

Simnet 0.9  
User Manual

September 7, 2004

# Contents

|           |  |           |
|-----------|--|-----------|
| <b>1</b>  | <b>Introduction</b>                                | <b>4</b>  |
| <b>2</b>  | <b>Current State</b>                               | <b>4</b>  |
| 2.1       | Directory Structure . . . . .                      | 4         |
| 2.2       | Known Issues . . . . .                             | 4         |
| <b>3</b>  | <b>Getting Started</b>                             | <b>4</b>  |
| <b>4</b>  | <b>The Simnet Architecture</b>                     | <b>5</b>  |
| <b>5</b>  | <b>System Architecture</b>                         | <b>6</b>  |
| 5.1       | Topology Parser . . . . .                          | 6         |
| 5.2       | The Simulator . . . . .                            | 6         |
| 5.3       | The User Interface . . . . .                       | 7         |
| <b>6</b>  | <b>Network Architecture</b>                        | <b>8</b>  |
| 6.1       | Nodes . . . . .                                    | 8         |
| 6.1.1     | Hosts . . . . .                                    | 8         |
| 6.1.2     | Router . . . . .                                   | 9         |
| 6.1.3     | NAT . . . . .                                      | 11        |
| 6.1.4     | IP Filters . . . . .                               | 11        |
| 6.1.5     | The BSD Packet Filter . . . . .                    | 12        |
| 6.2       | A note on Applications . . . . .                   | 12        |
| 6.3       | Links . . . . .                                    | 12        |
| 6.4       | Routing . . . . .                                  | 12        |
| 6.4.1     | Static routing . . . . .                           | 12        |
| 6.4.2     | Dynamic routing - RIP . . . . .                    | 12        |
| 6.4.3     | RIP architecture . . . . .                         | 13        |
| 6.5       | Protocols . . . . .                                | 14        |
| 6.5.1     | Packets . . . . .                                  | 14        |
| <b>7</b>  | <b>Customizing Simnet</b>                          | <b>14</b> |
| 7.1       | Plug In/Out Architecture . . . . .                 | 15        |
| 7.2       | Steps for the Plug Operation . . . . .             | 16        |
| 7.3       | Under the Hood . . . . .                           | 17        |
| <b>8</b>  | <b>Default Plugins</b>                             | <b>18</b> |
| 8.1       | Ping . . . . .                                     | 18        |
| 8.2       | Traceroute . . . . .                               | 18        |
| 8.3       | PacketGenerator . . . . .                          | 18        |
| 8.4       | PacketDump . . . . .                               | 18        |
| 8.5       | TCP . . . . .                                      | 19        |
| 8.5.1     | TCP implementation in Simnet . . . . .             | 19        |
| <b>9</b>  | <b>Individual Interface Windows (BETA)</b>         | <b>20</b> |
| <b>10</b> | <b>Developing Application Plugins</b>              | <b>20</b> |
| <b>11</b> | <b>Example Application: UDP Echo Client/Server</b> | <b>21</b> |
| 11.1      | The Client (see UDPEchoClient.java) . . . . .      | 21        |
| 11.2      | The Server (see UDPEchoServer.java) . . . . .      | 21        |
| <b>12</b> | <b>Bandwidth modeling</b>                          | <b>22</b> |
| 12.1      | Three Phase Modeling . . . . .                     | 22        |

|                                 |           |
|---------------------------------|-----------|
| <b>13 Performance</b>           | <b>22</b> |
| 13.1 RIP . . . . .              | 23        |
| 13.2 Latency modeling . . . . . | 24        |
| 13.3 Sample Tests . . . . .     | 26        |
| <br>                            |           |
| <b>14 Miscellaneous notes</b>   | <b>26</b> |
| Verboisity . . . . .            | 26        |
| “Fun” with Reflection . . . . . | 26        |
| Improved Scripting . . . . .    | 26        |
| Words of Wisdom . . . . .       | 26        |
| <br>                            |           |
| <b>A Commands</b>               | <b>27</b> |
| <br>                            |           |
| <b>B Otter File Format</b>      | <b>28</b> |

# 1 Introduction

Simnet is a network simulator written in Java. Simnet differs from existing network simulators in that it was designed specifically to analyze network centric attacks and countermeasures. To that end, we strive to provide a flexible and easily extendible framework for analyzing a myriad of attacks on realistic topologies. Every component of Simnet is modeled around the abstraction of *plugins*, and as such, we believe the architecture provides a framework that can be particularly useful for research and teaching.

The Simnet documentation includes two manuals:

1. **The Simnet User Manual** (this document), which provides an introduction to the simulator, describes its architecture and explains how to extend its various components.
2. **The Simnet Reference Manual** (currently not available), which details the Java classes that compose Simnet. The current javadocs serve as a quick reference to programmers, and is probably the quickest way to learn the *ins-and-outs* of Simnet.

## 2 Current State

The current version is **Simnet 0.9** developed in Java 1.4.x. To allow users to graphically display network topologies we use Otter [CAIDA], a network visualization tool developed by CAIDA. Since the most current version of Otter only compiles with Java 1.3, we provide a precompiled version in the `$$SIMNET/lib` directory. The most current version of the Simnet source code and documentation will be available from the Simnet homepage.<sup>1</sup>

### 2.1 Directory Structure

- `$$SIMNET`: The root directory of the Simnet source
- `$$SIMNET/doc`: Simnet documentation
- `$$SIMNET/lib`: Simnet, otter, bouncycastle, and gnu jars.
- `$$SIMNET/networks`: network topologies
- `$$SIMNET/scripts`: session scripts
- `$$SIMNET/src`: Simnet source code

### 2.2 Known Issues

Testing has been performed mainly on Linux<sup>TM</sup> and Mac OS X<sup>TM</sup> platforms. We are aware that there may be significant threading related problems that arise when used on Windows<sup>TM</sup> platforms. While some of the issues that were brought to our attention may have been resolved, we do NOT recommend that Simnet be run on Windows platforms.

Moreover, the functionality provided by the Interface Windows components (see Section 9), while particularly useful, is still in the **beta** stage, and has not been thoroughly tested — in particular, not all the default plugins automatically redirect *all* their output to these interface windows.

Please send email to [lucas@cs.jhu.edu](mailto:lucas@cs.jhu.edu) if you find any bugs.

## 3 Getting Started

Simnet requires Java 1.4.x for compilation. In order to compile the sources, you also need Ant (available from <http://ant.apache.org/>) — the java-based build tool. Once these requirements have been met, and the Simnet package has been downloaded, the following steps must be followed in order to install it:

---

<sup>1</sup><http://www.cs.jhu.edu/~fabian/simnet>

1. **tar xzvf Simnet-version.tar.gz**
2. **cd Simnet\_version**
3. **ant**

The first step in using Simnet is to acquire a network topology file. This file describes the topology of the network that will be simulated. Some sample topology files can be found in the `/$SIMNET/networks/` directory or on the Simnet home-page. The default file format used to describe networks topologies is the Otter file format (described in Appendix B). Once a topology file has been acquired, start Simnet by executing the script:

4. **./Simnet**, which starts the Simnet **User Interface**. NOTE: If you choose to run Simnet on a Windows platform, the file separators in this wrapper need to be changed accordingly.

From this point onwards, the following commands should be executed:

- **load file** - to build a simulation of the network described in **file**.
- **view** - (optional) display the network graphically
- **start all** - start the simulation. Prior to issuing the start command, all nodes remain in the *wait* state.
- **usp (updateshortestpath)** - update the routing tables on the shortest path (default) routers.

Now you can switch to a particular node in the network and execute node commands (*see Table 4*). For example, to ping node *CMU*, from node *JHU*:

1. **select JHU**
2. **ping CMU**

Simnet also provides the functionality of executing commands via script input. To do so, simply list each command on a separate line in a plain text file and execute the following command from the **User Interface**:

- **script file**, where **file** is the script file that contains all the commands.

Alternatively, scripts can be initiated directly from the command line using the `-s` flag. Command line options includes:

- **-l file** - load the network topology from **file**
- **-s file** - run the script in **file**
- **-v number** - set verbosity level to **number**

## 4 The Simnet Architecture

We divide Simnet's architecture by layers that span the high level components that encompass the system itself, to the low level components that are used to build the simulation. These design layers are structured as follows:

1. **System Architecture**. The highest level components of Simnet. This includes the **User Interface**, the simulation engine (subsequently referred to as the **Simulator**) and the **Topology Parser**. The System architecture is detailed in Section 5.
2. **Network Architecture**. The data structures used to represent the network internally. This includes the Link, Host, Router, Packet, LinkProcessor, and Routing classes. The **Network Architecture** is described in Section 6.
3. **Node architecture**. The data structures used to represent network Hosts and Routers. This includes the Link Processor, Transport, Application and IPFilter classes. The Node architecture is described in Section 6.1.

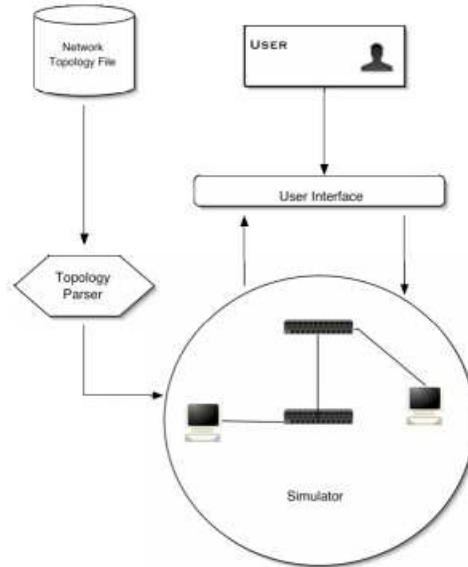


Figure 1: System Architecture

## 5 System Architecture

As mentioned in the previous section, the **System Architecture** includes the highest level components, namely:

1. **User Interface (UI)**: Interprets user commands and drives the simulation.
2. **Topology Parser**: Parses network topology files and instantiates the objects used to represent the network.
3. **Simulator**: Holds the objects used to represent the network.

