x-Sim User's Manual (Version 1.0)*

Lawrence S. Brakmo
Andrew C. Bavier
Larry L. Peterson
Venkatesan Raghavan

July 2, 1997

Abstract

x-Sim is a network simulator based on the x-kernel. It provides a complete and realistic framework for developing, analyzing, and testing network protocols. This document is intended to be used as a reference to x-sim by those already familiar with the x-kernel.

*This work supported in part by National Science Foundation Grants CCR-8811423, IRI-9015407, and CCR-9102040, Sun Microsystems Inc., Hewlett-Packard Company, ARPA Contract DABT63-91-C-0030, and NCSC University Research Grant MDA904-92-C-515.

Contents

1 Overview

x-Sim is a network simulator based on the x-kernel. It provides a framework for developing, analyzing, and testing network protocols; for evaluating congestion control mechanisms; and for studying network dynamics. This document is intended to be used as a reference to x-sim by those already familiar with the x-kernel.

x-Sim includes all of the modules necessary to simulate a large and arbitrarily complex internetwork. An internet composed of Ethernets and point-to-point links, FCFS and RED routers, and hosts with arbitrary protocol graphs can be specified. In addition, the user may write code to implement other types of networks and routers for inclusion in a simulation.

x-Sim is tightly integrated with the x-kernel architecture; it runs x-kernel protocols and protocols can be moved between the simulator and the x-kernel without any changes. x-Sim Version 1.0 adheres to Version 3.3 of the x-kernel interface [NSR96]. Despite the close relationship between x-sim and the x-kernel, there are two important differences between the standard x-kernel and the simulator of which the user should be aware. We discuss these differences here.

First, the x-kernel allows the hosts on which it runs to communicate with each other, either through direct network access (i.e., directly accessing the network device), or by emulating a low-level network connection via sockets. In contrast, the simulator does not use the network at all. Instead, it simulates communications between potentially thousands of hosts over an internetwork specified by the user. A special protocol called SIM is included at the bottom of all host protocol stacks; host driver protocols pass messages to SIM, and SIM implements the networks and routers that make up this internet.

Figures 1 and 2 show the protocol graphs of a user-level x-kernel and an instance of x-sim, respectively. Figure 1 shows two hosts, each running a user-level x-kernel with its own protocol stack, connected by an Ethernet. Protocol SIMETH on each host uses the Unix socket interface to send messages across the network.[1] In contrast, Figure 2 shows a protocol graph representing four simulated hosts connected by an internetwork. Protocol SIM contains code that implements the entire internetwork to which the hosts are attached. Note that in x-sim, each simulated host is implemented by its own protocol stack; this leads to multiple instantiations of protocol objects, and a large protocol graph.

Second, in a user-level x-kernel, the protocol graph is static and it is specified at compile time in the `graph.comp` file. In contrast, the simulator has been enhanced so that the protocol graph is created at runtime, allowing different simulations to be run without recompiling. The `graph.comp` file is still used by x-sim, but only as a means of specifying which protocol modules to include in the executable. In other words, if any simulated host is going to be running protocol P, then protocol P must appear somewhere in `graph.comp`. The relationships between protocols specified in `graph.comp` are ignored by the simulator.

At runtime, x-sim reads a configuration file, called `xsim.data`, or any other file that can be specified by the **`-simfile`** option, which specifies the network topology: the number and types of networks; the number of hosts and routers; how the networks, hosts and routers are connected; the protocol graph for each host; and various other parameters such as the addresses of each host and network, the speed and delay of each point-to-point link and network, whether Ethernet networks should simulate collisions, and so on.

[1] We call this device protocol SIMETH for historical reasons. SIMETH has nothing to do with SIM; the former simulates an ethernet using sockets, and the latter simulates an entire internet on a single machine.

