An Extensible Platform for Evaluating Security Protocols

Seny Kamara, Darren Davis, Lucas Ballard, Ryan Caudy and Fabian Monrose
{*seny, ddavis, lucas, caudy, fabian*}*@cs.jhu.edu*
Computer Science Department
Johns Hopkins University, USA

Abstract

We present a discrete-event network simulator, called Simnet, designed specifically for analyzing network-security protocols. The design and implementation is focused on simplicity of abstraction and extensibility. Moreover, its modular architecture allows operators to dynamically customize running simulations. To demonstrate its strengths we present cases studies that focus on examining security-centric problem domains. In particular, we present an analysis of worm propagation modeling for worms with varying target selection algorithms on topologies representing a few million hosts. Additionally, we examine the use of countermeasures such as aggregate congestion control as a defense against DDoS attacks, and present analysis for a variant called direct-Pushback. Lastly, we provide an empirical analysis of the computational and bandwidth overhead induced by proposed security extensions to DNS. These experiments hopefully illustrate that Simnet is not only scalable and efficient, but provides a viable platform for prototyping and analyzing non-trivial security protocols — a task which we argue cannot be easily accomplished elsewhere.

1. Introduction

The explosive growth of the Internet and our ever-increasing dependency on the services it provides has made it a part of the critical infrastructure of many societies and economies. Its scope has expanded from providing mostly email and web browsing services to include e-commerce, peer-to-peer networking, multimedia broadcasting and instant messaging. While providing great benefits, the ubiquity of these services has made the Internet more vulnerable to network based threats such as viruses, worms and denial of service attacks (DoS). These threats have helped focus attention on network security and have motivated efforts to design secure and practical solutions to network-centric attacks. However, as with any engineering effort, network security solutions require rigorous testing and validation — while small scale implementations are feasible for prototyping, large-scale deployment and testing is infeasible. In that regard, by providing a low cost, controllable and repeatable testing environment, we strive to offer a useful platform for analyzing network based attacks and countermeasures.

In this paper we present Simnet, an object-oriented discrete-event process based network simulator [4, 33]. Simnet is written in Java[TM] and is specifically designed to allow for rapid, yet efficient, development and examination of security related protocols. To that end, we strive to provide an intuitive and modular framework that can be used to implement and analyze a myriad of network-centric problems and solutions, based upon realistic topologies. We believe that in that respect, Simnet offers four major strengths:

- *Modularity*: its architecture includes a plugin framework that allows users to easily extend its base functionality. Various components such as transport level protocols, applications, routing protocols, filtering and queuing disciplines and the user interface can either be extended or replaced at will.

- *Code portability*: since our design follows the Java networking API as closely as possible, pre-existing Java networking code can be easily ported to Simnet. Similarly, protocols designed and analyzed in the simulator can be readily adapted to the Java networking API.

- *Dynamic customization*: in an effort to reduce the time spent testing modules, Simnet takes advantage of Java's class loading features to enable users to add and remove simulated components *during* running simulations.

- *Scalability*: its multi-threaded design allows us to directly exploit the efficiency provided by parallel architectures. However, since a multi-threaded design may be costly on single processor machines, Simnet enables users to perform simulations at various abstraction levels.

Though Simnet is primarily targeted at network security research, we believe it may prove useful to the wider networking community as well. We note that Simnet has already been used to examine a wide variety of security protocols, with the most noteworthy examples including implementations of Synkill [36], DNSSEC [10], Kerberos V4 [40], VPNs and Onion Routing [15], and probabilistic packet marking schemes including variants of [35, 38, 37]. We argue that the fact that Simnet allows for rapid prototyping of such advanced techniques is, in and of itself, a testament to its ease of use and modularity.

The remainder of the paper is organized as follows. Section 2 reviews related work in network simulation. Sections 3 and 4 describe Simnet's network and customization architectures, respectively. Section 5 presents case studies which we believe are reflective of our envisioned use, and which demonstrate the power of the simulator. The first study presents a simulation of the propagation of zero-day worms, including variants with target selection algorithms similar to Nimda [8] and Code Red II [11]. The second is an analysis of a variant of Pushback [25] when used as a defensive measure against DDoS attacks. The third study presents an evaluation of the impact of proposed security extensions to DNS. We present closing remarks in Section 6.

2. Related Work

At its core, simulation is a technique for building a model of a real or proposed system so that the behavior of the system under specific conditions may be studied. Ideally, one needs to model the behavior of the system as time progresses, and discrete-event simulation is one way to observe such time-based behavior. Discrete-event simulation can be achieved using a variety of design and implementation techniques; we refer the interested reader to [4, 33] for an introduction to the topic.

There are a multitude of network simulators available for research and education. Some are designed to model specific networking technologies, while others such as *ns* [3], Opnet [31] and cnet [26, 27] model a variety of network characteristics. *ns*, developed by the VINT project, is perhaps the most widely used discrete-event simulator for research in network protocol design. Written in C, it offers a large library of routing and inter-networking protocols and can model a diverse set of link layer technologies. In addition, *ns* is scalable and can handle large topologies by varying the level of detail in its simulations. It has been used extensively in the design and study of a variety of communication technologies, including routing protocols, inter-networking protocols, wireless and satellite networks and packet scheduling algorithms. However, though *ns* is powerful and remains the simulator of choice for modeling intricate systems on large topologies, we believe it is

complex and has a high learning curve, which unfortunately makes it not particularly well suited for education and rapid prototyping. While Simnet was not designed to replace *ns*, we feel it addresses some of the aforementioned weaknesses, and offers a viable alternative particularly for analyzing interactions in intricate security protocols such as Kerberos [40].

By comparison, the *Opnet* modeler [31] is a commercial network simulation environment written in C++. *Opnet* is quite modular and provides a high end GUI to facilitate model development. While feature rich, Opnet is neither open source nor readily available. In [6] the Johns Hopkins Applied Physics Lab DDOS-DATA project used the *Opnet* modeler to simulate various DDoS mitigation technologies and to compare their individual and combined effectiveness against DDoS attacks. The simulated technologies included a challenge-response protocol based on the use of client puzzles [21], Synkill [36] – an optimistic protocol for limiting the impact of synfloods – and rate limiting. By modeling these defenses the DDOS-DATA project demonstrated that *Opnet* can be successfully used for network security research. We believe Simnet can be as effective, and in fact, a myriad of mitigation technologies including those deployed in [6] have been implemented within Simnet in classroom settings. However, due to space constraints, we only discuss experiments which we believe are more complex and more challenging than those conducted with *Opnet* in [6].

Our design goals are more closely related to those of *cnet* [26, 27] and *simmcast* [30]. Cnet is an open source network simulator developed at the University of Western Australia designed specifically for teaching. However, *cnet* is intended for experimentation with various data-link and network layer protocols, and only provides the application and physical layers; its users are required to implement the transport, networking and link layers as needed. Moreover, while *cnet* offers a well designed and useful Tcl/Tk GUI, its overall architecture and event-driven programming style can be unintuitive [27]. *Simmcast* [30], on the other hand, offers a process-based platform similar to ours, but focuses only on providing a scalable implementation of the primitives required to effectively support group communication protocols (e.g, `join`, `leave`, `send`, etc.).