The interaction between these components is illustrated in Figure 1. As mentioned earlier the first step in any Simnet session is loading a network topology. This can be done from the command line (by using the `-l` option) or from the Simnet **UI**. The topology is described in a plain ASCII file in the Otter file format (*see Appendix B*). The **Simulator** uses the **Topology Parser** to parse the topology file and generate a representation of the network. Once a network is generated, users can start and drive the simulation using the commands available from the **UI**.

### 5.1 Topology Parser

In order for Simnet to run a simulation, it must be given the topology of the network under consideration. Presently, the network topology is expressed in an Otter-like file format (Otter does not support link labels such as bandwidth and latency). This file is parsed by the **Topology Parser** and used to instantiate the data structures needed to represent the network. While the default **Topology Parser** understands the Otter file format it can be easily replaced by a custom parser that reads a different format (see Section 7 for details on customizing Simnet). From versions 0.9 and above the Otter topology parser is replaced by the SimpleINetParser which is capable of dealing with IP addresses, as well as link latencies and bandwidths. In order to view these topologies using the Otter graphical interface the, the view command will translate the currently loaded topology into one that Otter will read — i.e, link labels are ignored. Any changes made using the Otter interface will not be reflected in the Simulator. As such, Otter mainly serves as an interface for viewing topologies.

### 5.2 The Simulator

Simnet represents networks internally using the following classes: Host, Router, Link, LinkProcessor, RoutingTable and Packet. The details about how these classes are defined can be found in the **Simnet Reference**

**Manual.** For the purpose of the following discussions, it suffices to know that each instantiation of the Router, Host and Link classes has a unique ID (unique among the other objects in its class). Since these classes are used to represent a network, we refer to their instantiations simply as network objects. Once network objects have been instantiated by the **Topology Parser**, they are stored in the **Simulator** which places them in the following tables:

- **Host Table.** A hashtable that contains all **Host** objects keyed by their IP addresses.
- **Router Table.** A hashtable that contains all **Router** objects keyed by their IP addresses.
- **Link Table.** A hashtable that contains all **Link** objects keyed by their unique IDs.

In addition to holding network objects, the **Simulator** also provides objects with the following services:

- translation of a node's name to its unique IP Address (*and vice versa*).
- formatting of a nodes' output according to a standard output format.

### 5.3 The User Interface

The **UI** is a command line interpreter that allows users to interact with the simulation being performed. The commands provided by the **UI** can be grouped into three categories:

1. **System commands** that apply to Simnet as a whole (*see Table 3*).
2. **Node commands** that apply to a specific Node in the simulation (*see Table 4*).
3. **Broadcast commands** that apply to all Nodes in the simulation (*see Table 5*).

Users of Simnet should note that the **UI** behaves similarly to a Unix shell and provides a view of the network in a manner similar to that of Unix filesystem. For example, prior to performing a **node command**, the user must switch to a specific node. One can think of nodes as Unix directories and **node commands** as any directory tool (i.e. **ls**, **mv** or **cd**). In the same way that a user must **cd** to a directory in order to list its contents, in Simnet a user must switch to a specific node in order to run a **node command** on it. On the other hand **System** and **broadcast commands**, can be run from any node in the simulation. To illustrate the use of node commands, we present a simple, yet concrete, example of a Simnet session:

| Command  | Description  |
|--|--|
| <code>load networks/test.net</code>                              | loads the network topology described by the file <b>networks/test.net</b> into Simnet.   |
| <code>start all</code>   | starts the simulation  |
| <code>select JHU</code>  | switches to the node named JHU   |
| <code>ps</code>  | display list of current Application plugins  |
| <code>traceroute CMU</code>                                      | performs a traceroute from the current node (JHU) to CMU.  |
| <code>select BELL-LABS</code>                                    | switches to the node BELL-LABS   |
| <code>print</code>   | prints the details of a node, including its filtering rules  |
| <code>fw add 100 deny any from MIT to CMU any via in RIPE</code> | adds a filtering rule in this node that is numbered 100, denies any packet from node MIT to CMU and that comes in on the RIPE link . |

## 6 Network Architecture

In this section we describe how Simnet represents networks. At a high-level we are mainly interested in modeling nodes, links and packets. More specifically we are interested in the following components of a network: hosts, routers, links, Ethernet frames, and IP, TCP, UDP, ICMP packets and routing. To this end Simnet implements the following classes:

- **Node**: represents a network node (either a **Host** or **Router**)
- **Host**: extends the **Node** class and represents a network host
- **Router**: extends the **Node** class and represents a router
- **Link**: represents a data link
- **Frame**: represents a data link frame
- **Packet**: represents a network packet

As previously stated, every instantiation of the **Host**, **Router** and **Link** class has a unique ID that is assigned by the **Topology Parser** upon creation. These unique IDs are used by the Simulator to differentiate between objects in the same class and can be used from the User Interface to refer to specific **Nodes**. A **Node**'s unique ID is its IP address; a **Link**'s unique ID is used internally by Simnet.

### 6.1 Nodes

We make the following assumptions about routers and hosts: hosts are found at the edge of the network, have only one incoming and one outgoing link and cannot forward packets. Routers, on the other hand, can be found either at the core or at the edge of the network, can have multiple incoming and multiple outgoing links and can forward packets. Therefore, **Hosts** are a more restricted version of **Routers** and both classes extend the **Node** class from which they inherit most of their functionality.

#### 6.1.1 Hosts

As stated in the previous section, Simnet **Hosts** are found at the edge of the network and have only one incoming and one outgoing link that connects it to a **Router**. The overall behavior of a **Host** is simple: it can only receive and send packets. By default hosts understand subsets of the IP, TCP, ICMP and UDP protocols but can be extended to implement any other protocol (refer to Section 7 on customizing Simnet).

**Hosts** are composed of the following structures:

- **Link Processor**. Takes frames off the incoming **Link** and gives them to the Host which passes the enclosed IP Packet to the appropriate **IP Filter**.
- **Incoming IP Filters (Firewall)**. Either deny (drop) or accept (keep) incoming packets according to a user defined security policy.
- **Outgoing IP Filters (Firewall)**. Either deny (drop) or accept (keep) outgoing packets according to a user defined security policy.
- **Incoming BSD Packet Filters (BPF)**. Allow for user-level packet capture by directly forwarding copies of selected incoming packets.
- **Outgoing BSD Packet Filters (BPF)**. Allow for user-level packet capture by directly forwarding copies of selected outgoing packets.
- **Transports**. Objects that implement transport level protocols.
- **Applications**. Objects that implement user-level applications.

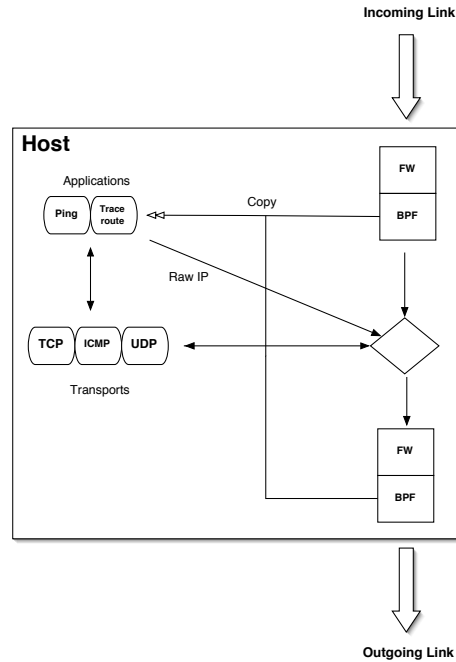


Figure 2: Host Architecture

The **Host** architecture is illustrated in Figure 2. Every **Host** has an associated thread that sleeps while the incoming **Link** is empty and stays awake while the **Link** is non-empty.

In the following discussion we outline the flow of an incoming **IP** packet  $P_{in}$  arriving at a **Host**  $H$ .  $P_{in}$  is encapsulated in a **Frame** and placed on  $H$ 's incoming **Link** by the upstream router. This action wakes the Host and its **Link Processor** proceeds to process the frame(s) on the **Link**.  $P_{in}$  is then extracted from its frame and given to the **IP Filter** which consults its security policy and makes a decision whether to drop or accept the packet. If the packet is accepted, it is further checked against the filter's **BPF** rules. Upon a match, a copy of the packet is forwarded directly to the associated object.

Next, the packet's transport layer protocol is determined and it is passed to the appropriate **Transport** which in turn strips its **IP** header off and processes the packet itself, or forwards the packet to the appropriate socket (for **TCP** or **UDP**), according to the destined port number.

Any outgoing packet,  $P_{out}$ , is first checked against the outgoing **IP Filter**'s policy and if accepted is checked against the filter's **BPF** rules. If it matches any **BPF** rule a copy of  $P_{out}$  is sent to the associated object. The packet is then encapsulated in a frame and enqueued on the outgoing link.

### 6.1.2 Router

**Routers** in Simnet 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 packets. By default **Routers** understand subsets of the **IP**, **TCP**, **ICMP** and **UDP** protocols and can be extended to understand any other transport or application level protocol.

**Routers** are composed of the following structures:

- **Link Processor.** Takes frames off the incoming **Links** and gives them to its Router which passes the enclosed **IP** Packet to the appropriate incoming **IP Filter**. The default **Link Processor** for every Router implements round-robin scheduling.
- **Multiple Incoming IP Filters (Firewall).** A unique **IP Filter** is associated with each incoming **Link**. The incoming **IP Filter** either denies (drops) or accepts (keeps) incoming **IP** packets on its associated **Link** according to a user defined rule set.

