RINSE: the Real-time Immersive Network Simulation Environment for Network Security Exercises

Michael Liljenstam[1], Jason Liu[2], David Nicol[1], Yougu Yuan[1], Guanhua Yan[1], Chris Grier[1]

[1]University of Illinois at Urbana-Champaign
Coordinated Science Laborabory
1308 W. Main St., Urbana, IL 61801
{mili,nicol,yuanyg,ghyan,grier}@crhc.uiuc.edu

[2]Colorado School of Mines
Mathematical and Computer Sciences
Golden, CO 80401-1887
xliu@mines.edu

Abstract

The RINSE simulator is being developed to support large-scale network security preparedness and training exercises, involving hundreds of players and a modeled network composed of hundreds of LANs. The simulator must be able to present a realistic rendering of network behavior as attacks are launched and players diagnose events and try counter measures to keep network services operating. We describe the architecture and function of RINSE and outline how techniques like multiresolution traffic modeling and new routing simulation methods are used to address the scalability challenges of this application. We also describe in more detail new work on CPU/memory models necessary for the exercise scenarios and a latency absorption technique that will help when extending the range of client tools usable by the players.

1. Introduction

The climate on the Internet is growing increasingly hostile while organizations are increasingly relying on the Internet for at least some aspects of day-to-day operations. They are thus being forced to plan and prepare for network failures or outright attacks—how it might affect them and what actions to take. With current system complexity, tools to assist in preparedness evaluation and training are likely to become more and more important.

The October 2003 Livewire cyber war exercise [1] conducted by the Department of Homeland Security, is one particular instance of preparedness evaluation and training that involved companies across industrial sectors as well as government agencies. More exercises of this type are currently being planned, and based on experiences from the first event, there was a desire for improved tools to automatically determine the impact on the network from attacks and defensive actions and the extent to which the network is capable of delivering the services needed. Providing network simulation tool support for exercises such as Livewire is particularly challenging because of their scale. Future exercises are expected to involve as many as a couple of hundred participating organizations, and will thus involve many "players" and a network of significant size.

We are currently developing the Real-time Immersive Network Simulation Environment for network security exercises (RINSE) to meet this need and address the challenges inherent in this type of application. Hence, the goal for RINSE is to manage large-scale real-time human/machine-in-the-loop network simulation with a focus on security for exercises and training. It needs to be extensible so that it can evolve over time, and it needs to be designed with an eye towards security and resilience to hardware faults since these exercises involve many people and last for several days.

The spectrum of approaches to general large-scale network modeling being explored in the literature range from hardware emulation testbeds like Emulab [30], network emulators like ModelNet [29], to network simulators like IP-TNE [27], GTNetS [6], PDNS [6], and MAYA [31]. Hardware emulation excels in application code realism (running the real thing), while simulations tend to be more flexible and have an advantage in terms of scalability. However, the middle ground is increasingly being explored; for instance, through increasing emulation support in simulators. For security exercises we like the flexiblity and scalability of simulation, and the safety of unleashing attacks in a simulated environment rather than on a real network.

Several simulators offer similar capabilities, including parallel execution, real-time/emulation support, and discrete event/analytic models. However, we believe RINSE is unique in the way it brings together human/machine-in-the-loop real-time simulation support with multiresolution network traffic models, attack models that leverage the efficiency of the multiresolution traffic representations, novel

Proceedings of the Workshop on Principles of Advanced and Distributed Simulation (PADS'05)
1087-4097/05 $20.00 © 2005 IEEE

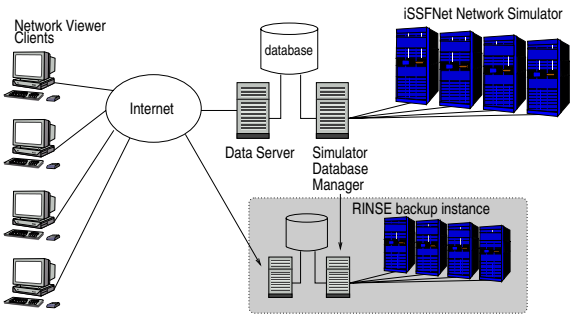

Figure 1: RINSE architecture

models of host/router resources such as CPU and memory, and novel routing simulation techniques. In this position paper we provide an overview of RINSE to show how these techniques are being brought to bear on the problem at hand. We also detail some specific new contributions: *i)* a technique for absorbing outside network latency into the simulation model and *ii)* models for including CPU and memory effects into network simulations.

The remainder of this paper is organized as follows. Section 2 describes the architecture of RINSE and outlines a simple example scenario that introduces the salient features of RINSE, described further in Sections 3 to 5. Finally, Section 6 summarizes and outlines future work.

2. RINSE Architecture

RINSE consists of five components: the iSSFNet network simulator, the Simulator Database Manager, a database, the Data Server, and client-side Network Viewers, as shown in Figure 1. The iSSFNet network simulator, formerly called DaSSFNet, is the latest incarnation of the C++ network simulator based on the Scalable Simulation Framework (SSF), an Application Programming Interface (API) for parallel simulations of large-scale networks [3]. iSSFNet runs on top of the iSSF simulation kernel, which handles synchronization and support functions. iSSF uses a composite synchronous/asynchronous conservative synchronization mechanism for parallel and distributed execution support [18], and has recently been augmented to include real-time interaction and network emulation support. iSSFNet runs on parallel machines to support real-time simulation of large-scale networks.