3. Overview of the Simnet Architecture

Simnet is initiated by first loading the network topology to be simulated. These topologies are described in plain ASCII using a format similar to the Otter [1] [19] file format. Once parsed, an internal representation of the network is generated and instantiated. From this point onward, users

---

[1]To allow users to graphically display the instantiated network, we provide limited support for CAIDA's Otter network visualization tool.

control the simulation interactively using commands available from the user interface or via scripts.

For ease of exposition, we divide Simnet's architecture into layers that span the high level components that encompass the system itself, to the low level components that are used to build the simulated entities. These design layers are structured as follows:

1. *System Architecture:* The highest level components of Simnet and includes (*i*) the User Interface (UI) which interprets user commands and scripts to control the simulation (*ii*) the Topology Parser, which parses network topology files and instantiates the data structures used to represent the network and (*iii*) the simulator which stores the structures used to represent the network.

2. *Node architecture:* The data structures that represent network hosts and routers. These includes the Link Processor, transport level protocols, applications, datagram sockets and IP filters.

3. *Network Architecture:* The data structures that represent the network internals. These structures essentially model links, hosts, routers, packets and routing algorithms.

3.1. Node and Network Architectures

At a high level, the network architecture provides the core functionality for modeling nodes, links and packets. More specifically, the network architecture is concerned with the representation of components of a network that encompass the routing algorithms, hosts, routers, links, Ethernet frames, and protocols such as IP, TCP, UDP and ICMP. Simnet also provides a socket interface that is closely mapped to the Java Socket API. The conceptual design for each of these components is presented in the following Sections.

HOSTS Hosts reside at the edge of the network and have only one incoming and one outgoing link that connects to a router. Hence, hosts can only send and receive packets. Each host is assigned an IP address and associated gateway. By default, hosts understand subsets of IP, TCP, UDP, ICMP and DNS but can be easily extended to implement other protocols (we discuss Simnet's customization architecture in Section 4). Hosts can be viewed as entities comprising of (*i*) a *Link Processor* responsible for extracting frames from the incoming link and forwarding the enclosed IP Packets to the IP filter for that link (*ii*) *Incoming and Outgoing IP Filters (i.e., firewalls)* that accept or deny incoming or outgoing packets according to a user defined security policy (*iii*) *Incoming and Outgoing BSD Packet Filters (BPFs [7])*

that allow for user-level packet capture by directly passing selected incoming or outgoing packets to an associated object (*iv*) an *Early Drop Policy* engine that drops outgoing packets (according to a given policy) before they are placed onto the outgoing link (*v*) *Transports* that implement transport level protocols such as TCP, UDP, and ICMP and (*vi*) user-level *Applications*.

To achieve selective filtering of packets, we support both IP filters and BSD packet filters. With IP filters each node has one filter associated with each incoming and outgoing link. This allows nodes to filter traffic based not only on IP and TCP headers but also on the incoming and outgoing link. In addition, user-level packet capture is supported via Berkeley packet filters. The BSD Packet Filter is a component of the BSD TCP/IP suite that allows user level programs to access IP packets. Any object can instantiate a BPF, set its filter and register it with a node.

ROUTERS Routers reside at either the core or edge of the network and can have multiple incoming and outgoing links. The overall behavior of a router is similar to a host, with the exception that it can forward (and filter) packets on multiple incoming and outgoing links. Each router is assigned an IP address and an associated subnet which it serves. By default, routers understand subsets of IP, TCP, ICMP and UDP and can be easily extended to understand other transport level protocols. By default, routing in Simnet is static and uses shortest path [41] though dynamic routing using the Routing Information Protocol [17] is also supported.

For improved scalability, the concept of *aggregate routers* is supported by the Simnet architecture. In its simplest form, the *aggregate router* functionality allows for a single node to serve as the access point for all reachable hosts within a given prefix. That is, if an AS serves as an *aggregate router* for prefix P, then for any reachable host within P, R responds on its behalf. In this way an entire class C network, for example, may be simulated using a single node.

LINKS The data link layer is modeled by queues that maintain references to two nodes: a *to*-node and a *from*-node. While individual links are unidirectional, we emulate bidirectional links between nodes by creating two links in opposite directions. Moreover, links can be dynamically manipulated by changing their bandwidth, latency, queue size and loss rate at any point during a running simulation. As we show later (see Section 5.2) such functionality is essential when attempting to analyze realistic attacks and countermeasures. Besides modeling bandwidth and latency, our links also model packet loss. In fact, links discard frames either naturally, according to an early drop policy or according to a packet loss distribution[14].

4. Plugin Architecture

To achieve modularity and flexibility we introduced a plugin architecture into Simnet. This functionality allows users to implement their own components that are dynamically loaded into the simulator. Our initial goals for designing such an architecture were twofold: (*1*) making the customization process simple and transparent to the user, and (*2*) allowing for dynamic customization that, for example, extends the default plugins. Dynamic customization takes advantage of Java's class loading features and allows users to customize Simnet at any point in time during a running session. This includes customizing components that directly relate to the simulation (i.e. parts of the network architecture) or indirectly (i.e. parts of the system architecture).

As a motivating example consider a user who wishes to simultaneously test different implementations (but abiding to the same specification) of a virtual private network (VPN) [43] protocol against each other. Static customization requires her to start a new simulation for each test, while dynamic customization allows her to perform all tests within the same simulation. This greatly reduces the time and effort spent testing and validating simulations, which is especially useful in prototyping. To implement dynamic customization, Simnet leverages Java's dynamic class loading capabilities[2]. Once a class is loaded into the Java Virtual Machine (JVM) our plugin architecture becomes responsible for adding, removing and replacing its instances from Simnet's data structures.

DYNAMIC DISPATCH Our plugin architecture allows certain components of Simnet to be dynamically loaded, unloaded or replaced. To achieve this the architecture makes use of three sets of Java classes: *plugin types*, *plugin managers* and *plugins*; and two Java interfaces: *pluggable* and *plugout listener*.

A plugin type is a class that represents one of Simnet's plugin components. Each plugin type defines a set of abstract methods that are used to interact with instantiations of the given type. These instantiations are referred to as plugins and must extend a plugin type and implement the corresponding abstract methods. Each kind of plugin is stored in different locations and in different data structures referred to as its *storage location*. This implies that Simnet cannot access and modify all plugins in a uniform manner. For transparency (to the user) and to provide a consistent interface to any object needing access to *plugins* we require that each plugin type have an associated plugin manager that

is responsible for performing all storage operations. These operations include initializing, finding, adding and replacing plugins.