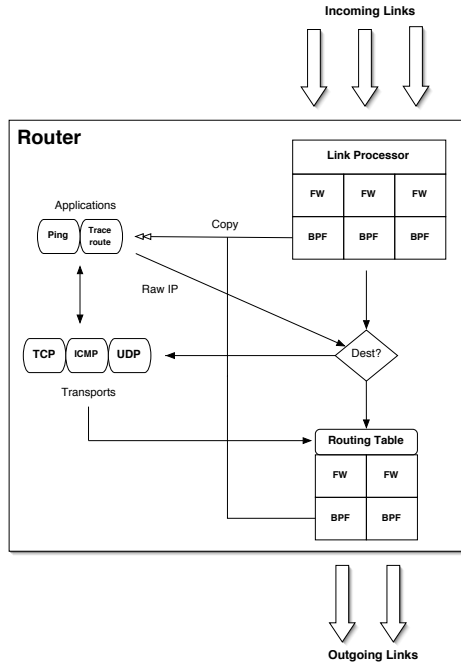


Figure 3: Router Architecture

- **Multiple Outgoing IP Filters (Firewall).** A unique **IP Filter** is associated with each outgoing **Link**. The outgoing **IP Filter** either denies (drops) or accepts (keeps) outgoing packets on its associated **Link** according to a user defined rule set.
- **Incoming BSD Packet Filters (BPF).** Allow for user-level packet capture by directly forwarding copies of selected incoming packets to the associated object(s).
- **Outgoing BSD Packet Filters (BPF).** Allow for user-level packet capture by directly forwarding copies of selected outgoing packets to the associated object(s).
- **Transports.** Objects that implement transport level protocols.
- **Applications.** Objects that implement user-level applications.

The Router architecture is illustrated in Figure 3. Every **Router** object has an associated Thread. This thread sleeps while all the Incoming **Links** are empty and stays awake while any of the incoming **Links** are non-empty.

We now outline the flow of an incoming **IP** packet  $P_{in}$ , arriving at a router  $R$ . We assume  $P_{in}$  is encapsulated in a frame and placed on one of the Router's incoming **Links** by an upstream Router. The frame's arrival notifies  $R$  of its presence, and the **Link Processor** proceeds to remove it from the Incoming Link.  $P_{in}$  is then extracted from its frame and given to the incoming **IP Filter** associated with the packet's source **Link**. The **IP Filter** consults its filtering rules and makes a decision whether or not to accept  $P_{in}$ . If  $P_{in}$  is accepted by the **IP Filter**, it is further checked against all its **BPF** rules. Upon a match, a copy of the packet is forwarded directly to the associated object. Next,  $P_{in}$ 's Time-to-Live ( $TTL$ ) field is decremented and if the resulting  $TTL < 1$ , an *ICMP Time Exceeded* packet is sent to  $P_{in}$ 's source. However, if its  $TTL \geq 1$ , the packet's destination is determined. If  $P_{in}$  is destined for this Router then it is passed on to the appropriate Transport level protocol, which can then forward it to an appropriate socket. If  $P_{in}$  is not destined for  $R$ , then  $R$  consults its Routing Table and determines which one of its upstream Nodes (connections) should be the next hop.

At this stage  $P_{in}$  is out-bound so we rename it  $P_{out}$ . It is first passed to the appropriate outgoing **IP Filter**, and if accepted by its policy it is then checked against its **BPF** rules. If a **BPF**-match is found, a copy of the packet is forwarded to the associated object. Finally  $R$  encapsulates  $P_{out}$  in a frame and puts on the appropriate out-bound link for  $P_{out}$ 's next-hop.

### 6.1.3 NAT

Network Address Translation (NAT) is a mechanism that allows a router to provide a private address space to a set of nodes. All packets that leave this private network will appear to originate from the router itself. Simnet provides NAT in the class `NATRouter`.

NATted nodes are assigned dynamic addresses by their specified `NATRouters` (see `networks/nat.net`) in the address space 10.0.0.0/8. Simnet provides up to 256 layers of NAT with 65536 hosts per layer. The first 8 bits of the IP address are always 10, the second 8 bits indicate the depth of the network and the last 16 uniquely identify a node within a network. To distinguish between different layers, a `NATRouter` first checks whether or not it is NATted before assigning new addresses. If so, and it has the address 10.x.y.z, it will assign addresses in the 10.x+1.0.0/16 address space.

Just as NAT is a “hack” on top of standard networking protocols and can break certain features, NAT in Simnet has several unavoidable characteristics. First of all, because the `hosts` and `routers` tables in `Simulator` are keyed by unique IP address, they are not able to hold NATted nodes (as NATted nodes from different private address spaces might have the same private IP address). To deal with this, all NATted nodes must be assigned a *unique* name and are stored in the hashtable `nattedNodes`. Thus, it is possible that performing a `lookup()` on a NATted IP address will fail. In some instances, it is possible to redirect such a lookup to the `NATRouter` responsible for the NATted node, and in such cases the lookup will succeed. When considering NATted nodes, one should perform lookup by node name.

Additionally, one can only replace (using the `plug` architecture) a `NATRouter` with another instance of a `NATRouter` (otherwise, the NATted nodes that rely on this router will not be able to communicate with other nodes).

### 6.1.4 IP Filters

A firewall is a network component that filters all incoming and outgoing traffic according to a given security policy. Firewalls can be hardware based, such as the Cisco PIX, or software based such as Checkpoint’s Firewall-1, OpenBSD’s Packet Filter [OBS] and FreeBSD’s IPFilter [FBS]. To enforce a security policy, a firewall is given a list of rules to enforce. Generally these rules are formed from a traffic pattern and an action to be performed on any matching traffic. Rules are checked sequentially and the action associated with the first matching rule is applied to traffic under consideration. Since the rules are applied sequentially, the order in which they are listed is very important.

Firewalls are grouped in two categories: IP Filters and Application Level Gateways (also called Application Level Proxies or Proxy Firewalls). At a high level, the difference between IP Filters and ALGs is that IP filters filter traffic according to TCP/IP headers while ALGs filter according to content. Consequently IP Filters must only understand the TCP/IP protocols, etc., while ALGs must understand every protocol they filter. This also means that IP Filters can be deployed in any network device, while ALGs require “a higher degree of sophistication”. In practice, Firewalls are further distinguished by the features they offer, such as stateful inspection and packet matching algorithms [GM01].

By default all firewalls in Simnet are IP Filters. Each Node has one filter associated with each incoming and outgoing link. This allows Nodes to filter traffic based not only on TCP/IP headers but also based on the previous and next hop (upstream and downstream Nodes). In Simnet firewall rules implemented as Rule objects and are kept in a list. A Rule’s position in the Rule list is also its unique ID. Upon receiving a packet, the filter passes it to each Rule in its list sequentially. If the packet matches a rule’s pattern, then that rule’s action (accept or deny) is performed on the packet. By default, Rules match packets based on the following fields:

- source IP address (with net mask)
- destination IP address (with net mask)
- source port
- destination port
- transport protocol
- incoming link

- outgoing link

Though it is trivial to add more fields to the IP Filter rules, in practice these are the only fields that are used. For more on extending the default Simnet IP Filter see Section (7).

### 6.1.5 The BSD Packet Filter

The BSD Packet Filter (BPF) [MJ93] is a component of the BSD TCP/IP suite that allows user level programs to have access to packets. It is at the heart of the Pcap library which itself is used by many Unix network security tools such as tcpdump and nmap.

In Unix, the BPF architecture works as follows: a user level program provides a regular expression that represents the profile of the packets it wishes receive from the TCP/IP stack. The regular expression is compiled into a filter that is passed on to the TCP/IP stack. TCP/IP will then pass a copy of every matching packet it receives to the application that requested it.

In Simnet this functionality is achieved by using the BPF class. An Application object instantiates a BPF object, sets the BPF object's filter (the syntax is the same as a IP Filter rule) and registers it with the Node. See Sections 8.2 and 10 for examples of using BPF's in Simnet.

## 6.2 A note on Applications

In Simnet, your Applications will typically communicate with each other using Sockets which can bind to any port. It is worthwhile to note that:

- Applications have associated process id's (`pid`) to enable multiple copies of a single application class to be loaded onto the same node at a time.
- A process id can prefix a command to specify which instance it should be directed to. For example `123.traceroute` will cause the application with process id 123 to (attempt to) execute the command `traceroute`.
- There is an explicit **BPFConsumer** interface that allows objects that are not applications to also have BFP rules.

## 6.3 Links

When representing a network, Simnet models the data link layer with the Link class. Links are implemented as queues and maintain pointers to two Nodes: a to Node and a from Node. A consequence of representing Links this way is that all Links are unidirectional. This means that only one Node (the from Node) is allowed to enqueue packets on the Link and only one Node (the to Node) is allowed to dequeue packets from the Link. While individual Link objects are unidirectional, Simnet can emulate bidirectional links between two Nodes by creating two Links of opposite direction between the Nodes. Additional information on modeling of links is given in Section 12.1.

## 6.4 Routing

### 6.4.1 Static routing

The default routing in Simnet is static. After loading a network topology, the **Simulator** instantiates a **Routing Table** object that generates a static routing table using Dijkstra's Shortest Path Algorithm (see [RT83]). Note that after the topology changes, the update shortest path (`usp`) command must be executed.

### 6.4.2 Dynamic routing - RIP

Dynamic routing in Simnet is achieved using the Routing Information Protocol (RIP). RIP is intended for use within a network connected by a number of gateways [H88]. It is a specialized distance vector protocol. All nodes participating in RIP maintain a table with an entry for every possible destination network. The entry contains

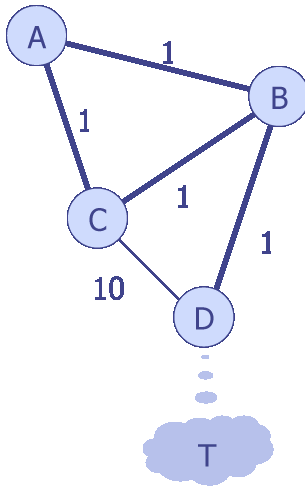


Figure 4: Counting to Infinity

the distance  $D$  to a network and the first gateway  $G$  on the route to that network. All nodes periodically send their routing table to their neighbors.

When a routing table update (*advertisement*) arrives from a neighbor  $G'$ , the cost of the network shared with  $G'$  is added to the distances in the advertisement. Each entry in this advertisement is compared to the current routing table. If the new distance  $D'$  is smaller than the existing value  $D$ , the new route is adopted. If  $G' = G$ , then  $D$  is updated to  $D'$ .