Each simulation node connects independently to the Simulator Database Manager, which delivers data from the simulator to the database and delivers control input from the database to the simulator. On the user/player side, the Data Server interfaces with client applications, such as the Java-based application "Network Viewer", which allows the user to monitor and control the simulated network. In the future, we plan to evolve the architecture towards using the emulation capabilities to support direct SNMP interaction with the simulated network devices, thus having regular networking utilities and network management tools as clients. In the current design, the Data Server performs authentication for each user, distributes definitions of the client's view of the network (using the Domain Modeling Language), and provides a simple way for the client applications to access new data in the database through XML-based remote procedure calls. The Network Viewer clients, a screen shot shown in Figure 2, provide the users with their local view of the network (usually only their organization's network) and periodically poll the Data Server for data. The Data Server responds with new data for each client, extracted from the database. The game managers, functioning as superusers of an entire exercise, also use Network Viewer clients, but can have a more global view of the network.

The Network Viewer clients have a simple command prompt where the user can issue commands to influence the model. User commands are sent in the opposite direction of the output data path and injected into the simulator. We currently divide the commands into five categories:

Attack– the game managers can launch attacks against networks or specific servers. RINSE focuses on Denial-of-Service effects on networks and devices, so attacks include DDoS and worms.

Defense– attacks can be blocked or mitigated, for instance by installing packet filters.

Diagnostic Networking Tools– functionality similar to some commonly used networking utilities, such as ping, are supported for the player to diagnose the network.

Device Control– individual devices, such as hosts and routers, can be shutdown or rebooted.

Simulator Data– commands can be issued to the simulator to control the output, turn on or off trace flow from a particular host, etc.

Depending on the type of a command, it may be addressing the whole simulator, a particular host or router, a particular interface, or a particular protocol or application on a host or router. A command handling infrastructure in the simulator passes the commands from the clients to the appropriate components of the simulation model.

2.1. Example Scenario

Consider a simple scenario where a player is responsible for a subnetwork, partially shown in Figure 2, containing among other things a server. Multiple clients are requesting information from the server through some form of transactions. By transaction we simply mean a request-response

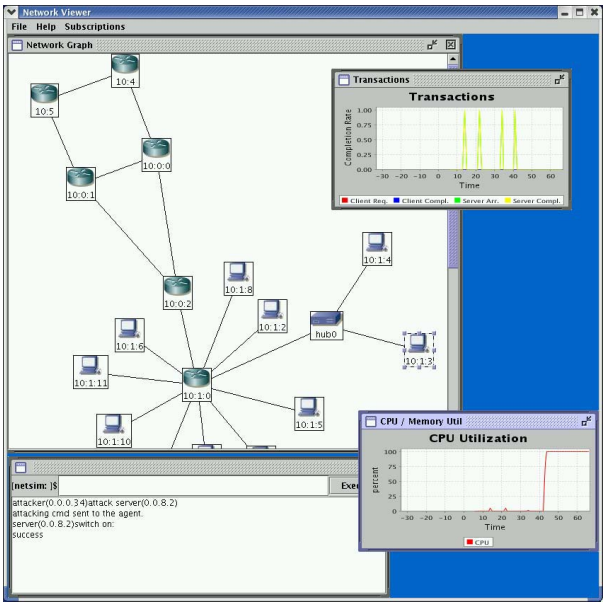

Figure 2: Network Viewer client screen shot

exchange between the client and the server. Section 4 outlines RINSE's models for efficient representation of traffic flows and route computation.

A game manager attempts to disrupt the operations of the server by launching a DDoS attack against open services on the server, and the player responsible for the network will need to diagnose what is going on and try to take remedial actions. The game manager launches the attack by issuing an attack command at the Network Viewer client:

```
ddos_attack attacker server 100 2000
```

Both `attacker` and `server` are symbolic names for the attacker's host and the targeted host, respectively. Upon receiving the command (via the command handling infrastructure), the attacker's host uses a simulated intermediate communication channel (e.g., Internet Relay Chat) to send attack signals to zombie hosts—hosts under the attacker's control. These zombie hosts then initiate the denial-of-service attack against the targeted victim. The attack is to last for 100 seconds and each zombie emits traffic at a rate of 2000 kbits/s. RINSE attack models leverage efficiencies in its high volume traffic representations (Section 4).

We will assume here that the DDoS traffic simply loads the open service daemons and thus induces a large CPU load on the server. This load disrupts the processing of legitimate transactions. As shown in the screen shot in Figure 2, the player managing the server can monitor the CPU utilization on the server and observe an abnormally high load. Models of host and router resources like CPU and memory are described in more detail in Section 5. After determining that the load likely stems from abnormal traffic, the player attempts to block traffic on a certain port that has been inadvertently left open by issuing the command:

```
filter server add 0 deny all all * all * 23
```

to install a filter on the server to deny packets coming in on `all` interfaces, using `all` protocols, from all source IP addresses ("`*`"), and `all` source ports, to all destination IP addresses ("`*`") and destination port 23. Successful filtering blocks packets from reaching the open service daemons and thus alleviates the load on the server at the expense of some processing cost for filtering.

We now proceed to describe aspects of the system mentioned here in more detail, starting with the real-time simulation support.

3. Real-time Simulation Support

In addition to supporting the RINSE Network Viewer client, we are currently developing support for the Simple Network Management Protocol (SNMP) to allow us to monitor and control the simulated network devices through industry-standard network management tools. For that reason, our real-time simulation support must be simple, flexible, and be able to accommodate real-time interactions at varying degrees of intensity, including both human-in-the-loop and machine-in-the-loop simulations. In this section, we describe the real-time support both in the iSSF parallel simulation kernel and in the network simulator supported by iSSF.