In addition to the previously defined classes, the plugin architecture uses two interfaces to facilitate the dynamic removal and replacement of plugins. The pluggable interface defines a call-back mechanism that is called before a plugin is removed. This allows the plugin to take action before being replaced, and provides the opportunity to transfer its state to its replacement. This proves useful, for example, if an IP filter is being dynamically replaced. In this case, the outgoing filter can transfer all its rules to the new incoming filter. The second interface is a plugout listener which enables objects to register themselves with the simulator in order to be notified when plugins are either replaced or removed.

This (stateful) dynamic loading capability allows us to, for example, maintain long lived TCP connections even when the underlying TCP stacks get updated — for instance, if the underlying stack is replaced by one resilient to TCP SYN flood attacks.

5. Case Studies

To better illustrate the strengths of the simulator, we describe and analyze three experiments conducted using Simnet. The goal of each is to illustrate a particular attribute of the simulator. The first study examines the spread of worms with varying target selection algorithms. While somewhat similar analysis on worm propagation exist elsewhere, e.g. [44, 9, 45], we believe that because effective demonstration of worm modeling requires very large topologies, this serves as a good exemplar of the scalability of our architecture. The second experiment studies the effect of various aggregate congestion control (ACC) mechanisms [25] in mitigating DDoS attacks. Because these ACC mechanisms rely heavily on bandwidth and latency information, we use this experiment to illustrate the accuracy of Simnet's link latency and bandwidth modeling. Lastly, a third study evaluates the computational and network overhead of security extensions to DNS protocol [10] and illustrates how Simnet's ease of use and modularity enables one to rapidly prototype and evaluate non-trivial protocols.

5.1. Worm Propagation

To simulate the effects of the spread of a worm on the Internet, it is essential that a relatively large, realistic, topology be used. Therefore, we base our simulations on a subset of an AS-level topology derived from the University of Oregon's Route Views project [28]. Route Views provides access to information about the routing system from the perspective of several different backbones and locations

---

[2]While Java class loaders can load classes into the Java Virtual Machine dynamically they cannot unload them [16]. However a technique known as hot deployment can be used to achieve dynamic replacement and unloading [23].

around the Internet. The chosen topologies consists of a set of nodes selected as follows: For each class A network, denoted as P_a, announced in the Routing Information Database (RIB) we choose at most k class B networks, $\{P_{a_1}, \ldots, P_{a_k}\}$, within the corresponding /8. For each P_{a_i} we consider each AS_PATH that reaches it. For simplicity, we assume that the final AS in the path $\{AS_j \rightarrow AS_{j+1} \ldots \rightarrow AS_{j+n}\}$, owns P_{a_i}[3]. AS_{j+n} is inserted in our topology as a router owning the address space spanned by P_{a_i}. However, given that a particular AS present in a chosen path may not have been inserted by the above selection criteria, transit ASes were inserted to abide with the connectivity implied by the AS_PATHs. By varying k and excluding certain P_a's we generated 5 topologies that were approximately multiples of 50.

To effectively model this task we make use of the *aggregate router* functionality provided by Simnet — each of the 192 nodes (excluding transit ASes) selected from the RIB are modeled by an *aggregate router*. Once instantiated, each *aggregate router* loads configuration parameters $< \alpha, \beta >$ used to model its address space allocation – α corresponds to the percentage of /24 within the prefix(es) owned by the aggregate that should be reachable; β corresponds to what percentage of the IP addresses within each reachable network are allocated. Setting the parameters $< \alpha = 0.4, \ \beta = 0.7 >$ yields a topology that is expected to comprise of approximately 2 million hosts. Worm propagation can then be modeled as a function of $< \alpha, \beta, v, r, w >$ where v denotes the percentage of reachable host that are vulnerable, r is the probe rate per infected host per second, and $w(i)$ is the probability of an infected host using the first i bits of its own address as a prefix in the target's address. Here, we only use $w(16)$, $w(8)$, and $w(0)$ which corresponds to the probabilities of staying within a host's class B network, staying within a host's class A network, and choosing any IP address, respectively[4]. In our experiments, we used $v = 0.25$, which results in nearly half a million hosts in our topology that can be infected by the worm.

To simulate worm propagation, a *worm modeler* is plugged in on all *aggregate routers*. As the name implies, the *worm modeler* analyzes the propagation characteristics based on various levels of susceptibility (v), probing rates (r) and target selection (w). We note that the worms modeled here are not meant to be a specific real worm, but instead share several characteristics with known worms. In particular, we examine target selection algorithms such as (*i*) a naive selection algorithm (*ii*) weighted selection as evidenced by Code Red II [11], and (*iii*) aggressive scanning as observed in Nimda [8].

Moreover, for analytical purposes we make the following

simplifying assumptions. First, a vulnerable host becomes infected once it receives a single UDP *probe* packet from an infected host—we believe this assumption is justified by the recent SQLSlammer Worm [39]. Second, we assume that once a host is infected, it remains infected by exactly one copy of a worm. Finally, the more aggressive worms attempt to be stealthy by reducing their probe rate, r, while the more naive worms increase their probing rate for effectiveness.

EXPERIMENTAL METHODOLOGY To perform such a large scale simulation within a relatively short time period, the global knowledge of the topology provided by the simulator must be leveraged. Let $\mathrm{net}(x)$ denote the class B network that the address x is in. Consider an infected host with IP address a such that $\mathrm{net}(a)$ is modeled by an *aggregate router* running a *worm modeler*. Let T_i denote the set of addresses that have the same $i \in \{16, 8, 0\}$ bit prefix as a. The *worm modeler* can only choose targets from T_i where i is chosen with pdf $w(i)$. The *worm modeler* has knowledge of the set class B networks existing in the topology (N), and the set of vulnerable[5] hosts ($V_{\mathrm{net}(a)}$) in *its* class B network.

Using the knowledge of N and $V_{\mathrm{net}(a)}$, a *worm modeler* can eliminate some targets from T_i which cannot possibly be vulnerable. The addresses that can be removed from T_i are all remote addresses in networks that do not exist, and all local addresses that are not vulnerable. More precisely, the addresses that can be removed are in the set

$$
\begin{aligned}
R_i \ = \ & \{g : g \in T_i, \mathrm{net}(g) \neq \mathrm{net}(a), \mathrm{net}(g) \notin N\} \cup \\
& \{g : g \in T_i, \mathrm{net}(g) = \mathrm{net}(a), g \notin V_{\mathrm{net}(a)}\}
\end{aligned}
$$

The smallest set of possibly vulnerable hosts that the *worm modeler* can determine using its knowledge of the topology is $P_i = T_i - R_i$. Then the probability of an address $x \in P_i$ selected uniformly at random from T_i is $p_i = \frac{|P_i|}{|T_i|}$. Furthermore, the number of addresses that must be selected from T_i until one is also in P_i is simply a geometric random variable [34] with probability p_i.

Each probe sent to an address selected from T_i can be thought of as having a *cost* of 1. The number of selections from T_i that are needed before choosing one in P_i is the cost of selecting an address directly from P_i. Since this is a random variable with the previously described distribution, rather than sending probes to targets in T_i each with a cost of 1, the *worm modeler* can send probes to targets in P_i each with a cost given by a random variable.