RIP is limited to networks where the longest path is 15 hops at most [H88, Page 4]. Furthermore, the protocol weighs all links equally. “Counting to infinity” is a major limitation of RIP. Suppose there exists a network as shown in Figure 4, and the link from B to D fails. Now B notices that its cost to D is infinity and A’s cost to D is 2. Since B is 1 hop away from A, it increases its cost to 3. A then raises its cost to D to 4 as it is 1 hop away from B. B then raises its cost to 5. This continues until A raises its cost to 12 and notices that the cost through C is 11, which is lower. Both A and B update their cost to D to 11.

Thus, by counting to infinity, we are able to update the path appropriately, but it takes more time than is desirable. This problem is solved by *split horizon with poisoned reverse* [H88, Page 14], i.e. advertising a distance of infinity to the router from which the route was learned. This technique solves the counting to infinity problem when only two routers are involved. Furthermore, to speed up convergence we use *triggered updates* i.e. a message is sent immediately whenever a change is made to the routing table.

When we get a message for a destination, we need to look up the most specific subnet. While this can be done by iterating over all the entries in the routing table, it will be done efficiently in a tree-like data structure. We use *tries*, a fast data structure for prefix querying [GT98, Page 566]. A trie  $T$  represents the set  $S$  by associating strings of  $S$  with paths from the root to the nodes of  $T$  [GT98, Page 568]. We used compressed tries in our implementation. It encompasses chains of single child nodes into individual edges and ensures that each internal node has a degree of two [GT98, Page 569].

### 6.4.3 RIP architecture

Nodes are either hosts or routers. We specify a *gateway* for every host and *fixed\_bits* that define a subnet for every router. Hosts forward all their network traffic to their gateway. We enhanced class *Rule* and its sub-class *BPFRule* with new constructors that can specify the subnet with a parameter for *fixed\_bits*.

Links can be added and removed during an active simulation. The *removeLink* method removes the link from the hashtable and rebuilds the link processor. These methods are in the *Node* class. They can be accessed by these new commands: *loadelement* for adding a link and *unload* for removing a link.

Enabling or disabling a link is also supported. This is useful when applications have filtering rules on an existing link. To accommodate this, flags are set to both ends of a link to represent its status. If both the

flags are *true*, then a link is considered enabled. However, if the flag for either end is *false*, a link is considered disabled. The new commands for enabling and disabling a link are *enablelink* and *disablelink*.

In addition to adding a link, a node can be added during an active simulation by the command *loadelement*. The arguments of this command are processed just like a line of the topology configuration file. A node can also be *isolated* or *integrated* by the commands *isolate* and *integrate*. A node *N* is isolated by disabling all the links connected to *N*. Specifically, the flag on the end connected to *N* is marked as *false*. A node is integrated by enabling all its links. Again, only the flags on the ends of links connected to the node are marked as *true*.

Simnet maintains a separate instance of ShortestPath at each node recording the distance from itself to all other nodes. However, since shortest path was designed for static routing, we provide a command, *updateshortestpath*, to rebuild the routing table whenever the underlying topology changes.

A routing table is initialized at each node to route appropriately in a dynamic network. A new class, *FilterRouter* extends *Router* and does automatic ingress and egress filtering. This is not plugged in by default.

RIP is implemented in the classes: *RIPEntry*, *RIPMessage* and *RIPRoutingTable*. The *RIPEntry* class has two final static variables – *EXPIRY* and *GARBAGE*. They have values of 180 seconds and 120 seconds respectively, as specified in [H88]. *RIPEntry* also contains fields for the destination, nextHop, distance, time\_stamp and fixed\_bits. It also has a method called getAge that returns the time since the entry was last modified, a resetTimeStamp method is to be called when the entry is modified, an expire method that is used when no update is received for the destination in the entry for 3 minutes.

The *RIPRoutingTable* class uses *RIPMessage* class to package and send the information in the routing table to its neighbours. As mention previously, the *RIPRoutingTable* uses tries to store the routing table. A class *Trie* implements the compressed tries and supports inserting, removing and finding an Object (*RIPEntry*) and iterating over all the Objects.

The *RIPRoutingTable* stores the routing table as routeTrie. It also has a timer in built. When a *RIPRoutingTable* is intialized, it initialized all its internal fields, and if it is plugged in on a router, it starts a *RouteD* application. *RouteD* is used to receive messages for *RIPRoutingTable* and pass them to *RIPRoutingTable*. When *RIPRoutingTable* receives a message, it functions as described in Section 6.4.2. The function lookup in *RIPRoutingTable* searches the trie to find the most appropriate nextHop, and return -1 if no route to host is found.

A *RIPTimerTask* inner class is used to cleanUpRoutingTable and send a *RIPMessage* to all the neighbors periodically. A new *RIPTimerTask* is scheduled each time the inner class is run. The cleanUpRoutingTable method finds *RIPEntries* that are stale (i.e. have been expired for 2 minutes or more) and removes them from the trie.

## 6.5 Protocols

Simnet does not implement all the TCP/IP protocols and it only implements subsets of IP, TCP, UDP and ICMP. The IP protocol is implemented directly in the **Node** class, while TCP, UDP and ICMP are implemented as **Transport** level plugins.

### 6.5.1 Packets

Currently Simnet implements the following packet types: IP, ICMP, TCP and UDP. Not all fields are included in the Simnet packets since most of them are not currently needed. Figure 5 illustrates the different packets and the fields they include. For the most up-to-date list of fields, look at the source code.

Packets may contain **transient** fields. These fields are just like any other fields, except that they are not used when computing a packet's size. An example of a transient field is a packet's **cached\_size**.

## 7 Customizing Simnet

To achieve modularity and flexibility we introduced a **Plugin** architecture into Simnet. This allows users to implement their own Simnet components, insert them into the Simulator and interact with these components within the Simnet **UI**. Using our architecture, if a user wishes to customize a specific component, she simply extends the appropriate classes and instructs Simnet where to load the new objects. Once loaded, any method

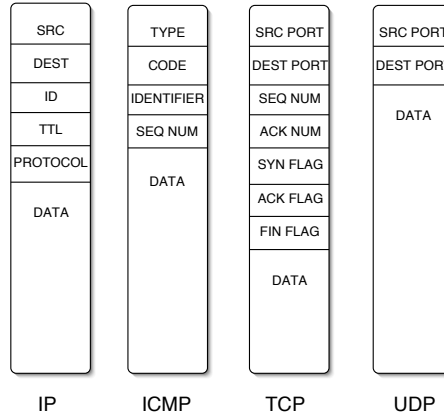


Figure 5: Simnet Packets

in the objects is accessible directly from the **UI**. As with commands we divide plugins into **Node** and **System** plugins depending on their function.

We refer to the components of Simnet that can be customized as **Plugin components**. After extending (and successfully compiling) one of the Plugin classes, a user can install the plugin by switching to the Node where it should be loaded, and executing the plug command to load (or remove) the object in question.

To interact with the new Application, the user can provide methods that will be directly accessible from the **UI**. For example, the user may wish to add a **chat** method to the **SecureChat** class that she can interact with via the **UI**<sup>2</sup>. This method takes a destination ID<sup>3</sup> as argument and processes all input from standard IN and outputs all incoming data to standard OUT. Once the new SecureChat class has been compiled, the user then inserts the object, and can interact with it as follows:

1. **select** *node<sub>i</sub>*
2. **plug in SecureChat** *pid*
3. **chat** *dest*

Steps 1 and 2 should be repeated for all nodes where the plugin is to be installed. Alternatively, if the plugin is to be installed in every Node the *all* option should be used in step 2 as follows: **plug in all SecureChat** *pid*.

In the above example, Simnet will load **SecureChat** and determine that it extends the **Application** class. The SecureChat object will then be instantiated and placed in node *i*'s **Application table**. Simnet will automatically find the **chat** method defined in the **SecureChat** plugin and invoke it with *dest* as its argument. The same concept can be applied to extend any plugin component.

## 7.1 Plug In/Out Architecture

A new plug in/out architecture has been introduced in Simnet version 0.8. A goal of the new architecture is to ensure that items being plugged out (removed) or replaced by a new plugin have a chance to clean up after themselves, and allow garbage collection to occur. There are two techniques that can be used to allow plugins to clean up (or be cleaned up), and several coding practices that need to be followed to allow for garbage collection and to take advantage of the new architecture.

Each type of plugin has a different way of storing its instances. Plugins are stored in one place (frequently a hash table on a node or in the Simulator itself) called their *primary storage location*. Each type of plugin has an associated subclass of **PluginType** to perform primary storage operations for it. The code in the **plug** function is easier to understand since all of the per-plugin type primary storage operations are delegated to several methods in the **PluginType** class.

<sup>2</sup>care should be taken not to use a method name that is already used by another **UI** command or plugin object.

<sup>3</sup>as an Integer and not an int. For more details see Section 7.3

| Plugin Type    | Arguments or Comments                       | Default Args |
|----------------|---|--------------|
| Application    | <pid>   [all*]                              |              |
| DropPolicy     | <destination>   all                         | all          |
| IPFilter       | (<other end>   all) [in   out   both]       | [all] both   |
| LinkProcessor  | <i>No Arguments</i>                         |              |
| Node           | <i>No Arguments</i>                         |              |
| RoutingTable   | <i>No Arguments</i>                         |              |
| TopologyParser | <i>No Arguments, plug out not supported</i> |              |
| Transport      | [all*]                                      |              |

Table 1: Plugin Types.

There are three methods in the `PluginType` class that handle primary storage operations. First, the `findInstance` method finds a plugin in primary storage and returns it (or null if no such object exists). Next, the `removeInstance` method removes a plugin from primary storage and returns if the operation was successful. Finally, the `initInstance` method will initialize an instance of a plugin and add it to primary storage, and returns if the operation was successful. A `PluginType` also contains three auxiliary methods. The first is `getType`, which returns the name of the (base) class of the plugin type that this `PluginType` supports. Closely related to `getType` is `getUsage`, which returns a line about how to use this type of plugin. Finally, the `isBatch` method is used to turn one call to plug into multiple calls to plug, for example, by iterating over all of a node’s applications, transports, links, etc.