3.1. Kernel Support

Over the years, we have seen many network emulators, ranging from single-link traffic modulators to full-scale network emulation tools, e.g. [25, 29]. Most network emulators are time-driven. For example, ModelNet [29] stores packets in "pipes" sorted by the earliest deadline. A scheduler executes periodically (once every 100 μseconds) to emulate packet moving through the pipes. There are two main drawbacks associated with the time-driven approach: *i*) the accuracy of the emulation depends on the time granularity of the scheduler, which largely depends on the target machine or the target operating system, and *ii*) there has not been a good model used by network emulators to accurately characterize the background traffic and its impact on the foreground transactions (i.e., traffic connecting real-time applications). Simulation-based emulation (also referred to as *real-time network simulation*), on the other hand, provides a common framework for real application traffic to interact with simulated traffic, and therefore allows us to study both network and application behaviors with more realism. Examples of existing real-time network simulators include NSE [5], IP-TNE [27], MaSSF [16], and Maya [31]. IP-TNE is the first simulator we know that adopts parallel sim-

ulation techniques for emulating large-scale networks. The real-time support in iSSF inherits many features of these previous simulation-based emulators. Our approach, however, is unique in several ways, which we elaborate next.

Extending SSF API. The real-time support is designed as an extension to the SSF API, thus making an easy transition for other SSF models that require real-time support. In SSF, an `inChannel` (or `outChannel`) object is defined as a communication port in an entity to receive (send) messages from (to) other entities. We extended the concept of the in-channel using it as the conduit for the simulator to receive events from outside the simulator (e.g., accepting user commands arrived at a TCP socket). We extended the API so that a newly created in-channel object can be associated with a reader thread. The reader thread converts (external) physical events into (internal) virtual events and injects them into the simulator using the `putVirtualEvent` method. A virtual event is created to represent the corresponding physical event and is assigned with a simulation timestamp calculated as a function of *i*) the wall-clock time at which the event is inserted into the simulator's event-list, and *ii*) the current emulation throttling speed (which we will elaborate momentarily). The SSF entities receive events from the in-channel objects as before, regardless of whether they represent special devices that accept external events. From a modeling perspective, there is no distinction between processing a simulation event and a real-time event. Similarly, we also extended the concept of the out-channel using it as a device to export events (for example, reporting the network state to a client application over a TCP connection). In this case, a writer thread can be associated with the special `outChannel` object. The writer thread invokes the `getRealEvent` method to retrieve events designated for the external device and converts the virtual events into physical events. Each of these events is assigned with a real-time deadline indicating the wall-clock time at which the event is supposed to happen. The real-time deadline is calculated from the virtual time and again the emulation throttling speed. The writer thread is responsible for delivering the event upon the deadline.

Throttling Emulation Speed. The system can dynamically throttle the emulation speed (either by accelerating or decelerating the simulation execution with respect to real time). This feature is important for supporting fault tolerance. For example, if a simulator fails over a hardware problem, after fixing the problem, the simulator should be able to restart from a previously checkpointed state and quickly catch up with rest of the system. We can accelerate the emulation speed and use the same user input logged at the database server to restore the state. In order to regulate the time advancement, we modified the `startAll` method in the `Entity` class (which is used to start the simulation in SSF), adding an optional argument to allow the user to specify the emulation speed as the ratio between virtual time and wall-clock time—for example, a ratio of one means simulation in real-time, "infinity" means simulation as fast as possible, and zero means halting the simulation. An `Entity` class method `throttle` is also added to make it possible to dynamically change the ratio during the simulation.

Prioritizing Emulation Events: We use a priority-based scheduling algorithm in the parallel simulation kernel to better service events with real-time constraints. In SSF, the user can cluster entities together as timelines, i.e., logical processes, that maintain their own event-lists. Events on the timelines are scheduled according to a conservative synchronization protocol [18]. In a "pure" simulation scenario, where the simulation is set to run as fast as possible, a timeline can be scheduled to run as long as it has events ready to be processed safely without causing causality problems. For that reason, during the event processing session, the kernel executes all safe events of a timeline *uninterrupted* to reduce the context switching cost. When we enable emulation, however, the timelines that contain real-time events must be scheduled promptly.

To promptly process the events with real-time deadlines in the system, we adopted a greedy algorithm in iSSF assigning a high priority to emulation timelines. These timelines contain real-time objects—special in-channels and out-channels that are used for connecting to the physical world. Whenever a real-time event is posted and ready to be scheduled for execution on these emulation timelines, the system interrupts the normal event processing session of a non-emulation timeline and makes a quick context switch to load and process the real-time events in the emulation timeline. This priority-based scheduling policy allows the events that carry real-time deadlines to be processed ahead of regular simulation events. Note that, however, since normal simulation events may be on a critical path that affects a real-time event, this method is not an optimal solution. We are currently investigating other more efficient scheduling algorithms that can promptly process emulation events as well as events on the critical path, so that the real-time requirement can be satisfied in a resource-constrained situation.