This cost is determined as follows. During each second, an AS with k infected hosts accumulates $k \times r \times w(i)$ *credits*. For each i, while the number of accumulated credits is

---

[3]If P_{a_i} is MOASed we choose the first AS encountered that owns P_{a_i}.
[4]For notational simplicity, we write $w = (w(16), w(8), w(0))$.

[5]In this Section, when referring to vulnerable hosts, we mean vulnerable but not yet infected.

greater than the cost of choosing a target in P_i, a probe is sent to a node in P_i. To prevent truncation errors from influencing the results, unused credits are allowed to accumulate over time. When $i = 16$, the chosen target is removed from $V_{\text{net}(a)}$ and P_{16}, which increases the cost of infecting a new target within $\text{net}(a)$ as time goes by.

Notice that real-time simulation of worm propagation in this manner can be somewhat problematic; after a *worm modeler* spends e ms sending all the probes for its AS, it waits for $(1000 - e)$ ms until the next *round* begins. As such, cases will undoubtedly arise where one second is not sufficient for all *worm modelers* to send their packets, or alternatively, all *worm modelers* will finish sending their packets before a second elapses. In such cases, we can dynamically adjust the threshold between rounds to improve scalability. The optimal strategy for progressing to the next round is to wait for exactly the time needed for all *worm modelers* to complete sending their packets in the current round[6].
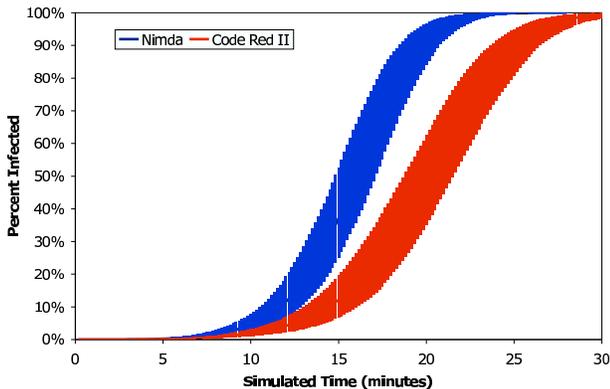
Figure 1. Overall infection rates: for Nimda, $w = (0.5, 0.25, 0.25)$; for Code Red II, $w = (0.375, 0.5, 0.125)$. With $r = 0.5$, after ≈ 20 minutes of simulated time, Nimda infected on average 45% more hosts than Code Red II (82% vs 37% of all vulnerable hosts).

EXPERIMENTAL RESULTS Figure 1 depicts the results of worm propagation for the different target selection algorithms. As also noted by [29], worm propagation typically follows a logistical growth rate – the number of infections grows exponentially until a majority of hosts are infected, and then the infection rate slows down exponentially. The rate at which vulnerable hosts are infected in the early stages greatly effects the time it takes the worm to spread. The results depict the average infection rate of each worm as a line, and the shaded region as the 90% confidence interval (i.e, between the 5th and 95th percentiles

[6] Note that while the sending of worm packets is performed in synchronized rounds, packets are received asynchronously.

of more than 100 tests). Experiments were conducted on a dual-processor 1.3 GHz XServe G4, with 1024 MB main memory, running Mac OS 10.2.6.

The graphs shows, for example, in the case of Nimda, that after 16 minutes of simulated time the confidence interval spans 30% of the infected hosts. Moreover, when both worms have the same probing rate, $r = 0.5$, after ≈ 20 minutes of simulated time Nimda infected on average 45% more hosts than Code Red II (82% vs 37% of all vulnerable hosts). The uniform case is not shown since even with a probing rate of 10 times that of Nimda, only 1.24% of the vulnerable hosts were infected after 1 hour of simulated time.

We believe that the modularity and flexibility offered by Simnet's architecture makes it well suited to perform experiments of this type, and can serve as an essential tool for prototyping similar extensions and for evaluating additional propagation models (e.g, [9]). In particular, we note that the *worm modeler* and aggregate plugins are just extensions of the `Application` and `Router` classes, respectively.
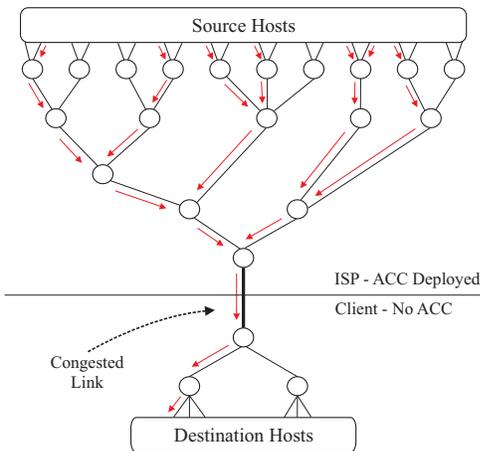
5.2. Pushback

To further illustrate the capabilities of the simulator, we present a second case study that takes advantage of the latency and bandwidth modeling provided by Simnet links. In particular, we experiment with an Aggregate-based congestion control (ACC) [25] technique proposed to deal with flooding-style DoS attacks in the Internet. ACC relies on the assumption that flood traffic causing congestion on a link is neither undifferentiated nor belongs to only a few well-defined flows, but rather, due to an intermediate case. In such cases, the packets causing the overload likely share some identifiable characteristics, and the collection of matching packets form an *aggregate*. Aggregate-based congestion control then seeks to alleviate the congestion caused by flood traffic by imposing rate limits on these aggregates *before* subjecting them to the same queuing disciplines as other traffic.

Additionally, using a mechanism known as *Pushback*, rate limits may be propagated upstream in the tree rooted at the victim (see Figure 2). In Pushback, rate limits are divided proportionally among the incoming edges at each router in the tree, and recursively propagated upstream if it appears that doing so will improve congestion (due to the aggregate) on a link. The Pushback mechanism is inspired by the observation that dropping a packet upstream can only help to alleviate congestion on the intervening links if it would have been dropped downstream anyway. This helps to protect good traffic (i.e, traffic that does not match any misbehaving aggregate) by preventing congestion due to rate limited aggregates on more links of the network. Moreover, collateral damage to poor traffic (i.e, traffic that is not

part of the aggregate but that matches the congestion signature) is also lessened by not propagating rate limits along non-contributing links.

In addition to ACC and Pushback we also study a Pushback variant called *Direct Pushback* that uses knowledge of the underlying topology and probabilistic packet marking (PPM) [35] to propagate rate limits to the furthest upstream router that observes the congestion signature (as opposed to initiating Pushback at direct neighbors). This allows for quicker relief to good traffic affected by congestion on intermediate links and to poor traffic traversing on non-contributing regions of the network. This hybrid technique is similar to other approaches suggested in [32, 20].

Our experiments are designed to test the ACC mechanisms' abilities to identify an attacking aggregate, to rate-limit it locally, and to propagate limits for the aggregate upstream towards source hosts of moderately varying depths, while following uneven distributions of attack traffic arrivals amongst incoming links.