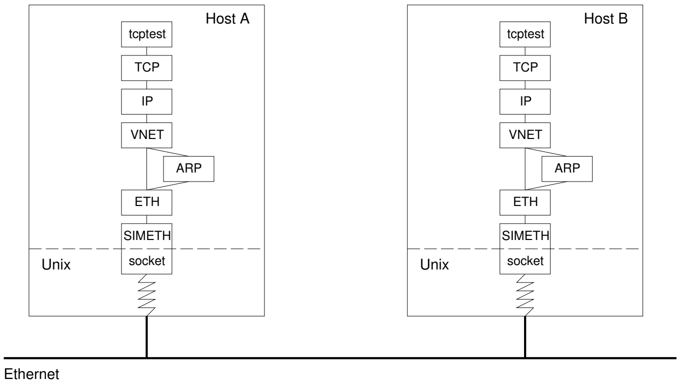

Figure 1: A user-level x-kernel protocol graph

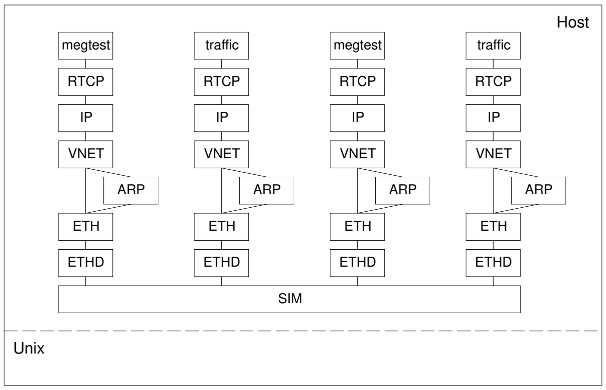

Figure 2: An x-sim protocol graph

In summary, building the protocol graph for a simulation involves two stages: including the necessary protocol modules in the simulator executable (via `graph.comp`), and specifying at runtime the protocol graphs for the individual simulated hosts (via `xsim.data`). To illustrate this point, here is the `graph.comp` file for the simulator instance shown in Figure 2:

```
@;

name=sim;
name=ethd;
name=eth;
name=arp;
name=vnet;
name=ip;
name=ttcp;
name=rtcp;
name=btcp;
name=vtcp;
name=megtest;
```

```
name=traffic;

@;

prottbl=prottbl.default;
prottbl=prottbl.xsim;
```

Notice that nowhere are multiple hosts mentioned in this `graph.comp` file, and no explicit protocol relationships are given. (You can give them if you want, but they have no effect.) Also note that we included multiple TCP protocols (i.e., TTCP, BTCP, VTCP), not all of which are involved in the simulation illustrated in Figure 2. If later we want to replace RTCP in our simulation with another TCP, or have different TCP protocols running on different hosts, we can do so without recompiling because we've already included these protocols in the simulator.

Following the standard *x*-kernel build procedures defined in the *x*-kernel Programmer's Manual [NSR96], the above `graph.comp` file produces an *x*-sim executable in the build directory. Section 2 shows the `xsim.data` file that is read at runtime by this executable, resulting in the protocol graph shown in Figure 2.

Finally, we reiterate that only minimal modifications to the *x*-kernel were necessary so it could be used as the *x*-sim infrastructure. No modifications were made to the *x*-kernel interface, thereby making it possible to move protocols between the simulator and the standard *x*-kernel without modification.

The next section provides an example configuration file for a simple internetwork and explains its components; Figure 2 shows the protocol graph produced by this file. Section 3 then documents all aspects of the `xsim.data` file in detail. Section 4 includes a more complex example `xsim.data` file. Finally, running a complex simulation is only half the battle. Often, analyzing the results takes much longer than running the simulation itself. A set of graphical display tools are included with *x*-sim to aid in analyzing the behavior of different parts of the simulator. These tools display graphs based on traces saved by protocols and routers, and have proven invaluable in helping to understand complex protocols. The final section of this documents briefly describes these display tools.

2 Example Internet Configuration

A simulated internet is constructed out of three kinds of building blocks: *networks*, *routers*, and *hosts*. The `xsim.data` file describes all aspects of the simulated internet, including its topology, its link types and speeds, the buffer space available on its routers, and the protocol stacks running on its hosts. The following section presents an `xsim.data` file for a simple internetwork.

Our example internet contains two Ethernets and a 200KBps point-to-point link. A FCFS router attaches to each Ethernet, and the point-to-point link connects the routers. Two hosts, "1" and "2", are configured on each Ethernet, for a total of four hosts. When the simulator is executed, a 1MB file is transferred between the hosts named "1"; the hosts named "2" conduct random *tcplib* transfers to provide the background traffic [CDJM91]. Figure 3 shows a picture of our example internet.

Here is the `xsim.data` file we've constructed to represent the pictured internet:

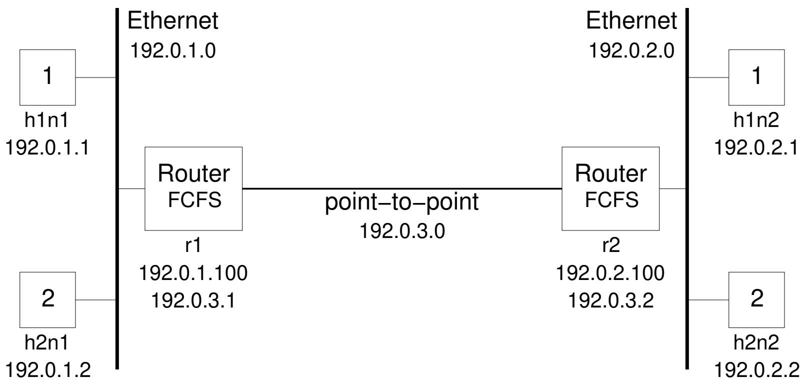

Figure 3: The internet to be simuated.

```
set ip = ip, ip vnet, vnet eth arp, arp eth, eth ethd, ethd sim;
set buf = 50;
set db = db.out;
set delay = 10ms;
set megTime = 30;
set time = 90;
set rbuf = 15;

#
# Define hosts
#
host h1n1;
  protocols = megtest rtcp, rtcp $ip;
  args = -c192.0.2.1 -buf=$buf -tcpTrace=60000
         -db=$db -delay=$delay -megTime=$megTime;

host h2n1;
  protocols = traffic rtcp, rtcp $ip;
  args = -hosts 1 192.0.2.2 -cit=30 -db=$db -trafficTime=$time
         -trafficTrace=60000 -ethdTraceOut -reset=10;

host h1n2;
  protocols = megtest rtcp, rtcp $ip;
  args = -s;

host h2n2;
  protocols = traffic rtcp, rtcp $ip;
  args = -s;

#
# Define routers
```

```
#
router FCFS r1;
  rtable = 192.0.1.0 192.0.1.0, 192.0.2.0 192.0.3.2;
  args = buf=$rbuf, shortTrace=10000, traceQueue=192.0.3.1,
         traceOut=192.0.3.1;

router FCFS r2;
  rtable = 192.0.2.0 192.0.2.0, 192.0.1.0 192.0.3.1;
  args = buf=$rbuf, shortTrace=10000, traceIn=192.0.2.100,
         traceQueue=192.0.3.2;

#
# Define networks, connectivity
#
net ETH 192.0.1.0;
  connections = r1 192.0.1.100, h1n1 ethd 192.0.1.1, h2n1 ethd 192.0.1.2;
  args = collisions=off;

net ETH 192.0.2.0;
  connections = r2 192.0.2.100, h1n2 ethd 192.0.2.1, h2n2 ethd 192.0.2.2;
  args = collisions=off;

net PP 192.0.3.0;
  connections = r1 192.0.3.1, r2 192.0.3.2;
  args = rate=200KB, delay=10ms;
```

We now highlight certain aspects of this `xsim.data` file, and point out how parts of this file correspond to features of the internet of Figure 3. For more details on any of the items discussed, see Section 3.

The first lines of the example file contain `set` commands. The `set` command simply associates a variable with a string. Subsequent references to the variable, prefixed with a `$` sign, are replaced by the associated string.

Next, we find a group of `host` entries. These entries define the configurations and behaviors of the four hosts shown in Figure 3. Following the keyword `host` in each entry is the host's name. The `protocols` line in each `host` entry indicates the protocol stack on that host; those hosts with `megtest` at the top of their stack will stream the 1MB file, while those with `traffic` on the stack will conduct the random transfers. (The protocol stacks are not shown in Figure 3.) Finally, the `args` line contains arguments interpreted by the host's protocols. An argument of particular significance is the `-c192.0.2.1` of host `h1n1`, which indicates that this host is a `megtest` client talking to a server at IP address 192.0.2.1; correspondingly, host `h1n2` specifies the `-s` argument to indicate its willingness to accept a `megtest` connection from any client. The `-hosts 1 192.0.2.2` argument of host `h2n1` similarly indicates that this host is a `traffic` client, talking to a server at the given IP address.

Following the host specifications are entries to define the two routers. The keyword `router` is followed by the type of router (FCFS) and the router's name. The `rtable` line lists the routing table for the router. Referring to Figure 3 and the entry for router `r1`, we note that since the router connects to network 192.0.1.0, it forwards packets with that network destination directly to the appropriate host. However, it directs packets addressed to network 192.0.2.0 to the router at IP address 192.0.3.2. The final lines of a given `router` entry contain arguments used to configure that router.

Now consider the `net` entries that specify the two Ethernets and the point-to-point link. The first line lists the type of network (e.g., ETH) and its network IP address. The `connections` line lists all of the routers and hosts connected to the network, and assigns them their IP addresses. In the case of hosts, a driver instance—`ethd` in the example—is also specified. The `args` line sets various network-specific parameters, such as the 200KBps rate of the point-to-point link.

The next section documents all aspects of the `xsim.data` file in detail. Section 4 then presents a second, much larger, `xsim.data` file.