The first technique is for an object to take care of its own cleanup. The `prePlugout` method in the `Pluggable` interface allows an object to take action before being removed based on the object replacing it. If it is being replaced with another object, it can modify some of its state to take into account the object will be replacing it (all of the fields that they have in common (from a common ancestor class) will be copied automatically, which can be thought of as “persistent storage”). If it being removed, it will have a chance to clean itself up before being plugged out. Note that classes should *only* implement this interface if they want to be replaced or removed. The `prePlugout` method allow a plug operation on a specific instance of a plugin to be aborted by returning false. The other technique is Plugout Notification (using the `PlugoutListener` interface), which allows an object to be notified when another object is getting plugged out. Like the `prePlugout` method, this technique also exposes an object’s replacement (which is null for plug out operations). This technique is more flexible then the first.

A plug operation can be executed at all nodes by specifying “all” to the plug command. Additionally, a plug operation can be performed at a node that is not equal to the currently selected node by prefixing the name of the node to do the operation on with a “@”. The complete syntax of the plug command is:

```
plug (in|out) [all|@<target node>] <class name> [<args>...]
```

A vertical bar means one of the enclosed choices, angle brackets are the names of arguments, square brackets are optional arguments, words are literals, and an asterisk after an argument means that it is only valid for plug out. The types of plugins and their supported arguments are given in Table 1. Note that when plugging out, the base class name (Application, Transport, DropPolicy, etc.) can be used if the arguments identify the instance to remove. With the exception of Transports (when all is not used), this “shortcut” can always be used.

## 7.2 Steps for the Plug Operation

1. The call to `plug` is parsed. First, a single string will be broken up into an array of arguments. Next, the arguments are parsed to determine if the operation is plug-in or plug-out, which class the operation is on and the arguments to pass it. The command will be executed on the currently selected node unless remote invocation (“@”) or run on all nodes are requested. The public wrapper to the internal plug implementation will pause (`stateful_kill`) the node the command is running on and will unpause (`stateful_start`) the node after the operation completes, if this is not a recursive (batch delegated) operation. Note that there are public versions of `plug` that for each step of parsing described.

2. The `PluginType` responsible for handling the specified class will be looked up<sup>4</sup>. If the appropriate `PluginType` cannot be found, the operation fails.
3. The `PluginType`'s `isBatch` method is called. That method can perform more plug operations (for example, by enumerating over a node's `Applications`, `Transports`, `Links`, etc.) and then return true to indicate that the operation was batch delegated and no more work needs to be done on the original operation. Otherwise, that method returns false, and normal processing continues.
4. The old instance (`old`) of the object involved in this operation will be looked up by using the plugin type's `findInstance` method. If no old instance is found, and this is a plug out operation, the operation (trivially) succeeds, although a failure code is returned.
5. If `old` was found, a check is performed to see that it implements `Pluggable`. **If the interface is not implemented, the operation fails.**
6. If a plug-in is being performed, a new instance (`new`) is created. The operation fails if an instance cannot be created, for example, if the class to plug in is abstract or does not have a public no argument constructor. (If a plug out is being performed, `new` will be null.)
7. If `old` was found:
  - (a) `old`'s `prePlugout` method is called, and `new` is passed to it. This allows the object to transfer some of its state to the new object, or clean up after itself.
  - (b) If `prePlugout` returns false (indicating that for some reason, `old` cannot be taken out of the simulator at the moment), the operation fails.
  - (c) If `new` is not null (i.e.: this is a plug-in), an ancestor copy is performed from `old` to `new`.
  - (d) The plugin type's `removeInstance` method is called to remove the instance from primary storage. (It returns an error if nothing exists to be removed.) If that function returns false, the operation fails.
8. The plugin type's `initInstance` method is called. That method stores the object in its primary storage and call the plugin-type specific "initialize" method (strictly speaking, the name of the method does not have to be initialize). An error is returned if an object already exists in that location. *To take full advantage of replaceable objects and persistent storage, most initialization code should be done by static initializers or in a no-argument constructor<sup>5</sup>. By the time an object is in its initialize method, some fields may have been copied over from what the object is replacing, so the initialize method shouldn't overwrite fields blindly.*
9. The plugout listeners for `old` get notified that the new object, `new` is replacing it. At this point, `old` can no longer be used. If a plugout registration or unregistration is done on `old`, a `ConcurrentModificationException` may be thrown. Plugout listeners can be used to transfer objects automatically, as is done for `Rules/BPFs` and `DatagramSockets`.
10. Strictly for debugging purposes, garbage collection is scheduled to run a few seconds later (through a `TimerTask`).

### 7.3 Under the Hood

To allow for dynamically loaded and customized **UI** commands, the **UI** works as follows: if it receives a command that is not recognized and the context has not been switched to a particular node, the class-loader checks whether the currently loaded **TopologyParser** or **RoutingTable** defines a method that matches the unknown command. If the context has been switched, then it checks if any of the **Applications** or **Transports** in the current **Node** match. If it cannot find a method matching the given command, an error is returned.

Since Java only allows Objects as arguments to methods that are invoked dynamically, the arguments of any method that defines a command must all be Objects. In particular this means that to implement a customized command that takes IDs as arguments in the **UI**, the method that implements it in a Plugin must take **Integer** objects as arguments and not **ints**.

<sup>4</sup>PluginTypes are registered in Simnet's main method by calling the Simulator's `registerPluginType` method.

<sup>5</sup>Note that plugins should *not* start threads in their constructors. A plugin is not really plugged in until the initialize method is called.

## 8 Default Plugins

In this section we briefly describe the plugins that Simnet provides by default. Table 6 summarizes these plugins and the commands they offer through the **UI**.

### 8.1 Ping

**Ping** is a network utility used to verify the status of a host. By default, upon receiving an *ICMP Echo Request* packet every standard **ICMP** implementation must return an *ICMP Echo Reply*. To determine whether a host is reachable or not the **Ping** utility issues an *ICMP Echo Request* to a host and waits for a reply. If the waiting time exceeds a set limit, then it assumes the host is unreachable. **Ping** can also be used to get an estimate on the round trip time to the destination.

In practice **Ping** can be implemented in many ways, but here we concern ourselves with the traditional **ICMP Ping**. In Simnet **Ping** is implemented as an **Application** level plugin. It starts by registering a **BPF** rule matching all incoming *ICMP* packets. Then it builds an *ICMP echo request* packet that it sends to the destination. Finally it waits for an incoming *ICMP Echo Reply* that encapsulates the original request packet. The Ping plugin provides a **ping** command that takes a destination as argument.

### 8.2 Traceroute

**Traceroute** is a network tool that determines the path taken by packets in order to reach a destination. There are two implementations of Traceroute, one based on **ICMP** and another based on **UDP**. Here we will only concern ourselves with the latter.

**UDP** traceroute works as follows: it starts out by registering a **BPF** rule matching any **ICMP** packet destined for the host where it resides. It then sends a **UDP** packet to its destination with a **TTL** = 1 and a random destination port higher than 1024 (**UDP** ports in this range are very rarely used). When the packet reaches the next hop, that host's **ICMP** implementation will cause it to return a *ICMP Time Exceeded* packet. Due to the **BPF** rules it installed, this packet will be forwarded to the **Traceroute** application which will use it to determine the IP of the first hop. **Traceroute** will then send a second packet with a **TTL** = 2 to determine the second hop and so on and so forth. Finally when it sends a packet whose **TTL** allows it to reach the destination, the destination will return a *ICMP Port Unreachable* packet, which **Traceroute** interprets as having reached its destination.

By default, Simnet provides every host with an application level **UDP** traceroute implementation that uses the Simnet **raw socket** and **BPF** functionality. The **Traceroute** functionality is wrapped in a method named **traceroute** that takes either a destination ID or Name as argument. This provides the **UI** with a **traceroute** command that can be used to run the **Traceroute** application.

### 8.3 PacketGenerator

The **Packet generator** plugin is an application level plugin that provides basic DoS attacks. We provide a **flood** command which implements a randomized packet flood where the initiator sends a specified number of packets with a specified distribution of characteristics. Using this command, a user can specify the total number of packets to be sent, and the percentage of those packets that should have a random source, a random destination and a random protocol. See `PacketGenerator.java` for more details. An example is given in `scripts/filter.script`.

### 8.4 PacketDump

While each node keeps track of how many packets it sends and receives (and forwards, in the case of a Router) often more detailed packet information is needed. **PacketDump** is an Application level plugin that enables users to log detailed information about all incoming packets that are destined to a specific node. Once the plugin is loaded into a Node, logging is initiated from the **UI** using the **dump** command and stopped using the **stopdump** command. **dump** takes a source and a destination node as arguments. The packet header information is output to a file named after the node. For example, to log all packets at node JHU that are destined to CMU we execute the following commands from the **UI**:

1. **select JHU**
2. **plug in PacketDump**
3. **dump any CMU**
4. **stopdump**

By default the packet header information will be output to the file *node.log* (**JHU.log** in the above example).<sup>6</sup>

The format of the log file is similar to a **tcpdump** log where each logged packet generates a single line that starts with a timestamp followed by its header information. The timestamp is in the form *hh : mm : ss : mm* and the header format is protocol dependent:

- TCP: *src.port > dest.port : sequence ack data=object*  
where **src** and **dest** are the source and destination Nodes and **port** is their TCP port. **Sequence** is the packet's sequence number, **ack** is its acknowledgement number and **window** is its window size.
- UDP: *src.port > dest.port : udp data=object*  
where **src** and **dest** are the source and destination Nodes and **port** is their UDP port.
- ICMP: *src > dest : icmp type code*  
where **src** and **dest** are the source and destination Nodes. **type** and **code** are the packet's ICMP Type and Code.

We add an **in** or **out** stamp to each line to indicate whether the packet was logged when it arrived on an incoming or an outgoing link. The size of each packet is also included in the log.