3.2. Latency Absorption

We realize that the real-time demand not only puts a tight constraint on how we process events to reduce the chance of missed deadlines, but also on the connectivity between the simulator and the real applications. For example, consider a scenario in which a path is established between a client machine running the `ping` application and the machine running the network simulator, as shown in Figure 3. The client machine, which assumes the role of a host in the simulated network (with a virtual IP address 10.5.0.12),

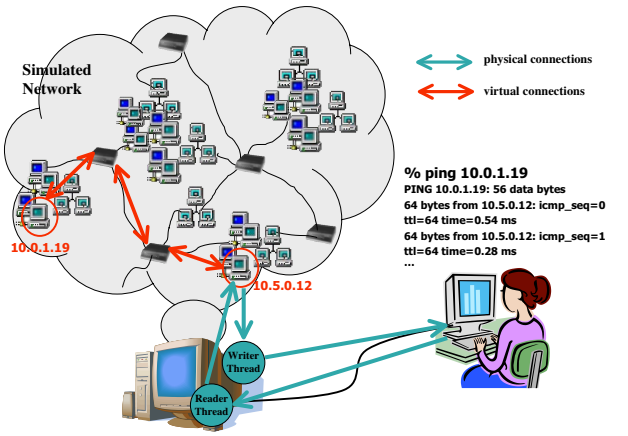

Figure 3: Emulation of a Ping Application

pings another host at 10.0.1.19. The ping application at the client machine generates a sequence of ICMP ECHO packets targeting 10.0.1.19. These packets are immediately captured by a kernel packet filtering facility [17] and then sent to the machine running the simulator. A reader thread receives these packets, and converts them to the corresponding simulation events. The simulator carries out the simulation by first putting the ICMP ECHO packets in the output queue of the simulated host 10.5.0.12. The packets are then forwarded over the simulated network to the designated host 10.0.1.19, which responds with ECHO REPLY packets. Once the packets return to the host 10.5.0.12, the simulator exports the events to a writer thread, which sends them to the client machine running the ping application. The client ping application finally receives the ECHO REPLY packets and prints out the result. Note that the segment of the path between the client application and the simulated host does not exist in the model. The problem is that the latencies of the physical connection can contribute a significant portion of the total round-trip delay. Simply on the forwarding path (from the client to the simulator), it may take hundreds of microseconds even on a high-speed local area network, before the emulation packet is eventually inserted into the simulator's event-list.[1] It can tremendously affect applications that are sensitive to such latencies.

Our solution to this problem is to hide the latencies due to the physical connection *inside* the simulated network. Since delays are imposed upon network packets transmitted from one router to another in simulation, we can modify the link layer model to absorb the latencies by sending the packet ahead of its due time. The simulator models the link-layer delay of a packet in two parts: the queuing time—the time to send all packets that are ahead of the packet in question, and the transmission time—the time for the packet to occupy the link before it can be successfully delivered, which we model as the sum of the link latency and the transmission delay—the latter is calculated by dividing the packet size by the link's transmission rate. Assuming that packets are sent in first-in-first-out (FIFO) order, the time required to transmit a packet is known as soon as the packet enters the queue at the link layer. Note that, if the FIFO ordering is not observed (e.g., packets are prioritized according to their types), one cannot predict the packet queuing time precisely. Furthermore, if we need to provide a more detailed model on lower protocol layers, the link state layer may play a significant role in determining the packet transmission time as well. In either case, we can still use a lower bound of the packet delays in our scheme. In the discussions to follow, we assume the delays are precise for better exposition.

We use a list to store the packets in the queue together with their precalculated transmission times. Let T_{now} be the current simulation time and P_0 be the last packet transmitted over the link. T_0 is the simulation time that P_0 starts transmission ($T_0 \leq T_{now}$). Let P_i be the i^{th} packet in the queue, where $0 < i \leq N$ and N is the total number of packets currently in the queue. The time to transmit packet P_i is therefore $T_i = T_0 + \sum_{j=0}^{i-1}(\lambda + \beta_j)$, where λ is the link latency and β_j is the transmission delay of packet P_j. Suppose that an ICMP ECHO packet is created externally at wall-clock time t_R, and the corresponding simulation packet P_d is injected into the simulator at time t'_R. As a result, the packet carries a virtual time deficit of $\tau_d = (t'_R - t_R)/R$, where R is the proportionality constant that indicates the emulation speed (i.e., the ratio of virtual time to real time). Rather than appending the packet to the end of the queue, we insert the packet right before packet P_k, where $k = \max\{i | i \geq 0 \text{ and } \tau_d < \sum_{j=i}^{N}(\lambda+\beta_j)\}$.[2] After inserting the packet in the queue, we reduce deficit of the packet by the total transmission times of all packets behind the packet in the queue: $\sum_{j=k}^{N}(\lambda + \beta_j)$. Further improvement can be made to transmit the emulation packets even earlier. When a packet with a deficit becomes the head of the queue, we can simulate the packet transmission in zero simulation time. That is, we can further reduce the deficit by the packet's transmission time. Note that in iSSF the delay of the link that connects hosts belonging to two separate timelines is used to calculate the lookahead for the conservative parallel synchronization protocol. It is required that the link latency λ for cross-timeline links must be larger

1 The delay includes the time for the sender's operating system to capture and send the packet, the transmission time of the packet, the time for the reader thread to receive the packet, and the time for the simulator to finally accept the event and insert it into the appropriate event-list.

2 We do this by scanning the list from the packet at the tail of the list. k=0 means that the packet is inserted at the front of the list.

than zero. In this case, we can only reduce the deficit by as much as the expected packet transmission delay.

It is reasonable to insert an event with a time deficit ahead of others in the queue. After all, were the physical connection latencies not present, the event would have entered the queue much earlier. However, in cases where the deficit is larger than the sum of transmission time of all packets in the queue (the packet is therefore inserted at the head of the queue), we can only allow the packet to continue carrying the remainder of the deficit to the next hop, and therefore preempt events at the next hop. The process continues until the deficit is reduced to zero, or the packet reaches its destination. Since we do not "unsend" packets that have been sent before the emulation packet with the deficit arrives, this scheme is simply an approximation once the deficit is carried to the next hop.