Figure 2. Topology used in Pushback experiments

EXPERIMENTAL METHODOLOGY The topology used in our experiments is shown in Figure 2. That topology depicts is tree that represents a client network connected to an ISP. The congested link connecting the ISP to the client network has only $\frac{3}{4}$ the capacity and $\frac{2}{3}$ the queue size of each of its incoming links. Moreover, Bandwidth is allocated in the topology such that severe congestion materializes only on the link emanating from the edge of the ISP. Bad traffic (i.e., traffic that is part of a rate limited aggregate) destined for the victim was transmitted from seven of twenty source hosts at a rate of 25 packets per second per sender. A mix of good and poor traffic emanated from the remaining thirteen hosts at a rate of 10 packets per second per sender. Each packet from those thirteen hosts was destined for a randomly chosen host in the client network (6

hosts, including the victim). All packets were 128 bytes in length. Experiments lasted 10 minutes (wall time) in length; for the experiments with on/off traffic, bad traffic ceased after 5 minutes.

As discussed previously, this topology is designed to test the ACC mechanisms' abilities to identify an attacking aggregate, to rate-limit it locally, and to propagate limits for the aggregate upstream towards source hosts of moderately varying depths. The timeouts for local ACC and Pushback decisions are set to 2 and 5 seconds respectively. As in [25], we define destination-based congestion signatures under the assumption that source addresses cannot be trusted.

ACC EXPERIMENTAL RESULTS Figure 3 depicts the results of our experiments. As depicted in the baseline case, without any countermeasures, more than 50% of the link's bandwidth is consumed by attack traffic. Fortunately, the two ACC techniques each had a profound effect on the traffic pattern over the congested link. Direct Pushback, which uses more accurate knowledge about the location (based on packet markings) and rate of upstream arrivals, achieves significantly better result for good traffic and reduced the bandwidth consumed by attack traffic to $< 20\%$ of the used bandwidth.

However, as shown in Figure 3, poor traffic was decreased proportionately to bad traffic for both ACC methods, due to the use of a preferential dropping rate-limiting mechanism [25] which relies more heavily on possible incorrect estimations of arrival rates than the alternative virtual queue implementation. Clearly, both Pushback and direct Pushback were able to modify the bandwidth usage strongly in favor of non-attacking traffic, when compared with the baseline result. The last two columns in Figure 3 represent bandwidth measurements when attack traffic was ceased after 5 minutes. These results illustrate that while effective, the two different ACC techniques impose a similar penalty on poor traffic after bad traffic stopped flowing, which could have likely been further reduced by the use of more responsive virtual queue rate-limiting.

We note that Simnet's architecture was particularly well suited for empirical evaluations of techniques such as Pushback; the implementation of ACC requires few extensions to the Router and DropPolicy plugins, in addition to a more complex Application plugin to implement the Pushback protocol as specified in [25]. Most notably, its modularity allowed for such an implementation and evaluation to be conducted by undergraduate students within the limits of a semester-long project — a feat which we believe is not as easily accomplished with other simulators.
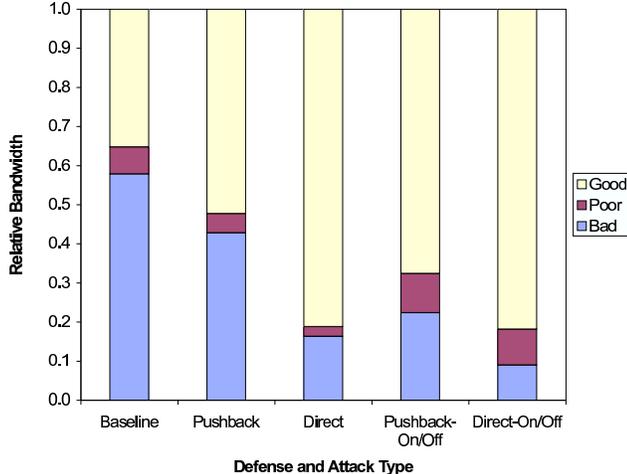
Figure 3. Bandwidth Allocation at the Congested Link During a DoS Attack Simulation

5.2.1 Discussion

The large number of probe packets generated by zero-day worms, such as Nimda [8], can be considered similar in nature to the flood of packets generated during a DDoS attack. Since Pushback is designed to mitigate the effects of DDoS attacks, we consider whether it might also be effective against such worms. We note that to the best of our knowledge, Pushback has not been proposed as a solution to worm propagation. Here, we consider the advantages of deploying Pushback both locally on ingress (i.e. as a defensive mechanism against other networks) and on a wide scale on egress (i.e. as a preventative measure in order to reduce the global infection rate of a worm).

If deployed locally on ingress, we believe Pushback will not be effective in preventing worm propagation. Since it is a defense against external traffic, it will have no effect on internal or outgoing packets — thus, clearly destination based aggregates (as suggested in [25]) will not suffice. Consequently, we expect that Pushback will slightly decrease the infection rate of worms that choose their targets uniformly (i.e. naive scanning), but will have a negligible effect on worms that target local hosts with high probability (i.e. aggressive scanning).

If deployed globally on egress, however, we believe Pushback could have an effect on worm propagation. This reasoning stems from the observation that once a network is infected, effective aggregate congestion control could potentially limit the number of probes exiting the network, which in turn would slow down the global infection rate.

5.3. DNS Security Extensions (DNSSEC)

DNSSEC embodies the security extensions proposed to secure the Domain Name Service (DNS) against attacks (such as *cache poisoning* [5]) mounted by active adversaries. These extensions essentially adopt one of two models, namely proposals based on asymmetric cryptography such as PK-DNSSEC [24], and proposals that instead advocate the use of symmetric cryptographic operations, e.g. SK-DNSSEC [2]. Here, we focus on examining the computational and bandwidth overhead imposed by the PK-DNSSEC model as it remains the leading contender to replace DNS.

In DNSSEC, each record that a resolver processes must be authenticated. To do so, special Resource Records (RRs) are incorporated in the security extensions. These RRs include signatures records (SIG RRs), public key records (KEY RRs) for the zone authoritative for a set of records, and NXT RRs for authenticating the non-existence of a requested record. As signatures expire after fixed intervals of time, every authoritative name server (re-)signs its RR sets periodically.

Upon submitting an iterative query, a local resolver receives the requested records in addition to the associated SIG RRs. The public key of the zone that created the SIG is used to verify its authenticity. In the event that the required KEY RR is not known by the local resolver, a request is issued for that RR, and the response validated.

As it is assumed that all nodes know the Root's public key, a chain of trust can always be inferred back to the Root. Once the results of an iterative query have been validated, the local resolver then authenticates its response to the stub resolver using a Transactional Signature (TSIG). The TSIG is merely a MAC (using a shared key) on the response message. We refer the interested reader to [24, 1] for more details on DNSSEC.