## 8.5 TCP

Transmission Control Protocol (TCP) sits on top of the IP layer and below the application layer (*see Figure 6*). and provides applications with a reliable stream service. A TCP implementation establishes a connection with a remote host and guarantees the delivery of each packet in the order it was sent. TCP offers the following services (only some of these are implemented in Simnet):

1. Connections. A connection is a virtual circuit between two applications that is defined by pair of tuples of the form *(host, port)*. TCP is responsible for establishing connections (using a three-way handshake) and for the reliable delivery of every packet sent through the connection. TCP uses a two-way handshake to tear down a connection,
2. Reliable and ordered data transfer. TCP guarantees the safe delivery of all packets by returning an acknowledgment for every packet received. If a packet is not acknowledged then it is assumed lost and it is resent. To guarantee that the data is received in the order it was sent, TCP sends a packet only after the previous one has been acknowledged.
3. Congestion control. TCP throttles the amount of data it sends according to the current congestion of the network.

For a detailed explanation of TCP see [RFC793] (we also recommend [RS94, WS95, DC95]).

### 8.5.1 TCP implementation in Simnet

Simnet's implementation of TCP is defined in the files `src/TCP.java`, `src/Socket.java`, and `src/ServerSocket.java`. While the current implementation reflects real TCP with enough accuracy to be used in meaningful simulations, it does not support fast retransmit or flow/congestion control. Additionally, sequence numbers do not wrap around (such a wrap will cause unexpected behavior), and only increase by 1. Furthermore RST packets do not contain acknowledgement or sequence numbers.

---

<sup>6</sup>The exact format of the current tcpdump logs differs slightly from that discussed here. See the logs generated, for example, by `scripts/tcp.script`

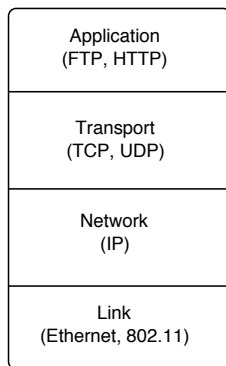


Figure 6: TCP/IP Layers

Reliable transmission is achieved through the use of `TimerTasks` that resend packets after `getRetransmitTimeout()` ms. Note that in order to have independence between the original packet that was sent (and possibly lost) and the retransmitted version of the packet, TCP must `clone()` the original packet before it is sent. Thus, all data that is sent over TCP in Simnet *must* be cloneable. While TCP handles `String` objects internally with the `StringWrapper` class, Applications that wish to send other uncloneable data types (e.g., `Integer` and `byte []`) must provide mechanisms to modify these types so they can be cloned (see `src/ByteArrayWrapper.java`).

## 9 Individual Interface Windows (BETA)

While all interaction with Simnet can be conducted through the `UI`, we provide additional individual interfaces for each Node and Application. These individual windows allow users to confine the scope of their input and output to an individual node or application.

When executed from a Node, the `window` command will open a window that is associated with the currently selected Node. The window consists of an input field and an output area. The input window takes commands destined for its corresponding Node (as well as broadcast and system commands), and output area is used to display all of the corresponding Node's output. For example to open a window for the Node JHU one would execute the following commands:

- `select JHU`
- `window`

Now any node command entered in the window's input field will be executed in the context of JHU.

Applications are also provided with `IIWs`. To open an Application `IIW` the window command must be used followed by the Application's pid <sup>7</sup>:

- `select JHU`
- `window pid`

These commands will open a window associated with the application plugin with process id `pid`. The only commands available from this window are the commands provided by the associated Application and only a single window can be opened for each Application. Multiple window can be opened at the same time as long as they are associated with different Nodes or Applications.

## 10 Developing Application Plugins

In this section we briefly review some steps needed to implement Application plugins. Every plugin must extend the Application class (which is an abstract class). This allows Simnet to recognize the class file as an Application

<sup>7</sup>A list of application process ID's on the current node can be obtained from the `ps` command.

plugin. If the Application needs to access raw IP packets directly (see Traceroute and PacketDump), it must implement the **inBPF** method from the **BFPConsumer** interface. Any initialization an application must perform should be done in the **customInit** method.

All Applications inherit the following methods (or has access to them through its node):

- **node.getTransport**: takes a Transport ID as argument and returns the corresponding Transport level plugin. The Application can then use the methods offered by the Transport plugin to send data according to the corresponding protocol.
- **printout**: takes a category string and verbosity level and a string as arguments and outputs the given string to the **UI** (or an open **IIW**) if the system verbosity level is equal to or higher than the specified level.
- **rawOut**: takes an IP packet as argument and passes it directly to the IP stack. This allows the Application to construct its own IP packets and send them as is.
- **window**: opens a new **IIW** associated with this Application.

## 11 Example Application: UDP Echo Client/Server

### 11.1 The Client (see UDPEchoClient.java)

The client is designed to have a single socket bound to an unspecified local port (the next free port). The run method of the client application reads from the socket and prints out any messages received. It will keep running until the socket is closed, or it is told to stop running (**keep\_going** is set to false, see more details later in this section).

The client maintains two variables that can be copied to its replacement. The first one is the default port number (**defaultPort**) to use. The other one is the DatagramSocket that is used to send and receive messages. Both of these variables use static initializers. Note that the other instance variable, **keep\_going** is declared as **transient**. This declaration prevents it from being automatically copied into a replacement.

The application specific initialization code is done in the **customInit** method. If it did not replace another client, and keep that client's open socket, it creates a new DatagramSocket and binds it to the next available local port. The socket is created with the client as its consumer (second argument to the constructor). This allows the socket to register a plugout notification with the simulator so that the socket can be informed when the client is going to be plugged out (whether it is replaced or not).

The client can be both replaced and removed since it implements the **Plugable** interface. In either case, the client's **prePlugout** method is called, which stop the client's thread from running by setting **keep\_going** to false and interrupting the thread. If the client is being replaced by a different application (not a UDPEchoClient), the socket is explicitly closed (the explicit close is needed because without it, the socket would transfer its consumer to be the replacement, but the replacement would not have access to the socket since it wasn't ancestor copied). If the socket was not closed, it will be transferred to the replacement when it receives the plugout notification for the client.

### 11.2 The Server (see UDPEchoServer.java)

The server allows multiple ports to be opened up, with a different thread listening on each port (with separate sockets). The list of ports and the associated threads are maintained in a hash table which can be manipulated using the open and close commands. Like the client, the server can be both replaced and removed. However, the cleanup on the server is more complicated due to using multiple threads.

A design decision was made to not keep sockets open across replacements, and as a result, the **prePlugout** method ignores its replacement. When being removed, the only work that **prePlugout** does is to clear the sessions hash table – it won't do our replacement any good to know about threads that are about to stop listening. Clearing the hashtable probably wasn't necessary since a new hashtable will be created during the new server's initialization.

No other cleanup is necessary because the threads clean themselves up: when the thread is constructed, it registers for a plugout notification when the server is going to be plugged out. As a result, when the thread's

plugoutNotification method gets called, it can close the socket (which will cause it to stop and be able to get garbage collected). When a thread finishes running, it will remove itself from the server's sessions hash table. Note that the socket in each thread will not be able to clean itself: the thread itself does not get plugged out, and will never be the target of a plugout notification (null is actually passed as the consumer to avoid a warning message).

## 12 Bandwidth modeling

### 12.1 Three Phase Modeling

The three phase approach to modeling bandwidth simulates the four types of delays that real packets experience: processing, queueing, transmission, and propagation. The node processing delay is explicitly taken to be 0, and any real time that is spent processing packets will be counted against the other three delays<sup>8</sup>. The propagation delay is exactly a link's latency.

To model transmission and queuing delays, each link maintains a "end of transmission time" variable called `eott`. When it is -1, the link has no packets to transmit, otherwise its value is the time at which the last packet in the transmission queue will finish being sent. The following equations are used to calculate these delays:

$$\text{transmission delay} = \quad \quad \quad td \quad \quad = \frac{\text{size}}{\text{bandwidth}} \quad \quad (1)$$

$$\text{transmission time} = \quad \text{frame.trans\_time} \quad = \max(\text{eott}, \text{now}) + td \quad \quad (2)$$

$$\text{arrival time} = \quad \text{frame.arrive\_time} \quad = \text{frame.trans\_time} + \text{latency} \quad \quad (3)$$

where `size` is the size of the packet (including the frame it is in) (in bytes), `bandwidth` is the bandwidth of the link (converted to bytes per millisecond), `now` is the current time (from `System.currentTimeMillis()`), and `latency` is the latency of the link (in ms). If the link's bandwidth is infinite (a value of 0), the transmission delay is 0. After the frame's times are calculated, the link's `eott` is updated to equal `frame.trans_time`.

It is important to understand that the times stored in a frame represent the earliest time at which a packet may leave the transmission queue or arrive at its destination. The experiments described in Section 13 detail how soon after a packet is expected to arrive it actually does. Packets are moved from one queue to the next (transmission to link to receive) by the `modelBandwidth` function. That function is called directly whenever a packet is `nqed`, and is called through a `TimerTask` when a packet is ready to move. There one global timer that is shared by all links, and each link has at most one `TimerTask`.

When a link has a finite sized transmission queue, it is possible that the queue can be full, and packets may need to be dropped. A new plugin type has been added to allow packets to be dropped even if the transmission queue has room for them. This plugin type is called a `DropPolicy`. Any packets that a link's `DropPolicy` decides to drop will not be included in bandwidth modeling. That is, packets that are *dropped early* will not change a link's `eott` and contribute to the queueing delay of packets that immediately follow it. This early dropping is in contrast to the links ability to randomly lose packets. Those packets are lost after they have gone through the link queue, and have already changed a link's `eott` and contributed to the queueing delay of the packets that immediately follow it.

## 13 Performance

This section is presented as a reference that provides some results pertaining to performance you can expect on topologies used in class. In particular, the section on latency modeling (section 13.2), provides some insight into the delays in packet delivery you should expect. Large deviations from these expected times should alert you to possible problems in your extensions. In particular, with zero latency and infinite bandwidth across a link, you can expect the majority of your packets to arrive at their intended destination in less than 1 second.