3 File `xsim.data` Specification

The `xsim.data` file specifies most aspects of a simulation, including the structure of the simulated internetwork and the tests to be run. This section fully describes all of the components of this file.

The `xsim.data` file consists of a set of *entries*, each of which describes an internet component (e.g., a network, router, or host), or defines a macro variable. Entries that describe components are multi-line, as follows.

- The first line identifies and names the component.
- The second line contains configuration information central to the type of component being defined. For networks, connectivity is specified; for routers, the routing table is specified; and for hosts, the protocol graph is given.
- The remaining line(s) list component-specific parameters. Many of these parameters control the kinds of tracing done by the component.

The format of each type of entry is documented in the following subsections.

3.1 The `set` Entry

The `set` entry allows you to associate a string with a variable; the string will replace references to the variable later in the text. For example,

```
set ip = ip, ip vnet, vnet eth arp, arp eth, eth ethd, ethd sim;

...

host h1n1;
  protocols = megtest vtcp, vtcp $ip;
```

means that the reference to `$ip` is replaced by the protocol stack on the right side of the `set` command, which is equivalent to the following

```
host h1n1;
      protocols = megtest vtcp, vtcp ip, ip vnet, vnet eth arp, arp eth,
                  eth ethd, ethd sim;
```

3.2 The `convert` Entry

This command is useful when using a program to automate the process of creating large internetworks. All hosts can be given the same arguments, and then the `convert` command can be used to save traces for only a few hosts. The format of the `convert` entry is.

```
convert <var-name> in <obj-name> to <var-name>;
```

An example illustrates its use.

```
convert tcpTrace2 in h1n20 to tcpTrace1;
```

This example specifies that the value of `tcpTrace1` should be used in place of the value of `tcpTrace2`, wherever the variable `tcpTrace2` is encountered in the specified object.

3.3 The `net` Entry

The `net` entries in the `xsim.data` file specify the kinds of networks to include in the simulated internet, the parameters for these networks (e.g., bandwidth, latency), and the IP addresses for all networks, routers, and hosts involved in the simulation. The connectivity for the entire internetwork is specified by the `net` entries alone.

In the `xsim.data` file, a particular `net` entry must come *after* the entries of all hosts and routers that it connects. The syntax of a network entry is defined as follows:

```
net <type> <net address>;
  connections = <name> [instance] <address>, <name> [instance] <address>,... ;
  args = <arg>, <arg>, ... ;
```

The `net` line specifies the type of network. Two types of networks are currently defined: Ethernet (`ETH`) and point-to-point (`PP`). The properties of each type are documented in subsequent sections. The `<net address>` field specifies the network IP address. The host portion of this address can be 0, although the full address may be specified, as is done in the example given in the previous section.

The `connections` line lists the IP addresses of the hosts and routers connected to this particular network; only the host part of the IP address should be specified. The optional `instance` is used only when specifying a host, and indicates the name of the driver instance that is connected at that IP address (e.g., `ethd/1`).

There are several things to keep in mind when specifying IP addresses for networks, hosts, and routers.

- The simulator only saves the last two bytes of an IP address in a trace entry. This is done to conserve space. Therefore, to get unambiguous trace results, each host and router should be uniquely identified by the last two bytes of its IP address.

- The simulator calculates the full IP address for a host or router by performing a logical OR between the network IP address (specified in the `net` line) and the host IP address (from the `connections` line).

- For a large internetwork, judicious choices of IP addresses can make the job of specifying routing tables in the `router` entry easier. See Section 3.4 for information on how to pick IP addresses so that it is easier to use the routing "shortcuts" provided by the simulator.

The `args` line includes parameters specific to the type of the network being defined, and are used to define aspects of the network such as its bandwidth, the network MTU, and the associated framing overhead. Parameters for Ethernet are listed in Section 3.3.1, paremeters for point-to-point links are given in Section 3.3.2. A syntactic point to be aware of is that no spaces should be included anywhere between the argument and its value: `rate=10Mb`.

3.3.1 Ethernet (ETH)

The Ethernet network type simulates a multiple access network with optional collision detection. Note that this is really an abstract network type, capable of simulating networks with a variety of characteristics. If no arguments are specified, the network's default configuration is that of a 10Mbps Ethernet, with the exception that collision detection is disabled. However, the bandwidth, latency, packet overhead, and MTU can be changed to produce networks with behaviors different from that of the standard Ethernet.

The following arguments can be specified on the `args` line of an Ethernet entry.

`collisions= on | off`

Enables/disables Ethernet collisions. When this option is off, two or more hosts are allowed to successfully transmit on the Ethernet simultaneously. When on, if two hosts send simultaneously, a collision is detected and the hosts perform the Ethernet exponential backoff algorithm. The default is `off`.

