is for informational purposes: it displays, on the top part, the types of RSVP messages that can be generated in the simulation and their associated colors. On the lowest part it displays the multicast groups (in terms of their IP addresses) and their associated color.

The top level of the interface consists of two menus. The *File* menu currently supports saving and loading topologies. Topology files are saved in a human readable format that allows for manual adjustments after the topology has been designed. For the creation of topologies, the *Create* menu is used. It allows the user to select the type of node to be created (router, TCP sender, UDP sender, to name a few) or add an edge or create a multicast group. The choice of an option from the *Create* menu simply enables the creation of the respective component. A subsequent click (or drag if it is an edge) of the left mouse button is required for that. The creation of most network elements involves additional interaction. For example, the destination for a sender's traffic needs to be specified, as well as the characteristics of the traffic transmitted by the sender. Fig. 6 displays a sample interaction window during the creation of a UDP source. Similar windows pop up for the creation of the other network elements.

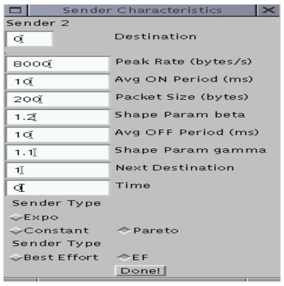

Figure 6. Pop-up window for getting the characteristics of a sender

On the top of the simulation panel, the simulation time is displayed as provided by the Parsec clock. There are a few more inputs that the user can provide. In one of those the user can specify the duration that the simulation is to be executed. This allows the stepwise execution of the simulation so that the user gets the chance to monitor the state of various nodes. Other useful features include zooming in and out, printing of topology graphs and viewing of various link characteristics.

8 Case Studies

So far we have used our simulator for two main reasons: As a learning tool for ourselves on how networks are built and work but also as a tool for our research work. We have used IRLSim in a number of projects in our research group. In the paragraphs that follow we describe two of the projects we have used IRLSim for.

8.1 RSVP Extensions for Mobile Users

IP-in-IP "tunnels" have become a widespread mechanism to transport datagrams in the Internet. Typically, a tunnel is used to route packets through portions of the network which do not directly implement the desired service (e.g. IPv6), or to augment and modify the behavior of the deployed routing architecture (e.g. multicast routing, mobile IP, Virtual Private Net). From the perspective of traditional best-effort IP packet delivery, a tunnel behaves as any other link. Packets enter one end of the tunnel, and are delivered to the other end unless resource overload or error causes them to be lost. IP-in-IP tunnels cause problems in the regular processing of RSVP messages, since RSVP messages get "lost" when they cross a tunnel [3]. The main idea proposed in [27] to solve this problem, is to have a separate RSVP session between the tunnel endpoints. The tunnel entry point R_{entry} serves as the sender for the Tunnel session, while the tunnel exit point R_{exit} serves as the receiver. The tunnel session can exist independently from the End-to-End sessions (e.g created via a management interface), or its creation can be triggered by End-to-End messages. When an End-to-End RSVP session crosses an RSVP capable tunnel it is mapped to a tunnel RSVP session. The tunnel RSVP session views the two tunnel end-points as the two end hosts. Then a reservation is made from R_{exit} to R_{entry} for the amount of data crossing the tunnel. The original, end-to-end RSVP session views the tunnel as a single (logical) link along the path between the source(s) and the destination(s). PATH and RESV message of the End-to-End session are encapsulated at one tunnel end-point and get decapsulated at the other end, where they get forwarded as usual. Data packets are encapsulated with the IP as well as a UDP header when crossing the tunnel. In this way, packets belonging to different flows can be distinguished by routers inside the tunnel using standard RSVP processing.

In [25], we have used the *RSVP Tunnels* mechanism described above to provide RSVP functionality to mobile users. In the current Internet mobility model [20], mobile hosts are associated with a *Home Agent* (HA) that is responsible to deliver the traffic to the mobile node when this node is outside its *home domain*. In order to support forwarding from a mobile node's HA to the *Foreign Agent* (FA) we implemented a mechanism equivalent to Mobile-IP [20]. When the mobile node moves to a different *cell* it registers with the router of that cell. This message is delivered to the node's routing entity which alters the routing entry for that node and forwards the requests to the mobile node's home agent. Once the home agent receives this message it adds a *tunnel* routing entry for packets address to the mobile node. When subsequent packets packets addressed to the mobile host arrive at the home agent, the home agent encapsulates them and sends them to the current foreign agent. Once delivered to the foreign agent, packets are decapsulated and are then directly delivered to the mobile host.

From this description it is easy to see that to support RSVP reservations from and to mobile nodes, one needs to create an RSVP Tunnel between the HA and the FA.

We have investigated two cases:

1. Tunnels between base stations are created dynamically when mobile nodes cross between domains.

2. Tunnels between neighboring nodes are preconfigured and when mobile nodes move to a new domain their end-to-end reservations are mapped onto the preexisting allocations between the tunnel endpoints.

Using simulations, we have shown that pre-existing tunnels result in dramatically fewer dropped data packets and therefore improved application support.