---

<sup>8</sup>To model no processing delays, the time that a packet will arrive is the time *before* processing begins plus the appropriate delays. Since it does take some real time to process a packet, that processing time can be part of a packets staleness.

| Benchmark  | Total Time (sec) |           |              | Lookup Time (ms) |              |
|------------|------------------|-----------|--------------|------------------|--------------|
|            | Mean             | Std. Dev. | > sp-orig By | Mean             | > sp-orig By |
| sp-orig    | 5.979            | 0.174     | N/A          | 0.611            | N/A          |
| sp-ip      | 6.933            | 0.193     | 0.954        | 0.708            | 0.097        |
| rip-de     | 7.416            | 0.200     | 1.437        | 0.757            | 0.146        |
| rip-de-tri | 7.529            | 0.183     | 1.550        | 0.769            | 0.158        |
| rip-pl     | 7.939            | 0.266     | 1.69         | 0.811            | 0.200        |

Table 2: Benchmark Results

## 13.1 RIP

The following reflect tests performed by Darren Davis and Ashima Munjal based on their extensions to Simnet V0.7 to incorporate dynamic routing. These extensions have been adopted in Simnet, starting with version V0.8.

In contrast to Shortest Path Routing, RIP Routing is an ongoing process with a more complex lookup mechanism. As a result, RIP routing has some overhead above and beyond that of Shortest Path. To measure this overhead, tests comparing routing in Shortest Path to routing in RIP were performed. Since Shortest Path does not involve changing topologies, the tests were conducted after RIP’s routing tables had converged. A separate set of tests were performed to measure RIP convergence.

To facilitate automated benchmarking in Simnet, new commands were added to the user interface. These commands allow a named timer to be initialized to the current time (*settimer*), and for the time elapsed since initialization to be printed (*getElapsedTime*). Our benchmark initialized a timer, ran a series of traceroutes, and printed the time elapsed. A traceroute was done between every pair of nodes in the topology comprising 17 nodes. The traceroute command was modified to avoid tracerouting from a node to itself, as there is no support for loopback interfaces in Simnet. This modification resulted in a total of  $17 \times 16 = 272$  traceroutes in the benchmark. For each benchmark, 100 executions were performed, and we report the mean and the sample standard deviation. These tests were run on a PowerBook G4 800MHz, with 512 MB of RAM<sup>9</sup>, with output redirected to a file. The benchmarks were executed under the following 5 different sets of conditions:

**sp-orig** The original version of Simnet-0.7 as a baseline.

**sp-ip** Baseline with modified *ShortestPath* (to handle IP addresses) as the default routing table.

**rip-de** Simnet with RIP (routing table implemented with a hash table) used as the default routing table.

**rip-de-tri** Simnet with RIP (routing table implemented with a compressed trie) used as the default routing table.

**rip-pl** Simnet with *ShortestPath* as the default routing table, with RIP (routing table implemented with a hash table) plugged in.

The mean and sample standard deviation for each benchmark are given in Table 2. There are a total of 17 nodes in the topology, for a total of  $17 \times 16 = 272$  traceroutes per test, since a node does not traceroute to itself. For the topology, the average number of lookups per traceroute is  $\approx 36$  (corresponding to an average path length of 3). Thus, each benchmark has a total of  $\approx 9792$  lookups. The mean time per lookup, as well as the mean time per lookup beyond the baseline test (sp-orig) are also given in Table 2.

Since there are many factors that can affect the observed value of a benchmark’s time, the observed mean time is a random variable. Thus, we need to test the hypothesis that the mean times for different benchmarks are the same. The statistical method to test this hypothesis is to calculate a *t*-statistic<sup>10</sup> for each pair of tests and determine if its absolute value is too large [R95, Section 11.2].

Let  $t_\alpha$  be the  $\alpha$  percentile of the distribution of  $t$ . Under the hypothesis that the means of two tests are the same, the probability that their observed *t*-statistic is greater than  $t_\alpha$  is  $1 - \alpha$ . A table of percentiles of the *t*

<sup>9</sup>During the test, many other applications were running, and there was not enough physical memory for all applications to run in.

<sup>10</sup>In our case, the sample size is 100, and let tests  $X$  and  $Y$  have means  $\bar{X}$  and  $\bar{Y}$ , and standard deviations  $S_X$  and  $S_Y$ . Define  $s_p^2 = (S_X^2 + S_Y^2)/2$ . Then  $t = (\bar{X} - \bar{Y})/(s_p \sqrt{1/50})$  has a *t* distribution with 198 degrees of freedom.

distribution [R95, Appendix B, Table 4] was used for the following analysis. If  $|t| > t_{0.95} \approx 1.65$ , then there is only a 5% chance that such a discrepancy happened by chance, and so the hypothesis of equal benchmark times will be rejected.

All combinations of pairs of tests that did not involve any of the rip-\* tests had  $t$ -statistics greater than  $t_{0.95}$ . The  $t$ -statistic between rip-de and rip-pl was just under  $t_{0.95}$ , and the  $t$ -statistic between rip-de-tri and rip-pl is  $\approx t_{0.90}$ . The  $t$ -statistic between rip-de and rip-de-tri is  $\in [t_{0.60}, t_{0.70}]$ . Thus, approximately 1 out of every 3 runs of this pair of benchmarks would be expected to have a  $t$ -statistic this great, so these two tests may actually have the same running time. However, all of the other combinations of tests have  $t$ -statistics sufficiently great to reject the hypothesis of equal benchmark times.

We suspect that rip-de and rip-de-tri run in close to the same amount of time because there are only a very small number of routers (9) in the topology. Thus, the difference between iterating over a hash table and searching a trie was negligible. However, with a larger number of routers, one would expect rip-de-tri to be faster than rip-de.

## 13.2 Latency modeling

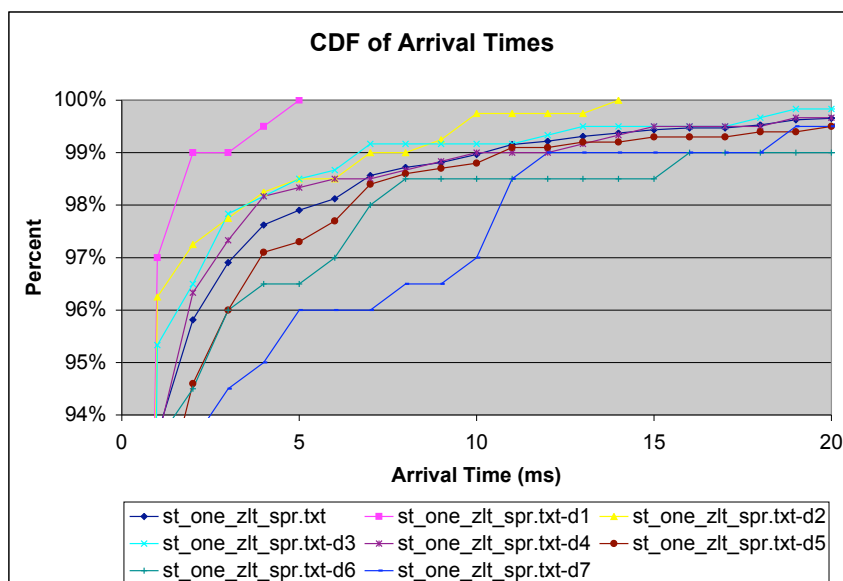


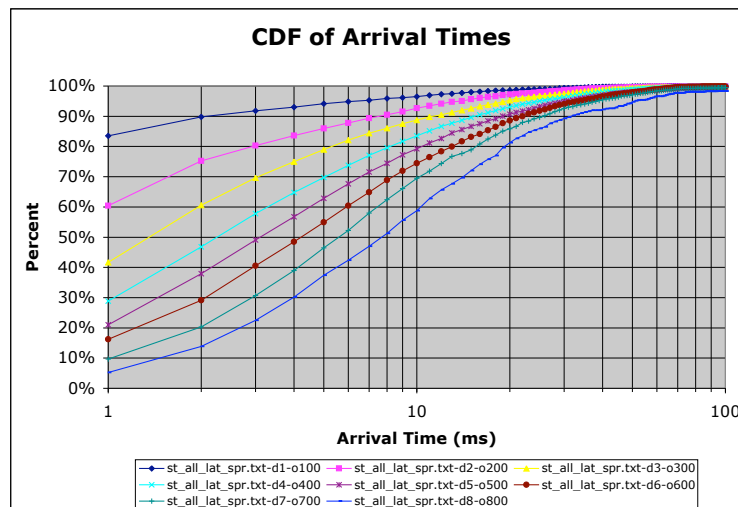
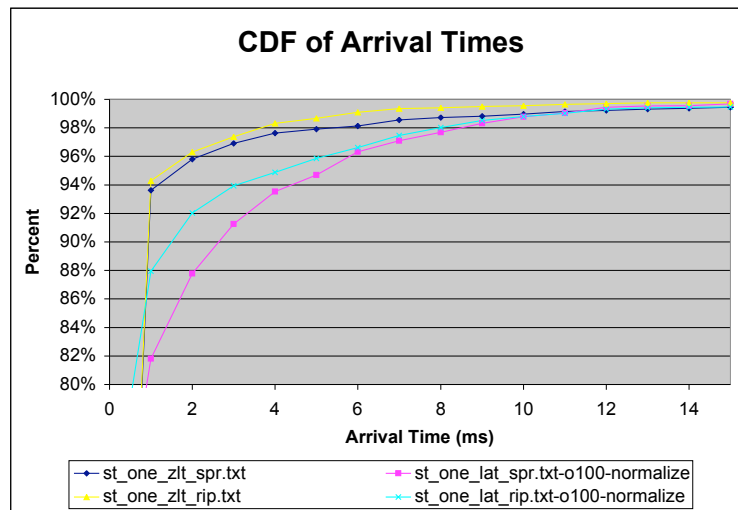
Figure 7: The Effects of Distance

The following results depict the overhead attributed to modeling latency in Simnet. Figure 7 compares the overhead of sending packets a varying number of hops when there is one sender, no latency, and shortest path routing is used. These results show, for example, that 95% of packets that had to travel 7 hops were processed within 10ms of being sent. The second example (Figure 13.2)(top) compares the overhead of packets due to latency. It also compares the differences between overhead in RIP and ShortestPath.