`delay=<integer>[unit]`

Sets the latency, that is, the time it takes for a packet to propagate from one end of the network to the other. The optional unit can be `us` (microseconds), `ms` (milliseconds), or `s` (seconds). The default value is 0, and the default unit is microseconds.

`maxPacket=<integer>`

Specifies, in bytes, the maximum packet size for the network. Packets larger than this maximum should be fragmented by a higher-level protocol; if they are not, they will be dropped. The default `maxPacket` for `ETH` is 1518, comprising 1500 bytes of payload, 14 bytes of Ethernet header, and a 4-byte CRC.

`minPacket=<integer>`

Specifies, in bytes, the minimum packet size for that network. Packets smaller than the minimum will be padded. The default `minPacket` for `ETH` networks is 64 bytes, 14 of which are for the Ethernet header, and 4 for the CRC.

`pktHeader=<integer>`

For use with minPacket and maxPacket. The simulator adds the value of pktHeader to the length of the message pushed to SIM to determine the packet's length. In ETH, since the header is attached by the ethd protocol, the network's pktHeader defaults to 0. In other words, the size of a *packet* is given by the size of the *message* pushed to SIM, plus pktHeader. If *packet* $<$ minPacket, it is padded; if *packet* $>$ maxPacket, it is dropped. When using ETH networks to model other types of networks, one may want to use negative values for pktHeader when the modeled network's header is less than 14 bytes.

```
pktOverhead=<integer>
```

Conceptually, this option adds the specified number of bytes to each packet as it is transmitted on the network. The intended use is to model the overhead consumed by framing information. The added pktOverhead does not count as part of the packet length from the standpoint of minPacket or maxPacket. The default is 8, as Intel's Lance uses 8 bytes for framing.

```
rate=<real>[unit] | T1 | T3
```

Sets the bandwidth, in units per second, for the network. The rate can be a positive real number followed immediately by an optional unit. The unit is one of: MB, Mb, KB, Kb, B, or b. A B in all units represents bytes and b denotes bits. For instance, 4.5Mb specifies a bandwidth of 4.5 mega*bits* per second. If no unit is specified, the default is bytes. T1 and T3 can also be specified as rates. T1 corresponds to approximately 1.5Mbps, and T3 is about 45Mbps. For Ethernet, the default bandwidth is 10Mbps.

3.3.2 Point-to-point Links (PP)

The point-to-point network type can be used to connect two routers or hosts. It differs from the Ethernet in that it should be thought of as two links, one in each direction. Packets flowing in one direction cannot affect flows in the other direction.

The following arguments can be used to modify the behavior of a simulated point-to-point link. They have the same meanings as those specified above for the Ethernet.

```
delay=<integer>[unit]               # default is 0
minPacket=<integer>                 # default is 0
maxPacket=<integer>                 # default is 1500
pktHeader=<integer>                 # default is 0
pktOverhead=<integer>               # default is 0
rate=<real>[unit] | T1 | T3         # MUST be specified
```

3.4 The router Entry

A router entry indicates a type of router, gives its routing table, and sets the number of router buffers. Routers can output several different kinds of trace information, and the type and amount of tracing is specified here too. The format of the entry is:

```
router <type> <name>;
  rtable = <destination> <next hop>, <destination> <next hop>,... ;
  args = <arg>, <arg>, ... ;
```

The `router` line specifies the type of network. Two types of routers are supported at present: `FCFS` and `RED`. The router's name must match that used in the `net` entry.

The `rtable` line specifies the routing table. The `destination` is the network portion of a packet's IP address and the `next hop` is where the router should send a packet with that network destination. If the next hop has no host address, then it is assumed that the router is directly connected to the destination network and it should send the packet to the final destination host. Note that the entries specified on the `rtable` line are scanned *in order* by the simulator. In other words, if multiple `next hops` are present for a particular destination, the router sends the packet to the first one listed. This fact becomes important when using shortcuts to set up a routing table, as described below.

The simulator interprets a few IP addresses in a nonstandard way. This is done to make it easier to configure a large internetwork. If IP addresses are chosen so that networks with similar IP addresses are near each other in an internetwork, then routing tables can be specified quickly and easily by using the special addresses as "destinations" in the routing table. Given a packet with destination IP address of the form **a.b.c.d**, the following IP addresses have special meaning when used in the `destination` field:

- 0.0.0.0: Route packet to the specified next hop, regardless of destination. This allows the specification of a default route.
- 1.0.x.0: If $c \leq x$, route packet to specified next hop.
- 2.0.x.0: If $c \geq x$, route packet to specified next hop.
- 3.x.y.0: If $x \leq c \leq y$, route packet to specified next hop.