Another issue concerns accommodating the physical connection latencies in the reverse path (from the simulator to the client application). A simple solution is to assume such latencies in the reverse path to be the same as in the forwarding path, and use a deficit of the same amount for all packets traveling in that direction. The problem with this approach is that the simulated network always tries to make up for the deficit within the first few hops, while in fact such a deficit is expected at the last segment of the path from the simulator to the application client. This means the interactions between the packets with deficits and other packets in simulation do not represent reality. We expect that, since in large-network simulations there are much fewer emulation packets than simulation packets, the effect of such a distortion may not be significant at all. We plan to quantify such effect in our future work.

4. Traffic, Attacks, and Routing

iSSFNet includes several novel techniques for modeling traffic, network attacks, and routing of traffic flows. A key technique employed in iSSFNet to make real-time simulation of large networks feasible is multi-resolution representations of traffic whereby the level of detail with which a traffic flow is simulated depends on how interested we are in the detailed dynamics of the flow. Traffic that is "in focus" (*foreground traffic*) is simulated with high fidelity at packet-level detail. Traffic that represents other things going on in the network (i.e., *background traffic*) is abstracted using fluid modeling, either using fine grained per-flow models, or coarse time-scale periodic fixed point solutions.

Fluid modeling [11, 15] is being explored also in other network simulators, such as MAYA [31], IP-TN [12], and HDCF-NS/pdns [24]. The models used in iSSFNet are based on our previous work to develop discrete-event fluid modeling of TCP and hybrid traffic interaction models such that the packet and fluid representations can coexist in the same simulation [21]. Recent work has addressed coarser models using fixed point solution techniques [22] of flow competition through a network, permitting several orders of magnitude speedups [19] and thus making it possible to represent larger networks and more flows in real time.

Attack models in RINSE focus on assets at a network resource level, i.e., things like network bandwidth, control over hosts, or computational or memory resources in hosts. Current attack models include DoS attacks, worms, and similar large-scale attacks typically involving large numbers of hosts and high intensity traffic flows. We are thus generally only interested in the coarse behavior of the attack traffic (a large volume of traffic) rather than the detailed traffic dynamics. Consequently, we leverage the coarser multi-resolution traffic models for efficient attack models, including zombie hosts emitting fluid DoS flows and fixed-point solutions of worm scan traffic intensities based on our previous work in multi-resolution worm modeling [14].

Memory and computational demands for routing of traffic have been identified as significant obstacles for large-scale network simulation and emulation. Some studies [23, 9, 2] start from the premise of shortest path routes and try to reduce computational and representational complexity through spanning tree approximations [9, 2] or lazy evaluation [23]. Others have achieved memory reductions in detailed protocol models, such as BGP (policy based routing) through implementation improvements [8, 4].

In iSSFNet we have developed a method for on-demand (lazy) computation of policy based routes, as computed by BGP [13]. For efficiency reasons and to ensure that traffic (attack traffic in particular) can address and reach a destination network even if the destination is missing, we need hierarchical addressing. Hence, our routing model is currently being extended to handle route aggregation. We are thus able to preload partial (precomputed) forwarding tables based on a priori known traffic patterns in the model, such as scripted background traffic, and compute routes for other flows as needed.

5. Modeling Device Resources

Earlier exercises of the type RINSE is targeting indicated the need to model not only limited network resources, like bandwidth, but also some aspects of constraints on computational resources in hosts and routers. Partly since they may be targeted for Denial-of-Service, but also to preclude unrealistic defensive strategies. For instance, zero cost packet filtering allows unrealistically large numbers of filters. Consequently, we need "light-weight" models of computational resources (CPU) and memory in RINSE, a problem that has not received much attention in network simulation to date since simple models have generally sufficed. For instance, a

uniformly distributed compute delay has been used in studies of simulations of BGP routing [7], or a simple fixed cryptographic cost for S-BGP processing [20]. The sensor networking community, being very conscious of the constraints imposed by tiny sensors, are particularly interested in modeling the power consumption of different components, including the CPU [26].

5.1. CPU Model in RINSE

In RINSE a fair amount of detail is necessary and we identified the following requirements on our CPU model:

- *Interference* between different CPU intensive tasks.
- *Traffic delay* could result from high CPU load–in particular during abnormal (attack) conditions.
- Possibility of *packet loss* due to sustained high load.
- *Observable CPU load:* the user should be able to monitor CPU load to diagnose the system.
- *Light weight:* we must strive for the simplest possible models that can at least approximately represent the desired effects.

Thus, we require more behavior detail than many other applications do to be able to capture, at least coarsely, interactions between different tasks and traffic flows in terms of processing. This results in significant implementation hurdles, as will be described, and the situation is also complicated by the fact that the multi-resolution representation of traffic necessitates a multi-resolution representation of computational workload (i.e. hybrid discrete and fluid representations).

Interference: to observe interference between different tasks, we need to model how processing cycles are allocated. The generic UNIX process scheduling mechanism[3] [28] is based on priority scheduling, where process priorities are continuously recomputed to try to achieve good responsiveness and latency hiding for I/O bound tasks.

We do not want to get into the details of the scheduling mechanism, but be able to observe competition for resources. Within the CPU, a set of *tasks* are defined, where a task can be thought of as a process or thread. For instance, these could be application layer processes like web clients/servers, a database server, or lower layer functionality like a firewall process doing packet filtering on incoming packets. Figure 4 illustrates how each task services the work it has to do in FCFS order, but cycles are allocated among tasks using processor sharing. In this first model we simplify the problem by assuming that the tasks we consider have roughly the same priority (same range), so that

3 It varies somewhat between different flavors. Linux has a slightly different mechanism, but for the purposes of this discussion it's essentially the same.