EXPERIMENTAL METHODOLOGY Our analysis of DNSSEC is simulated based on real-world statistics observed in [22, 42]. These results show that the Root name servers tend to observe waves of DNS requests throughout the day that correlate with working hours. To abide with this model, we let T denote the time period over which requests are generated, and subdivide this region into k intervals. Given M as the maximum number of requests per interval, each stub resolver issues $\sum_1^M r$ requests where $r = 1$ with probability $\frac{((2\pi tk/T - \pi/2) + 1)}{2}$ or 0 otherwise, and t within the current interval.

In our simulations we let $M = 35$ requests, $T = 24$ seconds, and $k = 3$ seconds. Cache durations are set to 3 seconds, request timeouts to 3 seconds, and zones are resigned every 6 seconds. Our network consists of 40 nodes representing DNS addresses in the *com.* and *edu.* domains. DNS

queries are generated by 16 clients for `A` and `NS` records using the distribution outlined in [42] with failure rate of 11% (reflecting bogus requests [22]). Given that the majority of DNS requests made by a client are typically for local addresses [22] we let clients issue lookups for addresses within its immediate domain (e.g., .jhu.edu) with probability 0.5, for addresses one level higher (e.g., .edu) with probability 0.3 and for another top-level domain (e.g., .com) with probability 0.2.

EMPIRICAL RESULTS We evaluated the computational load (on a local resolver) and network bandwidth overhead for the DNSSEC approach, compared to standard DNS. We use DSA [12] for signatures, and RSA [12] as our asymmetric cipher. HMAC-MD5 [13] was use to generate our TSIGs. All public key operations use 768 bits of strength.
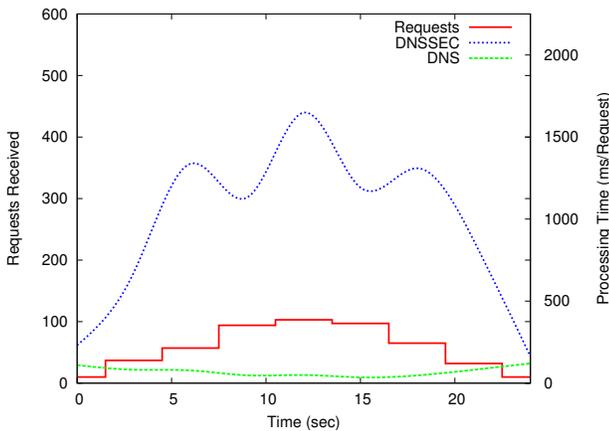
Figure 4. Processing time of a moderately loaded local-resolver when servicing DNS and DNSSEC requests.

Figure 4 illustrates the increase in delay experienced at a local resolver servicing 3 stub resolvers (averaged over 100 runs with outliers over two standard deviations away from the mean removed). Observe that the response time for standard DNS requests does not substantially increase even at the peak period of requests. However, a more dramatic effect is observed at the DNSSEC-enabled resolver. This increase is due to the overhead of verifying each response to an iterative query. The overhead reflects the processing involved in verifying signatures on returned RRs (including for KEY RRs additionally requested). The spike in delay near $k = 6, 12$ and 18 reflects the additional cost induced on this local resolver (as well as other authoritative name-servers) when resigning[7] their zone files $\approx$ every 6 seconds.

Table 1 depicts the overhead in traffic at key nodes in the system. The view from the local resolver is the same used in Figure 4. The increase in packets can be attributed to the request for KEY RRs. The increase in the size of responses is indicative of the SIG, NXT, and other RRs returned by DNSSEC (DNSSEC signatures are computed over RR Sets, thus the entire RR Sets must be returned for signature verification).

Similar reasoning applies to the other key nodes shown in Table 1. However, note that the Root's traffic increased by $\approx$ a factor of 3.3. This surge in traffic occurs because at peak periods of requests (e.g, between $k \in [10, 14]$), nearly all queries result in cache misses at the local resolver — as the local resolver attempts to verify the incoming requests, entries in its cache get invalidated due to the timeout expiring, resulting in queries being dispatched to the Root for subsequent requests. The problem is exacerbated until load on the local resolver recedes. Note that the increase in size of packets received at the Root does not increase because DNSSEC requests are essentially the same as DNS requests. However, the size of packets received at the COM and EDU nodes increases because these nodes also act as local resolvers for a small subset of nodes. Similarly, the mean size of the packets sent does not increase as drastically at these nodes because they also generate DNS and DNSSEC requests (whereas the Root just sends responses).

Again, we argue that the flexible and efficient architecture provided by Simnet was ideal for evaluating security centric protocols like DNSSEC. While one would expect that the public-key operations can have a significant impact at the local resolvers, the impact in traffic at the Root, for example, its not entirely obvious. In fact, we are unaware of any published empirical evaluation of DNSSEC, and so the results presented here (albeit preliminary) provide insights not easily found elsewhere. Moreover, since Simnet provides DNS as a base protocol, implementing the functionality required by DNSSEC only requires that one extend the `Resolver` plugin in DNS. The necessary cryptographic operations are provided by Simnets' `CryptoEngine` plugin which itself extends the primitives provided by Bouncy-Castle[8] .

6. Summary

We present Simnet, a discrete-event network simulator designed specifically for analyzing network security protocols. Its architecture is modular and extensible and allows for dynamic customization. While Simnet was not designed to replace the more widely used simulators, such as *ns*, we believe it offers a viable alternative especially for evaluating intricate protocols such as DNSSEC and Kerberos.

---

[7]In practice, a separate process not in the critical path may be assigned

the role of resigning the zones offline to negate these effects.

[8]See the Crypto API at `http://www.bouncycastle.org`

| | DNS | DNSSEC | increase (%) | increase (%) | increase (%) |
Node	Pkts	Pkts	Pkts	Bytes / Pkts Rcvd	Bytes / Pkts Sent
ROOT	124	541	336.2	1.9	505.8
EDU	685	872	23.9	140.1	327.2
COM	393	487	27.2	123.7	358.1
Local (3 Clients)	763	917	20.2	207.5	262.9

Table 1. Relative increase in DNS-related traffic received in DNSSEC compared to DNS.

Furthermore, we show that with negligible performance overhead, Simnet can be used to simulate non-trivial attacks on realistic topologies. As an example, we show how to model the impact of zero-day worms (like Nimda and Code Red II) on AS-level topologies representing a few million hosts. Additionally, we show that the simulator provides a powerful platform for analyzing the effectiveness of different DDoS defense strategies, and presented an analysis for a variant we called direct-Pushback. Lastly, we present a preliminary empirical results on analyzing the impact of proposed security extensions to DNS, particularly with respect to computational overhead for resolvers and induced network traffic overhead.

7. Availability

Simnet version 1.0 is freely available from `http://simnet.isi.jhu.edu`.

8. Acknowledgments

References

[1] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements, July 2004. draft-ietf-dnsext-dnssec-intro-11.

[2] G. Ateniese and S. Mangard. A new approach to DNS Security (DNSSEC). In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 86–95. ACM Press, 2001.

[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. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving simulation for network research. Technical Report 99-702, USC Computer Science Department, 1999.