The **c** in all of the above relations refers to the third component in the packet's destination address, **a.b.c.d**. Clearly, these special addresses are most useful if all networks in a simulation are uniquely identified by their **c** components.

For example, consider the following entry in a routing table:

```
router FCFS r1;
  rtable = 3.1.5.0 192.1.10.100, 0.0.0.0 192.1.20.100;
```

Packets with IP destination address of the form **a.b.c.d** will be routed to 192.1.10.100 if $1 \leq c \leq 5$. Therefore, a packet destined to 192.12.3.79 will be sent there, but 192.12.6.3 will not. The latter packet falls through to the last entry, which routes that packet to 192.1.20.100.

The `args` line of the `router` entry contains parameters for the router. These parameters set the number of router buffers and select trace options. Parameters for the FCFS router are listed in Section 3.4.1. A syntactic point to be aware of is that no spaces should be included anywhere between the argument and its value; e.g., `buf=30`.

3.4.1 First Come First Serve (FCFS)

A `FCFS` router queues and forwards packets in the order in which they arrive. Each output port of the router has a queue associated with it. These queues are of limited size, and packets routed to an output port while its queue is full are dropped by the router.

The following arguments can be used with the router:

`buf=<integer>`

Sets the number of buffers in the router's queue. The router will be able to queue up to this number of messages; any additional messages will be dropped. The default is 0, so this parameter must be specified.

`shortTrace=<integer>[:<integer>]`

Specifies the lowest level of trace detail, mainly information about losses. The integer arguments specify the quantity of resources dedicated to saving the trace information. The first integer argument specifies the size, in trace entries (about 8 bytes each), of the circular buffer used to hold trace information in memory. If the optional `:<integer>` portion of the argument is *not* specified, this number also represents the maximum size of the trace file. Thus, an argument of `10000` specifies a buffer size of 10,000 entries, and the trace file will contain no more than this number of entries. If the second integer argument is given, it specifies the maximum number of memory buffers that the disk file may contain. When this argument is specified, then once the circular buffer is full, the buffer is dumped to the trace file before its traces are overwritten. Thus, `10000:3` specifies a buffer size of 10,000 entries and a trace file of up to 30,000 entries. Be aware that the trace file also operates as a circular buffer, and if more than the maximum number of trace entries are generated, the earliest will be overwritten. In general, it is recommended that the file be bigger than the internal trace buffer. Also note that niether for the vaules for the variable can exceed 320000. Therefore the largest trace buffer you can have is `320000:320000` which means there will be a maximum of 320000 entries of 320000 length buffers in the trace file.

`pktTrace=<integer>[:<integer>]`

Specifies a medium level of trace detail, mainly information is saved for each packet routed (e.g., packet length, IP addresses, TCP ports). See `shorttrace` for information on the meaning of the integer arguments. Also see `pktTracePortIn` and `pktTracePortOut`.

`trace=<integer>[:<integer>]`

Specifes the highest level of trace detail. This option traces everything a router does, and is used mostly for debugging. Since `trace` implies all of the tracing of `pktTrace`, the options used to identify interfaces to be traced with `pktTrace` (e.g., `pktTracePortIn`) can also be specified with `trace`. See `shorttrace` for information on the integer arguments.

`pktTracePortIn=<IP address>`

If `pktTrace` is specified, save trace information for packets arriving on the router interface specified by `<IP address>`. If you wish to trace multiple ports, include multiple `pktTracePortIn` options, one for each port.

`pktTracePortOut=<IP address>`

If `pktTrace` is specified, save trace information for packets leaving by the router interface specified by `<IP address>`. If you wish to trace multiple ports, include multiple `pktTracePortOut` options, one for each port.

`traceIn=<IP address>`

In conjunction with any trace function (e.g., `shorttrace`, `pktTrace`, or `trace`), traces the average number of bytes/second arriving on the specified interface. If you wish to trace multiple ports, include multiple `traceIn` options, one for each port.

`traceOut=<IP address>`

In conjunction with any trace function (i.e., `shorttrace`, `pktTrace`, or `trace`), traces the average number of bytes/second sent on the specified interface. If you wish to trace multiple ports, include multiple `traceOut` options, one for each port.

`traceInOut=<IP address>`

In conjunction with any trace function (i.e., `shorttrace`, `pktTrace`, or `trace`), traces the average number of bytes/second arriving on any interface, but directed to the specified interface. Note that this number may be higher than the `traceOut` for the same interface, since more packets may be arriving than the bandwidth of the outgoing link can accommodate. If you wish to trace multiple ports, include multiple `traceInOut` options, one for each port.