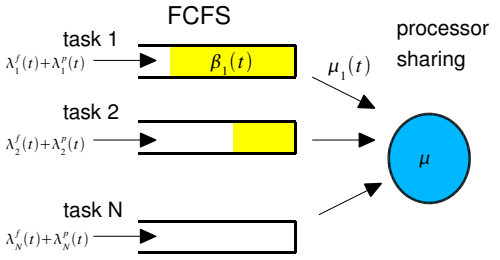

Figure 4: Processing work model where work is handled FCFS by tasks that are allocated "cycles" on the CPU

they are treated equally. The requests (incoming traffic) to each task may be a mixture of packets and fluid traffic flows. As in the hybrid packet/fluid traffic model in [21], we form a hybrid queue by fluidizing the packet load through estimating the packet rate. However, the service model interleaving the tasks actually make things even more complicated here than most hybrid traffic models since service is not FIFO. Assume there are N tasks. Let $\lambda_i^f(t)$ be the incoming fluid workload rate for task i (in cycles per second) at time t, and and μ is CPU service rate (i.e. its speed). A packet has an associated workload, w_t in cycles. By estimating the the packet arrival rate over a time window $[t', t]$, we get the estimated packet workload rate $\lambda_i^p(t)$. Let the total arriving workload for task i be $\lambda_i(t) = \lambda_i^f(t) + \lambda_i^p(t)$. We need to allocate a service rate to each task $\mu_i(t)$, determine backlog $\beta_i(t)$ and possibly lost work $\xi_i(t)$. A discrete workload arrival (packet workload) at t is always added to backlog on arrival $\beta_i(t) \leftarrow \beta_i(t) + w_t$. Note, however, that if no discrete arrivals preceded it in $[t', t]$, then $\lambda_i^p(t) = 0$.

We consider two cases:

Non-overload, the total incoming workload rate over all tasks is less than or equal to the workload service rate the CPU can handle, i.e. $\sum_i \lambda_i^f(t) + \lambda_i^p(t) \leq \mu$. In this case each task is first assigned the fluid service rate it requires $\mu_i(t) = \lambda_i^f(t) + \lambda_i^p$(t). Tasks that have any backlog ($\beta_i(t) > 0$), and this applies to any tasks processing packets, are marked as *greedy*. Let g be the number of greedy tasks. Any left-over cycles $\gamma(t) = \mu - \sum_i \mu_i(t)$ are allocated equally to greedy tasks $\mu_i(t) \leftarrow \mu_i(t) + \gamma(t)/g$. This ensures that the backlog gets drained as quickly as possible and thus packets are processed as quickly as possible. Consequently, fluid workload results in a processor utilization in proportion to the incoming rate, while discrete workload results in bursts of full utilization.

Overload, the sum of the incoming fluid workload rates and the averaged packet workload rates is greater than the service rate of the CPU. That is, there is a sustained

overload condition. In this case the tasks are denied cycles in proportion to their fraction of the total workload, and what cannot be handled accumulates as backlog.

$$\mu_i(t) = \frac{\lambda_i(t) \cdot \mu}{\sum_i \lambda_i(t)} \quad (1)$$

An arriving discrete workload (packet) that does not yet have an average rate estimate poses a problem in this case. It is given $\mu_i(t) = 1$ (full utilization) without affecting other flows. This is unrealistic in that the total CPU service rate is now briefly more than μ, but is a reasonable approximation for occasional packets. If the packet is the first in a series with high average workload rate, then the service rates will be corrected the moment the first arrival rate estimate is calculated.

When a task is defined, a buffer space size b_i can be assigned to it to limit the backlog and introduce the possibility of loss of work if the task cannot keep up. Packets occupy buffer space according to their size until serviced. Fluid flows are assumed to have a simple linear relationship between the workload rate (cycles/second) and memory used for backlog rate (bytes/second). Modeling loss in hybrid queues is a delicate matter, as pointed out in [21]. If a discrete workload (packet) arrives to a back-logged task queue such that there is not enough space to fit it in the buffer we consider the state of the queue. If it is draining, the average arrival rate is less than the service rate, and we assume that it will fit (replacing fluid buffer space with the packet). If the queue is filling, we give the packet a probability of fitting into the queue equal to its proportion of the total task load $p = \lambda_i^p(t)/(\lambda_i^f(t) + \lambda_i^p(t))$.

In an overload condition tasks become coupled through competition for CPU and through the traffic flow, with fluid loads possibly leading to a cyclic dependency of traffic and CPU work; an unexpected complication. Figure 5 illustrates how we consider the cost of filtering and traffic forwarding in a firewall router, and if the CPU gets overloaded it needs to report back to the protocol layers so that they can reduce the traffic rate emitted. However, since the traffic flow passes first through filtering (A) and then forwarding (B) there is a feedback loop in terms of rate adjustments. When B changes its load to the CPU, it must update the serviced load for A. A must then update the traffic rate emitted to B, which must then perform another load update to the CPU. For n tandem tasks, where work is proportional to flow rate, the principle of proportional loss (equation 1) limits the feedback. Consider the i:th task. Let f_i be the inflow, $\lambda_i = k_i \cdot f_i$ be the (offered) workload, and c_i^n be the cycles allocated for task i. Initially, flow rate f_1 is sent through all tasks, so equation 1 implies we allocate cycles as $c_i^n = k_i / \sum_{j=1}^n k_j$. Tandem dependencies means