8.2 Differentiated Services

Differentiated Services is a recent effort to provide scalable service differentiation in the global Internet. The Differentiated Services architecture [4], is based on a simple model where within the core of the network, routers decide how to service packets by looking at the DS field (previously called TOS byte) in the IP header. Each DS field value corresponds to a different treatment, called a Per Hop Behavior (PHB). For example, if the value a packet carries translates to a "low delay" treatment, routers could put that packet in a priority queue to service it promptly. Since core routers only have to look at DS codepoint to decide how to service a packet, no intricate classification or per-flow state is needed, leading to increased scalability and flexibility. To ensure that network resources are not over-allocated, traffic entering the network is classified and possibly shaped or policed at the boundaries of the network. The reason for policing resource usage at points close to the traffic sources is that load at those points

[3] RSVP packets use the Router Alert option to indicate to routers on the path that they require special handling. When RSVP messages are encapsulated with an outer IP header, the Router Alert option becomes invisible

is light allowing for more complex operations. The synthesis of per hop behaviors and traffic conditioning creates the end-to-end services visible to network users.

So far most of the work in Differentiated Services has focused on defining Per Hop Behaviors and ways to provide them, while management plane issues have received little attention. The management plane is concerned with the configuration of network elements and the allocation of network resources to network users. In [24] we proposed an architecture that attempts to deal with these tasks. The tenet of our architecture is what we call *Two-Tier resource management*. By this term we mean that resource allocation should be done in two levels. The first level is resource allocation inside each administrative domain while the second level is resource management across neighboring domains. The role of the inter-domain protocol is to communicate resource agreements between the neighboring domains and to set the appropriate parameters at the edges devices. The task of the second component of the Two-Tier architecture, the intra-domain resource allocation mechanism, is the allocation of resources inside DS domains. Different DS domains are free to choose a mechanism that best fits their needs. A couple examples of such mechanisms are over-provisioning and static allocation of resources. In [26] we present another proposal for this task that uses RSVP to allocate resources for aggregate flows between a domain's ingress and egress routers. The co-ordination of intra- and inter-domain resource allocations creates the end-to-end services observable to end users.

In [26] we used IRLSim to validate our model with extensive simulations. Using these simulations we have been able to investigate the engineering trade-offs on the design of the inter-domain protocol and we were able to fine tune some of the protocol's parameters. Our initial results using the simulator, have shown that the Two-Tier resource allocation scheme provides end-to-end services comparable to those provided by other architectures at a much lower network overhead.

9 Related Work

Network simulation has been the focus of considerable research and numerous simulation packages have emerged from these research efforts. Here we list some of them and compare the most prominent among them with IRLSim. There is a wide span of focus in different simulators. Some model a narrow aspect of a network with varying degrees of detail that is only relevant to the problem at hand. Many others focus on the study of specific protocols (e.g a multicast protocol) or specific networking environments. GlomoSim [30] is such a simulator and focuses on the simulation of mobile wireless networks. It shares the same implementation infrastructure as IRLSim since it is implemented in PARSEC as well. At the other end, there are more general simulation packages that target a wide range of protocols and environments. Such include ns (and its predecessor REAL [15]), OPNET [9], and others.

Those high end simulation packages are usually distinguished by the level of abstraction they offer and the way they address the issue of scalability. Most come with their own simulation language and a set of assorted protocol libraries. Most high end simulators provide different kinds of simulation interfaces, like programming using a high level scripting language (like Tcl in the case of ns), or a traditional programming language or a combination of both.

Comparing IRLSim with other simulator packages, particularly ns, we can say the following. IRLSim started at one end of the simulator spectrum, that is, it focused on simulating only a specific protocol (RSVP), and gradually it evolved to a multi-featured one. Many of the scenarios that can be studied with IRLSim can also be studied with ns. Nevertheless, we believe that IRLSim has some advantages over ns. Despite its success, ns is difficult to use since it has a rather steep learning curve for people not familiar with the scripting environment. IRLSim on the other hand, allows for the easy set up of simulations scenarios and the easy monitoring of the progression of experiments. ns can be used only in conjunction with nam, making it impossible to interact with the simulation as it progresses. IRLSim is build on top of PARSEC, a programming language targeted towards simulation. The degree of scalability of IRLSim is depended on the effectiveness of PARSEC which according to [2] is very promising. It is true however that the communication required by some protocols does not always allow for that potential to be fully exploited. The ns simulation engine is written in a system language and making extensions is harder and parallelization is not possible.

10 Summary and Future Work

In summary, IRLSim is a scalable and easy to use network simulator. To achieve scalability, we use PARSEC with entity aggregation where one entity is used to simulate multiple nodes. In this way, we can significantly reduce the memory usage and execution time of simulations. For ease of use, we provide a GUI with which users can create new topologies, load and edit existing topologies, as well as observe the progress of a simulation in real-time. IRLSim boasts a complex set of features, including QoS capabilities, such as resource reservation, traffic shaping, traffic policing, and EF forwarding. Extensibility and portability is achieved by having a simulator architecture which is very close to networking stacks in real-life systems.