`pktTraceBeginTime=<real>[unit]`
`pktTraceEndTime=<real>[unit]`

Denote an interval during which to save packet traces. Frequently, packet traces are not of interest until traffic connections have stabilized, so the interval can exclude the first few seconds of a simulation. The optional `unit` can be `s` (seconds), `ms` (milliseconds), or `us` (microseconds). The default is milliseconds.

`timeInc=<integer>[unit]`

Specifies a time period used by some of the trace functions. Functions `traceIn`, `traceInOut`, `traceOut`, and `traceQueue` compute an average and a minimum/maximum over this time period, at the end of which a trace entry is saved and a new interval begun. The default value is 100ms. The optional unit can be `us` (microseconds), `ms` (milliseconds), or `s` (seconds). The default unit is microseconds. Multiple `timeInc` options can be specified per line, and the ordering of arguments is important. When a `timeInc` is encountered, all appropriate trace functions following this `timeInc` will use its time period, until another `timeInc` occurs. This allows a different tracing granularity to be used for separate network interfaces.

`traceQueue=<IP address>`

In conjunction with any trace function (i.e., `shorttrace`, `pktTrace`, or `trace`), traces the minimum, maximum, and average number of packets queued to be sent on the specified interface. If you wish to trace multiple ports, include multiple `traceQueue` options, one for each port.

`traceQueueLen=<IP address>`

In conjunction with any trace function (i.e., `shortTrace`, `pktTrace`, or `trace`), this option traces the minimum, maximum, and average number of bytes queued to be sent on the specified interface. This option works like `traceQueue`, except that the number of *bytes* in the queue are traced, rather than the number of *packets*.

3.4.2 Random Early Detection (RED)

A RED router queues and forwards packets in the order in which they arrive, but also implements the RED algorithm to control the queue sizes in the routers. Each output port of the router has a queue associated with it. These queues are of limited size, and packets routed to an output port while its queue is full or if the RED algorithm decides packets are dropped by the router. For more information on the RED algorithm see [FJ93].

The following arguments can be used with the router. They have the same meanings as specified above for the FCFS router.

```
buf=<integer>
shortTrace=<integer>[:<integer>]
pktTrace=<integer>[:<integer>]
trace=<integer>[:<integer>]
pktTracePortIn=<IP address>
pktTracePortOut=<IP address>
traceIn=<IP address>
traceOut=<IP address>
traceInOut=<IP address>
pktTraceBeginTime=<real>[unit]
pktTraceEndTime=<real>[unit]
timeInc=<integer>[unit]
traceQueue=<IP address>
traceQueueLen=<IP address>
```

Other options specific to the RED router which speicify RED parameters include

```
minThresh=<integer>
```

This sets the size of the queue for the minimum threshold for the RED algorithm. IF no value is specified this defaults to zero.

```
maxThresh=<integer>
```

This sets the size of the queue for the maximum threshold for the RED algorithm. If no value is specified this defaults to the number of router buffers - 10.

```
qWeight=<real>
```

This specifies the queue weight for the average queue calucation for the RED algorithm. If now value is specified the default is 0.008

3.5 The `host` Entry

The `host` entries are used to configure the protocol stacks of the hosts attached to the internetwork. The actions a host takes when the simulator is run is determined by its protocol stack, especially by the "application-level" protocol(s) (i.e., `megtest` and/or `traffic`) located at the top of the stack.

The only difference between a simulated host's protocol graph in *x*-sim and a real host's protocol graph in the *x*-kernel is that they specified through different means. In the *x*-kernel, the `graph.comp` file configures the protocol graph. In *x*-sim, the `graph.comp` file is used only to determine which protocols to include in the simulator executable; the relationships between the protocols are ignored. The simulator uses the `host` entries in the `xsim.data` file to configure protocol graphs for each host. Note that although we have created special, heavily-instrumented, versions of many protocols, *any* *x*-kernel protocol can be included in the simulator.

The syntax of a host entry is:

```
host <name>;
  protocols = <hlp> <llp> <llp> ... , <hlp> <llp> <llp> ... , ... ;
  args = <arg> <arg> ... ;
  pargs <protocol instance> = <arg> <arg> ... ;
  pargs <protocol instance> = <arg> <arg> ... ;
  ...
```

The name given in the `host` line must match the name given in the `net` entry.

The `protocols` line configures the protocol stack running on the simulated host. It consists of groups of protocols separated by commas. The first protocol in a group (`hlp`) is configured over the remaining protocols (`llp`) in the same group.