The graph labels simply indicate what type of *speed test* was performed. “One” denotes that there was only one sender (JHU), while “all” means the multiple-sender case. “Zlt” means that all links has no latency, while “lat” means that all links have 100ms latency (in both cases, links had infinite queue sizes and bandwidth).

“Spr” denotes that shortest path routing was used, while “rip” indicates that RIP routing was used (again, test were not started until RIP converged).

These graph shows that there is very little difference in overhead between RIP and ShortestPath. It also shows that using latency causes a numerically larger amount of overhead, but a relatively smaller percentage overhead, compared to not using latency. Finally, Figure 13.2(bottom) shows the actual amount of overhead for packets when latency is used, all nodes were sending, and ShortestPath routing was used. There is a separate graph for each distance. These results show, for example, that 90% of packets that traversed 8 hops were processed within 30ms of being sent (which is only  $30/800 \approx 4\%$  overhead).



## 13.3 Sample Tests

A number of small test scripts are provided as examples in `$$SIMNET/scripts/`. These may be executed by running `java Simnet -s scripts/filename`. These tests provide a good introduction to some of the functionality provided by Simnet.

## 14 Miscellaneous notes

**Verbosity** The amount of output that Simnet produces can be controlled in two ways. First, the verbosity level determines which printouts actually get printed. Level 0 means use only the most important printouts, while higher values produce more output.

The second method of controlling output is by using categories. Every call to **printout** or **debug** should use one of the `Simnet.VC_xxx` constants depending on what type of printout it is. If no category is specified (or null is explicitly used), output for that category cannot be turned off through the User Interface. The special category of `“*”` means use the printouts for all categories.

To support IIW’s, printouts need to be done differently, depending on the calling context. Applications should use their own **printout** method. Any printouts associated with a specific node should use that node’s **printout** method. Finally, any general or system wide message should use the Simulator’s **debug** method.

### “Fun” with Reflection

- Object sizes are determined using reflection via (`getObjectSize`)
- Object sizes are cached whenever possible to minimize overhead.
- The `ancestorCopy` method will copy all non-static, non-transient, and non-null fields from an object’s common superclass into another object. Note that static, transient, and null fields are not included when calculating an object’s size.

**Improved Scripting** New scripting functionality added to support:

- c++ block style comments: `“/*”` and `“*/”`.
- early exit from script with `“END_SCRIPT”`.
- a record command that saves the commands that are typed in (to help generate scripts).

### Words of Wisdom

- When performing benchmarks (e.g., for your projects), it is prudent to always redirect output to a file.
- Note that with the use of `plug`, a possible race condition may exist: after an object (typically an Application) calls `updateConsumer` in its `preCopy` method, a packet can arrive before the object’s replacement gets initialized. In this case, a “dirty hack” is implemented to ignore the packet, rather than force `inBPF` to operate before the application was initialized. This race condition may no longer persist as a node is now first paused when performing `plug` operations — though, we have not thoroughly tested this assumption.
- There is no local loopback device available; the lookup function in both routing implementations will return -1 (no route to host) if the destination is the node itself.
- If there is a change in the topology and shortest path routing is being used, the update shortest path command (`usp`) must be run in order to update the routing tables.

## References

- [CAIDA] *UC-San Diego's Cooperative Association for Internet Data Analysis*. See <http://www.caida.org>.
- [DC95] D. Comer. *Internetworking with TCP/IP Volume 1: Principles, Protocols and Architecture*. Prentice Hall, 1995.
- [FBSD] The FreeBSD project. See <http://www.freebsd.org>.
- [IB02] J. Ioannidis and S. M. Bellovin. *Implementing Pushback: Router-based defense against DDoS Attacks*. In *Proceedings of Network and Distributed System Security Symposium*, 2002.
- [GM99] P. Gupta and N. McKeown. *Packet Classification on Multiple fields*. In *SIGCOMM '99*, pages 147-160.
- [GM01] P. Gupta and N. McKeown. *Algorithms for Packet Classification*. In *IEEE Network Special Issue*, vol. 15, no. 2, pp 24-32, 2001.
- [HNc] B. Huffaker, E. Nemeth, k claffy. *Otter: A General Purpose Network Visualization Tool*. See <http://www.caida.org/outreach/papers/1999/otter/>.
- [MJ93] 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.
- [OBSD] The OpenBSD project. See <http://www.openbsd.org>.
- [RFC793] The Internet Engineering Task Force (IETF). See <http://www.ietf.org/rfc/rfc0793.txt>.
- [RT83] R.E.Tarjan. *Data structures and network algorithms*. SIAM, 1983.
- [SSV99] V. Srinivasan, S. Suri and G. Varghere. *Packet Classification using Tuple Space Search*. In *SIGCOMM '99*, pages 135-146.
- [SV00] S. Save, D. Wetherall, A. Karlin, T. Anderson. *Practical network support for IP Traceback*. In *Sigcomm 2000*, pages 295-306.
- [RS94] R. Stevens. *TCP/IP Illustrated Volume 1: The Protocols*. Addison Wesley Publishing Co., 1994.
- [WS95] G. Wright and R. Stevens. *TCP/IP Illustrated Volume 2: The Implementation*. Addison Wesley Publishing Co., 1995.
- [H88] Hedrick, C.L. June 1, 1988, "Routing Information Protocol", RFC 1058, Rutgers University.
- [KR00] Kurose, James F. and Ross, Keith W. 2000, *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison-Wesley Publishing Company, Reading, Massachusetts.
- [R95] Rice, J. A. 1995, *Mathematical Statistics and Data Analysis*, Second Edition. Duxbury Press: Belmont, CA.
- [GT98] Goodrich, Michael T. and Tamassia, Roberto 1998, *Data Structures and Algorithms in Java*, John Wiley and Sons, Inc.

## A Commands

The following tables depict the list of commands — for an up-to-date list it is best to examine the source code and related Javadocs.

| Command                             | Description  |
|-------------------------------------|--|
| load file                           | loads network topology described in <b>file</b>  |
| view                                | display the currently loaded network graphically using Otter.                                  |
| lookup node                         | returns the IP address/name of the node with the specified name/IP                             |
| script file                         | runs the script <b>file</b>  |
| wait time                           | pauses execution of a script for <b>time</b> ms  |
| select node                         | makes the current node the one with the specified name/IP                                      |
| help                                | lists all the commands and their syntax (including some commands that are not in these tables) |
| print                               | runs the toString() method on a node   |
| loadelement le <topology file line> | loads a node or link   |
| unload ul l L <source> <dest>       | removes a link   |
| isolate iso <node>                  | isolates a node  |
| integrate int <node>                | integrates a node  |
| disablelink dl <source> <dest>      | disables a link between source and dest  |
| enablelink el <source> <dest>       | enables a link   |
| save <file>                         | saves the current topology to a file   |
| updateshortestpath usp              | recomputes shortest path routing table   |
| settimer <name>                     | sets a timer with given string   |
| getElapsedTime <name>               | gets the elapsed time since a timer was set with current string                                |

Table 3: System Commands (nodes can be specified by either their IP address or name)

## B Otter File Format

Every line in an Otter file starts with a tag that specifies the kind of entry the line contains. Simnet only supports a few of the many Otter tags (see [HNc] for more details). The supported tags are summarized in Table 7.

All Otter files must start with a **t** and **T** line which set the total number of Nodes and Links respectively. This is followed by **?** lines which define all the Nodes and finally by **L** and **I** lines which define uni-directional and bi-directional links respectively. Lines that start with the **#** sign are ignored.

| Command   | Description   |
|---|---|
| <code>start</code>  | starts the current node   |
| <code>stop</code>   | stops the current node  |
| <code>clear</code>  | clears statistics of the current node   |
| <code>ps</code>   | prints list of current Application plugins  |
| <code>fw add rule_id deny allow protocol<br/>from src to dest dest_port via<br/>in out link_src_ip</code> | adds a firewall filtering rule to the current node that is numbered <b>rule_id</b> and either <b>denies</b> or <b>allows</b> packets of type <b>protocol</b> that originate from node <b>src</b> and are destined to node <b>dest</b> at port <b>dest_port</b> and that are either <b>in</b> or <b>out</b> bound on link <b>link_src_ip</b> |
| <code>fw delete rule_id in out<br/>link_src_ip</code>   | deletes the rule numbered <b>rule_id</b> in the firewall associated with the <b>in</b> or <b>out</b> bound link with ID <b>link_src_ip</b>  |
| <code>plug in out obj</code>  | loads or removes plugin <b>obj</b> to/from the current node.  |

Table 4: Node Commands

| Command                | Description      |
|------------------------|------------------|
| <code>start all</code> | start all nodes  |
| <code>stop all</code>  | stops all nodes  |
| <code>clear all</code> | clears all nodes |

Table 5: Broadcast Commands

| Command                              | Description   |
|--------------------------------------|---|
| <code>ping node</code>               | pings the node with the specified name/IP   |
| <code>traceroute node</code>         | performs a traceroute to the node with the specified name/IP  |
| <code>synflood node dest_port</code> | performs a SYN Flood against the node with the specified name/IP to the specified destination port                            |
| <code>dump src node dest node</code> | starts to log all packets coming from the node with the specified name/IP and destined to the node with the specified name/IP |
| <code>stopdump</code>                | stops logging packets at the current Node   |

Table 6: Additional commands offered by default plugins — a node can be specified by either its IP address or its name.

| Tag                          | Description   |
|------------------------------|---|
| <code>t number</code>        | total number of Nodes   |
| <code>T number</code>        | total number of Links   |
| <code>? IP name r/h</code>   | defines a Node with the specified <b>IP addr</b> , <b>name</b> and whether it is a router ( <b>r</b> ) or a host ( <b>h</b> ) |
| <code>L from_IP to_IP</code> | defines a bi-directional Link between <b>from_IP</b> and node <b>to_IP</b>  |

Table 7: Otter tags (see networks/test.net for example)