In the future, we plan to improve our simulator along several directions. First, we want to experiment with parallel simulations. Up to now, all of our simulations run on sequential machines and therefore we have not tapped into PARSEC's biggest potential benefit. Since the speed up of parallel programs over sequential programs depends heavily on reducing the interaction between entities running on different processors, we need to refine the algorithm that distributes network nodes to entities in a way that minimizes the communication between entities. Second, we plan to support dynamic events such as node and link failures as well as dynamic routing protocols that recompute routing tables after these failures happen. At the networking layer we seek to implement more scheduling and queue management mechanisms such as Fair Queuing and CBQ [11], while at the application layer we aim to add more TCP applications. Our first priority is to model protocol behavior similar to HTTP which is the dominant traffic on the Internet today.

References

[1] M. Allman, V. Paxson, and W. R. Stevens. TCP Congestion Control. *RFC 2581*, 1999.

[2] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *IEEE Computer*, 31(10):77–85, Oct 1998.

[3] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejae, P. Sharma, S. Shenker, K. Varadhan, H. Yu, Y. Xu, and D. Zappala. Improving Simulation for Network Research. Technical Report 99-702, University of Souther California, Mar 1999.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. *RFC 2475*, Dec 1198.

[5] R. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP), Version 1 Functional Specification. *RFC 2205*, Sept. 1997.

[6] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proceedings of SIGCOMM'88*, 1988.

[7] D. D. Clark, S. Shenker, and L. Zhang. Supporting Real Time Applications in an Integrated Services Packet Network: Architecture and Mechanisms. In *Proceedings of SIGCOMM '92*, pages 14–26, Aug. 1992.

[8] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, May 1990.

[9] F. Desbradndes, S. Bertolotti, and L. Dunant. OPNET 2.4: An environment for communication network modeling and simulation. In *Proceedings of the European Simulation Symposium*, Oct 1993.

[10] D. Estrin, M. Handley, J. Heidemann, S. Mccanne, Y. Xu, and H. Yu. Network visualization with the VINT network animator nam. Technical Report 99-703, Universtity of Southern California, Mar 1999.

[11] S. Floyd and V. Jacobson. Link Sharing and Resource Management Models for Packet Networks. *IEEE/ACM Transactions on Networking*, 3(4):365–386, Aug 1995.

[12] V. Jacobson and S. Floyd. Random Early Detection gateways for congestion avoidance. *Transactions on Networking*, 5(6), Dec 1997.

[13] V. Jacobson, K. Nichols, and K. Poduri. An Expedited Forwarding PHB. *RFC 2598*, Jun 1999.

[14] S. Jamin and P. Danzig. tcplib: A Library of TCP Internetwork Traffic Characteristics. *USC Technical Report USC-CS-91-495*, 1991.

[15] S. Keshav. REAL: a network simulator. Technical report, University of California, Berkeley, Dec 1988.

[16] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture . In *Proceedings of the 1993 Winter USENIX Technical Conference*, Jan 1993.

[17] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, Reading, Massachusetts, 1996.

[18] K. Nguyen and N. Sturtevant. A Simulation and Analysis of the Cache Group Management Protocol (CGMP). Available at http://irl.cs.ucla.edu/ khoi/cgmp, 1997.

[19] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. *RFC 2474*, Dec 1998.

[20] C. Perkins. IP Mobility support. *RFC 2002*, Oct. 1996.

[21] J. Postel. User Datagram Protocol. *RFC 768*, August 1980.

[22] J. Postel. DoD Standard Transmission Control Protocol. *RFC 793*, 1981.

[23] R. Ramjee, . Kurose, D. Towsley, and H. Schulzzrinne. Adaptive playout mechanisms for Packetized Audio Applications in Wide-Area Networks. In *Proceedings of IEEE Infocom '94*, 1994.

[24] F. Reichmeyer, L. Ong, A. Terzis, L. Zhang, and R. Yavatkar. A Two-Tier Resource Management Model for Differentiated Service Networks. *Internet-Draft, work in progress*, Nov 1998.

[25] A. Terzis, M. Srivastava, and L. Zhang. A Simple QoS Signaling Protocol for Mobile Hosts in the Integrated Services Internet. In *Proceedings of INFOCOM'99*, Apr. 1999.

[26] A. Terzis, L. Wang, J. Ogawa, and L. Zhang. A Two-Tier Resource Management model for the Internet. In *Proceedings of GLOBECOM'99*, dec 1999.

[27] A. Terzis, J. Wroclawski, J. Krawczyk, and L. Zhang. RSVP Operation Over IP Tunnels. *Internet-Draft, work in progress*, Nov. 1999.

[28] W. Willinger, M. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high variability: statistical analysis of Ethernet LAN traffic at the source level. In *Proceeding of the ACM SIGCOMM'95*, Sep 1995.

[29] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2 The Implementation*. Addison-Wesley, 1995.

[30] X. Zeng, R. Bagrodia, and M. Gerla. "GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulations*, May 1998.

[31] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5), Sept. 1993.