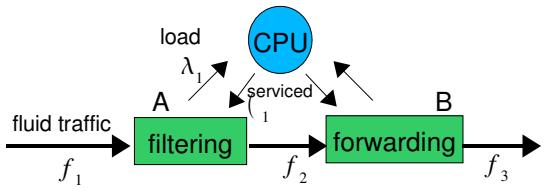

Figure 5: CPU control feedback on tasks and fluid flows

that $f_i = (c_{i-1}^n/\lambda_{i-1}) \cdot f_{i-1}$, and thus

$$\lambda_i = k_i \frac{c_{i-1}^n}{\lambda_{i-1}} f_{i-1} = \frac{k_i \cdot k_{i-1} f_{i-1} \cdot f_{i-1}}{\sum_{j=1}^n k_j f_{i-1} k_{i-1} f_{i-1}} = \frac{k_i}{\sum_{j=1}^n k_j}$$

That is, the required cycles λ_i to handle the adjusted inflow f_i equals the fraction of cycles assigned c_i^n, so the allocation stabilizes immediately. But completely avoiding this feedback loop does not appear possible, so we rate limit the feedback from the CPU to the protocol layers. Through this rate limiting, we mimic the control delay imposed by the scheduling mechanism and bound the computational costs in the model.

Traffic delay: one difficult issue was how to implement delays within the protocol stack without incurring significant overheads and code complexity. iSSFNet uses a protocol model inspired by the x-kernel design [10], where protocol sessions have a well defined common interface through which they can be plugged together. These are the push and pop methods. For maximum efficiency, the programming patterns used in the protocol stack are based on event-orientation through timer objects and continuations. Rather than switch to process-orientation to support arbitrary suspension points for packet processing delays, we opted to limit the possible suspension points to the push/pop entry interfaces (the socket API for the application layer). Thus, multiple delays on a packet within one protocol session will be merged into one delay that is not incurred until the point where the packet enters the next protocol session. The push/pop API's are good candidate suspension points because the state of processing of a packet (or a fluid flow) is passed in the packet itself along with a small number of additional parameters. Hence, we can safely assume that there are no additional state variables earlier in the execution stack that need saving. So, upon return we continue processing from the push or pop call without reconstructing the process stack. Other data structures in the protocol sessions, such as queues of packets that have been delayed pending some condition, evolve over time and thus do not require saving.

The accumulated delay for a packet within a protocol session is stored in the packet and thus detected as the packet reaches the next push/pop suspension point. Suspen-

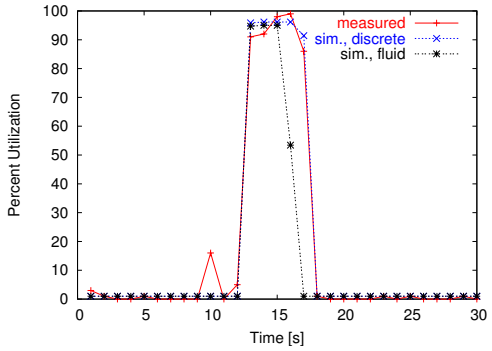

Figure 6: Example: `scp` transfer

sion points can be enabled or disabled through the DML configuration; the idea being to make it easy to aggregate delays, and thus aggregate events, by having fewer enabled suspension points. Displacing the suspension point from the point in the code where the delay should take place alters the causal ordering of state modifications in the model, i.e. the interleaving of updates in simulation time will be slightly altered. We believe this will not be a significant issue for the protocols under consideration here, but more experience with the model will be needed to bear this out.

5.2. Example

We illustrate the CPU model through a very simple example. In an experiment a 41.6 MB file was downloaded from a Linux laptop (acting as the server) using `scp` (secure copy). The CPU load on the data source (server) was monitored using `vmstat`. Figure 6 shows the CPU utilization during the transfer as "measured". This scenario was modeled in iSSFNet using its packet level TCP model. A client host is connected directly to a server host through a 100 Mb/s link. When modeling the CPU load, we have two choices: use a fluid representation of the load on the CPU, or use discrete chunks of work. The fluid representation is simple to use and has very low simulation cost. The drawback is that it is coarse and will not impose any delay on the the packets. Discrete work is more expensive to simulate, but is more fine-grained and delays packets.

Using fluid work, we simply call a `setFluid` method on the CPU, as the transfer starts, to set the instruction rate during the transfer (we simply match the observed utilization). When the transfer is completed the instruction rate is set back to zero. The result, shown as "fluid load", indicates a shorter transfer time than what was measured. Alternatively, we can use discrete workloads. Examining the OpenSSH scp implementation indicates that it transfers data through a 2 KB buffer, so we write data to the socket in 2 KB blocks and impose a compute delay on each block for data transfer and encryption. The computation cost is registered through a call to `cpu.use(...)` with the number of instructions used and a pointer to the socket being used. The socket `send()` code hides a call to `cpu.delay(...)` causing the socket processing to be suspended and delayed. We also use a timer to add a small idle delay between each block to model latencies. After tuning these delays, the result shown as "discrete load", can be made to match reality fairly well.

There is a significant difference in simulation cost between these two approaches. Using fluid CPU load, no extra events are added by the CPU model, but with the discrete workload model, each block requires a resource departure event and results in an event for drained backlog. Thus, the total event count increases by a factor of about 2.4 and the execution time by a factor of 4. It is up to the modeler to determine when the additional cost is justified.

Aside from approximations arising from implementation decisions, the current CPU resource model represents many simplifications. The principle of proportional loss is frequently used for fluid traffic and alleviates the allocation feedback issue mentioned previously. But we see the need for more emphasis on distinction of task priorities to better mimic prioritization of processes and threads. For instance, kernel level processes should be largely insulated from demands at the user level. We are looking into new allocation policies that can prioritize demands.