[4] P. Ball. Introduction to discrete event simulation. In *2nd DYCOMANS workshop on Management and Control : Tools in Action*, pages 367–376, May 1996.

[5] S. M. Bellovin. Using the Domain Name System for System Break-Ins. *Proceedings of the Fifth Usenix Unix Security*, pages 199–208, 1995.

[6] W. Blackert, D. Gregg, A. Castner, E. Kyle, R. Hom, and R. Jokerst. Analyzing interaction between distributed denial of service attacks and mitigation technologies. In *DARPA Information Survivability Conference and Exposition*, page 26, May 2003.

[7] S. Canne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Technical (Winter)*, pages 259–270, 1993.

[8] CERT Coordination Center. See `http://www.cert.org/advisories/CA-2001-26.html`.

[9] Z. Chen, L. Gao, and K. Kwait. Modeling the spread of active worms. In *INFOCOM*, 2003.

[10] D. Eastlake. Domain name system security extensions (RFC 2535), 1999.

[11] eEye Digital Security. See `http://www.eeye.com/html/Research/Advisories/AL20010804.html`.

[12] Federal Information Processing Standards. Digitial Signature Standards (DSS) – FIPS 186, May 1994.

[13] Federal Information Processing Standards. The Keyed-Hash Message Authentication Code (HMAC) – FIPS 198, March 2002.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[15] D. Goldschlag, M. Reed, and P. Syverson. Hiding routing information. In R. Anderson, editor, *Information Hiding*, volume 1174 of *LNCS*, pages 137–150. Springer-Verlag, 1996.

[16] S. Halloway. *Component Development for the Java Platform*. Pearson Education, 2002.

[17] C. Hedrick. Routing information protocol (RFC 1058), 1998.

[18] N. Hu and P. Steenkiste. Evaluation and characterization of available bandwidth probing techniques. *IEEE Journal on Selected Areas in Communications*, 2003.

[19] B. Huffkaker, E. Nemeth, and K. Claffy. Otter: A general purpose network visualizatin tool. In *ISOC INET 99*, 1999.

[20] J. Ioannidis and S. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *Network and Distributed Systems Security Symposium*, February 2002.

[21] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Network and Distributed Systems Security Symposium*, 1999.

[22] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS Performance and the Effectiveness of Caching. *IEEE/ACM Transactions on Networking.*, 10(5):589–603, 2002.

[23] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, 1998.

[24] A. Lioy, F. Maino, M. Marian, and D. Mazzocchi. DNS security. In *Proceedings of the TERENA Networking Conference*, 2000.

[25] R. Manajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM Computer Communication Review*, 32(3), 2002.

[26] C. McDonald. A network specification language and execution environment for undergraduate teaching. In *ACM Computer Science Education Technical Symposium*, pages 25–34, March 1991.

[27] C. McDonald. Network simulation using user-level context switching. In *Australian UNIX Users' Group Conference '93*, pages 1–10, 1993.

[28] D. Meyer. University of Oregon Route Views Project. See http://www.routeviews.org.

[29] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM 03*, 2003.

[30] H. Muhammad and M. Barcellos. Simulating Group Communications Protocols Through an Object-Oriented Framework. In *Proceedings of the 35th Annual Simulation Symposium*, pages 143–151. IEEE, 2002.

[31] Opnet Modeler. See http://www.opnet.com.

[32] T. Peng, C. Leckie, and K. Ramamohanarao. Defending against distributed denial of service attacks using selective pushback. In *Ninth IEEE International Conference on Telecommunications*, June 2002.

[33] M. Pidd. *Computer Simulation in Management Science*. Wiley, 1992.

[34] S. Ross. *A First Course in Probability*. Prentice Hall, 1998.

[35] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for ip traceback. In *2000 ACM SIGCOMM Conference*, pages 295–306, 2000.

[36] C. Schuba, I. Krsul, M. Kuhn, E. Spafford, A. Sundaram, and D. Zamboni. Analysis of a denial of service attack on TCP. In *IEEE Symposium on Security and Privacy*, 1997.

[37] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Stayer. Hash-based ip traceback. In *SIGCOMM01*, 2001.

[38] D. Song and A. Perrig. Advanced and authenticated marking schemes for ip traceback. In *IEEE Infocomm*, 2001.

[39] SQL Slammer. See http://www.cert.org/advisories/CA-2003-04.html.

[40] J. Steiner, C. Neuman, and J. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Association Winter Conference 1988 Proceedings*, pages 191–202, February 1988.

[41] R. Tarjan. Data structures and network algorithms. *SIAM*, 1983.

[42] D. Wessels and M. Fomenkov. Wow, That's a Lot of Packets. *Workshop on Passive and Active Measurements*, 2002.

[43] P. Wolfe, C. Scott, and M. Erwin. *Virtual Private Networks*. O'Reilly, 1998.

[44] C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *9th ACM Conference on Computer and Communications Security*, pages 138–147, 2002.

[45] C. Zou, W. Gong, and D. Towsley. Worm propogation modeling and analysis under dynamic quarantine defense. In *ACM Workshop on Rapid Malcode*, pages 51–60, 2003.

A Appendix

EXAMPLE PLUGIN   As a simple example, and to better illustrate how Simnet's plugin architecture is typically used, we describe the steps required to implement a UDP Traceroute application in Simnet and in *ns*. Traceroute is a network tool that determines the path taken by packets in order to reach a destination.

To implement Traceroute as a Simnet Application, a user needs only to define a `Traceroute` class that extends the `Application` class and that implements two methods: `traceroute()`, which initiates a trace and serves as an interface between the UI and the Traceroute plugin; and `inBPF()`, which processes incoming ICMP packets. `traceroute()` registers a BPF for all incoming ICMP packets and proceeds to send probes with increasing TTLs. The `inBPF()` method receives ICMP packets and according to their type (time exceeded or port unreachable) either stops the trace or notifies `traceroute()` to continue probing.

Though the implementation of UDP Traceroute is straightforward in Simnet, we believe it is less so in *ns*. Because *ns* does not provide all the features of UDP and ICMP that are needed for Traceroute (in particular the generation of ICMP error packets), the user must implement them herself. This includes modifying the `TTL` class to return error messages when a packet's $TTL = 0$, and modifying the `UDP` agents to increment their packet's TTL field. Since the user can not take advantage of a builtin ICMP infrastructure, it becomes necessary to implement a Traceroute application that initiates traces *and* echoes packets (at the destination). Furthermore, a Tcl class must be implemented to interface with the application.

PERFORMANCE & SCALABILITY   In this section we provide a performance evaluation of the modeling of latency and bandwidth presented herein. We believe the results show that the implementation of latency and bandwidth modeling is efficient, and that the platform provided by

Simnet is ideally suited for the types of experiments presented in Section 5.

Link bandwidth and latency are modeled within the simulator by using three distinct queues (transmission, link, and receive), and by using four per frame delays (processing, queuing, transmission and propagation). Each link is responsible for updating the delays of each frame and moving them to and from the three queues. Each frame maintains a transmission time (i.e, the time when it will be moved onto the the link queue), and an arrival time (i.e, the time the frame is moved to the receive queue). In addition, each link records an *end of transmission time* (*eott*) which corresponds to the time at which the last frame on the transmission queue will be moved onto the link queue.