6. Summary and Future Work

RINSE incorporates recent work on *i)* real-time interaction/emulation support, *ii)* multi-resolution traffic modeling, *iii)* efficient attack models, *iv)* efficient routing simulation, and *v)* CPU/memory resource models, to target large-scale preparedness and training exercises. Described here were efficient CPU/memory models necessary for the scenario exercises, and a latency absorption technique that will help when extending the range of client tools usable by the players.

Aside from model refinements, our ongoing and future work includes more fundamental issues such as supporting fault tolerance and efficient real-time scheduling of compute intensive tasks like background traffic calculations and major routing changes. For example, we would like our simulation framework to permit certain background tasks, such as background traffic calculations, to be adaptively scheduled based on higher priority load.

Acknowledgements This research was supported in part by DARPA Contract N66001-96-C-8530, NSF Grant CCR-0209144, and DHS Office for Domestic Preparedness award 2000-DT-CX-K001. Thus, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce this contribution. Points of view expressed are those of the

authors and do not necessarily represent the official position of the U.S. Department of Homeland Security.

References

[1] T. Bridis. Gov't simulates terrorist cyberattack. Associated Press, `http://www.zone-h.org/en/news/-read/id=3728`, November 2003.

[2] J. Chen, D. Gupta, K. Vishwanath, A. Snoeren, and A. Vahdat. Routing in an Internet-scale network emulator. In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2004.

[3] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global Internet. *Computing in Science and Engineering*, 1(1):42–50, January 1999.

[4] X. Dimitropoulos and G. Riley. Large-scale simulation models of BGP. In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2004.

[5] K. Fall. Network emulation in the Vint/NS simulator. In *4th IEEE Symposium on Computers and Communications (ISCC'99)*, pages 244–250, July 1999.

[6] R. Fujimoto, K. Perumalla, A. Park, H. Wu, M. Ammar, and G. Riley. Large-scale network simulation – how big? how fast? In *Symposium on Modeling, Analysis and Simulation of Computer Telecommunication Systems (MASCOTS)*, October 2003.

[7] T. Griffin and B. Premore. An experimental analysis of bgp convergence time. In *9th International Conference on Network Protocols (ICNP)*, November 2001.

[8] F. Hao and P. Koppol. An Internet scale simulation setup for BGP. *ACM SIGCOMM Computer Communication Review*, 33(3):43–57, July 2003.

[9] P. Huang and J. Heidemann. Minimizing routing state for light-weight network simulation. In *Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, August 2001.

[10] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[11] G. Kesidis, A. Singh, D. Cheung, and W. W. Kwok. Feasibility of fluid-driven simulation for atm network. In *Proceedings of IEEE Globecom*, November 1996.

[12] C. Kiddle, R. Simmonds, C. Williamson, and B. Unger. Hybrid packet/fluid flow network simulation. In *17th Workshop on Parallel and Distributed Simulation*, June 2003.

[13] M. Liljenstam and D. Nicol. On-demand computation of policy based routes for large-scale network simulation. In *2004 Winter Simulation Conference (WSC)*, December 2004.

[14] M. Liljenstam, D. Nicol, V. Berk, and R. Gray. Simulating realistic network worm traffic for worm warning system design and testing. In *2003 ACM Workshop on Rapid Malcode (WORM)*, October 2003.

[15] B. Liu, D. R. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *IEEE Infocom*, Anchorage, Alaska, April 2001.

[16] X. Liu, H. Xia, and A. Chien. Network emulation tools for modeling grid behavior. In *3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[17] S. McCanne and Van Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Winter USENIX Conference*, pages 259–269, January 1993.

[18] D. Nicol and J. Liu. Composite synchronization in parallel discrete-event simulation. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):433–446, May 2002.

[19] D. Nicol, J. Liu, M. Liljenstam, and G. Yan. Simulation of Large-Scale Networks Using SSF. In *Winter Simulation Conference (WSC)*, December 2003.

[20] D. Nicol, S. Smith, and M. Zhao. Evaluation of efficient security for BGP route announcements using parallel simulation. *Simulation Pratice and Theory Journal, special issue on Modeling and Simulation of Distributed Systems and Networks*, June 2004.

[21] D. Nicol and G. Yan. Discrete event fluid modeling of background TCP traffic. *ACM Transactions on Modeling and Computer Simulation*, 14:1–39, July 2004.

[22] D. Nicol and G. Yan. Simulation of network traffic at coarse time-scales. In *Workshop on Principles of Advanced and Distributed Simulation (PADS)*, 2005.

[23] G. Riley, M. Ammar, and R. Fujimoto. Stateless routing in network simulations. In *Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, May 2000.

[24] G. Riley, T. Jaafar, and R. Fujimoto. Integrated fluid and packet network simulations. In *Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, October 2002.

[25] L. Rizzo. Dummynet: a simple approach to the evaulation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, January 1997.

[26] A. Savvides, S. Park, and M. B. Srivastava. On modeling networks of wireless micro sensors. In *SIGMETRICS*, June 2001.

[27] R. Simmonds and B. Unger. Towards scalable network emulation. *Computer Communications*, 26(3):264–277, February 2003.

[28] A. Tanenbaum. *Modern Operating Systems, 2nd ed.* Prentice Hall, Upper Saddle River, NJ, 2001.

[29] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large scale network emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, December 2002.

[30] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, December 2002.

[31] J. Zhou, Z. Ji, M. Takai, and R. Bagrodia. MAYA: integrating hybrid network modeling to the physical world. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14(2):149–169, April 2004.