Therefore, when a node passes an outgoing frame, f, to its link, its transmission delay can be expressed as $td_f = \frac{\lambda_f}{c \times 8000}$ where td_f is the frame's transmission delay, λ_f is its size in bytes and c is the link's capacity (bandwidth) in bps. f's transmission time is given by $tt_f = \max(eott, now) + td_f$

where tt_f is the frame's transmission time and now denotes the current time. Similarly, f's arrival time is then $at_f = tt_f + latency$ where at_f is the frame's arrival time and $latency$ is the link's propagation delay in milliseconds. Once the times are calculated, the frame is enqueued on the transmission queue and the link's *eott* is updated to tt_f.

Unlike the transmission and propagation delays, a frame's queuing and processing delays are modeled implicitly. Given an outgoing frame f that has not yet been enqueued, its queuing delay can be calculated as $q_f = \max(eott - now, 0)$ where q_f is f's queuing delay. For reasons outside the scope of this paper we chose not to model processing delays explicitly in Simnet. However, if required, processing delays can simply be incorporated into to the latency of a link.

To estimate the processing overhead for modeling latency on a given topology we set a latency of $100ms$ across all links and calculate the transmission delay over a number of rounds. For each round of communication, a packet generator residing on a node sends a UDP packet with an 8-byte payload (comprising of a timestamp of when the packet was sent) to all other nodes. Each node's packet generator waits for a uniformly distributed delay, δ, in-between sending packets. Figure 5 presents the results for our experiment on a topology consisting of 192 nodes with $\delta \in [256, 512]$ ms – that is, the case where one packet is sent every 2 ms. Notice that 95% of the packets traversing exactly one hop incur an overhead of at-most 2 ms. Moreover, 95% of the packets traversing 7-hops incur a maximum overhead of only 37 ms – i.e., an overhead of 5%. In order to evaluate issues of scale, the number of nodes in the topology was increased while the total packets sent per second by the entire network remained constant. The results are described
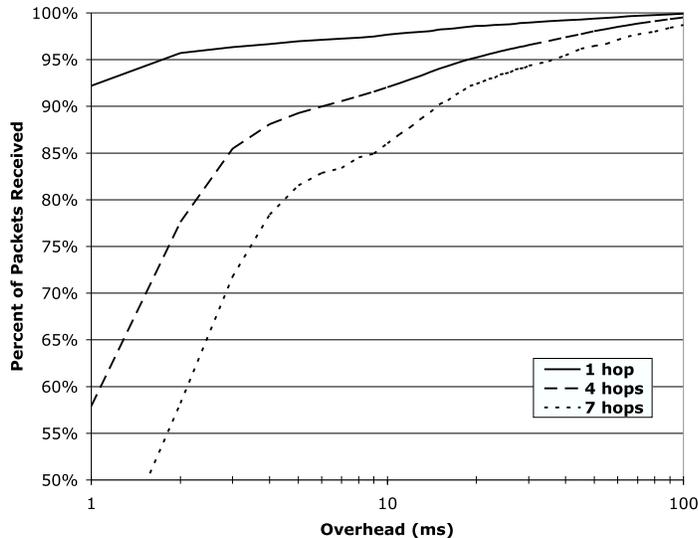
Figure 5. Latency modeling across an AS-level topology comprising of 192 nodes. For each of 10 rounds, each node sends a 56 byte packet to all other nodes. A total of 500 packets are sent by all nodes every second.

in Table 2. Notice that for all percentiles, the extra delay stays relatively constant as the topology grows in size. Table 3, on the other hand, depicts an experiment where the topology size is constant (192 nodes) but the total packets sent per second varies. In this case, when sending more than 600 packets per second the extra delay grows rapidly as the topology grows in size. We note, however, that 90% of the packets reach their destination with a reasonable delay. These results imply that in Simnet, the extra delays incurred for latency modeling are a function of the total packets sent per second and not of the topology size.

To validate our bandwidth modeling, we evaluate bandwidth allocation by measuring discrepancies in the arrival

Network Size	packets received				
	75%	90%	95%	97.5%	99%
57	2	5	11	20	39
95	2	3	8	16	30
151	2	3	10	21	42
192	2	4	14	32	66
252	2	4	12	23	47

Table 2. Extra delay, in ms, for specified percent of packets that travelled 3 hops in topologies of verifying sizes; total packets per second sent by the entire network was 500.
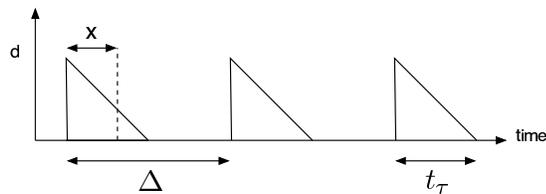
pkts / second	packets received				
	75%	90%	95%	97.5%	99%
200	1	3	9	18	32
400	2	4	14	26	55
600	2	7	18	33	57
800	4	17	39	76	111

Table 3. Extra delay, in ms, for specified percent of packets that travelled 3 hops in the network with 192 nodes, for varying numbers of total packets sent per second by the entire network.

at random (from $[1000, 2000]$ ms) between probes.

Empirical results averaged over 200 trials (not shown) resulted in executions where 90% of the time we achieved an estimation error of $\leq 20\%$. This error rate is related to our earlier simplifying assumption which we anticipate will be addressed in future work, for example by exploring techniques in [18]. Nonetheless, in Section 5.2 we showed that even under these constraints the simulator can still be used to evaluate non-trivial concepts such as Pushback.

times of packets traversing a link. This estimation can be achieved by sending a fixed size packet at a fixed rate (i.e, a packet train) across a link, interleaved with probe packets (containing their time of creation) sent at random intervals across that link. Assuming that probe packets do not delay the packets of the train, the difference between the average elapsed times for the packet train and the probe packets can then be used to estimate the available bandwidth on the link. Note that this difference, denoted as d, is equivalent to the average queuing delay of the probe packets (see Figure 6 for an illustration).

As in our previous discussion on links, let p be a packet, λ_p its size in bytes, t_p its transmission delay in seconds, q_p its queuing delay in seconds, c the capacity of the link in bits per second, and u the bandwidth being used in bits per second. Furthermore let train packets be denoted as τ and probe packets as π.

Bandwidth estimation was empirically evaluated as follows: the capacity, c, on a link between two nodes was set to $8960 = 56 \times 8 \times 20$ bps and the link's latency set to zero. We then set u to be 40% of the available bandwidth by transmitting an infinite series of 56-byte packets across the link. Packets in this series are sent once every 125 ms – i.e., we create a packet train at a rate of 8 packets per seconds. Next, the packet train is interleaved with successive 56-byte probe packets sent with a delay chosen uniformly

Figure 6. Illustration of variables used in bandwidth estimation.