There are a couple of points to be aware of when building a host's protocol graph. First, all protocol graphs in the simulator must contain the special `sim` protocol on the bottom. Second, an application-level protocol like `megtest` or `traffic` must be placed at the top of the stack, otherwise the hosts will not have any reason to send data to each other

The following sections contain more information about these, and other, special simulator protocols. Figure 4 shows a typical protocol graph. This graph is represented in the `protocol` line of a `host` entry as shown below.

```
 protocols = megtest rtcp, rtcp ip, ip vnet, vnet eth arp, arp eth,
             eth ethd, ethd sim;
```

Many protocols accept configuration parameters; the `args` line lists those parameters that all the protocols interpret. Each protocol instance, at initialization, scans through the `args` line to see if it contains any arguments that the protocol understands.

In addition to, or in place of, the `args` line, there can occur one or more `pargs` lines (`pargs` stands for "protocol args"). A `pargs` line allows you to an specify argument list for a specific protocol instance that appeared in the `protocols` line.

An argument listed on the `args` line is interpreted by *all* protocol instances that understand the argument. However, a `pargs` entry becomes necessary when multiple instances of a protocol are configured on a host, and each instance must be given different parameters. For example, assume that a host's protocol stack contains two instances of the Ethernet driver, named `ethd/1` and `ethd/2`. An `-ethdDelay` argument meant exclusively for the `ethd/2` instance may be specified as:

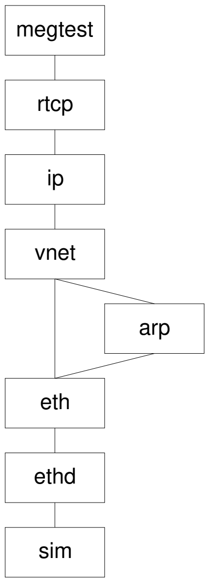

Figure 4: Example protocol graph

```
pargs ethd/2 = -ethdDelay=10us;
```

The following subsections list, by protocol, arguments that can be included on the `args` and `pargs` lines.

3.5.1 Simulator Interface (`sim`)

`sim` is a protocol that lies at the bottom of the protocol graph. It `sim`ulates the networks and routers that make up an internetwork. Passing a message to the `sim` protocol is equivalent to putting it onto a network's physical wire. The `sim` protocol then `sim`ulates the message's journey through the modeled network until it reaches a protocol subgraph representing the final destination host, at which time `sim` passes the message up to that host's protocol graph. `sim` also replies to ARP and RARP requests made by hosts. Special protocols, called driver protocols, are used between standard *x*-kernel protocols and the `sim` protocol. There are no arguments used with the `sim` protocol.

3.5.2 Ethernet Driver (`ethd`)

The `ethd` protocol is the "device driver" that a host uses to connect to either an Ethernet or a point-to-point link. The following arguments for `ethd` may be specified on an `args` or `pargs` line.

`-ethdDelay=<integer>`

Used to simulate the time it takes, in microseconds, for a host to process a packet. During the time period specified, the host cannot accept or process any other packets. If this option is not specified, hosts are assumed to be infinitely fast.

`-ethdDelayRange=<integer>`

A random number of microseconds is added to each delay period specified by `-ethdDelay`. If the argument given is x, then the interval of the modification is between $-x$ and x. Note that the range specified should be less than or equal to the delay specified using `-ethdDelay`.

`-ethdQueue=<integer>`

Specifies the maximum number of messages that can be queued for transmission by ETHD. This is useful when using hosts running IP as routers; it is the only way to specify the maximum queue size for the router outports.

`-ethdTrace=<integer>[:<integer>]`

Indicates that the Ethernet driver protocol should save trace information (e.g., opens, sends, drops) and specifies the amounts of memory and disk space that may be used for this purpose. The first integer argument specifies the size, in trace entries (about 8 bytes each), of the circular buffer used to hold trace information in memory. If no colon and second argument is specified, this number also represents the maximum size of the trace file. The second argument specifies the maximum number of memory buffers that the disk file may contain. See the explanation of `shortTrace` in Section 3.4.1 for an example.

`-ethdTraceIn`

In conjunction with `-ethdTrace`, traces the average number of bytes/second arriving on the interface associated with the `ethd` driver.

`-ethdTraceOut`

In conjunction with `-ethdTrace`, traces the average number of bytes/second sent on the interface associated with `ethd`

3.5.3 Big Window TCP (`btcp`)

An adaptation of BSD Lite's Big Window TCP. Big Window TCP is an implementation of a TCP extension that allows send windows greater than 64KB in size. See RFC 1323 for more information on Big Window TCP. The following arguments for `btcp` may be specified on an `args` or `pargs` line.

`-tcpAcknow`

If this option is specified, the protocol acknowledges every segment. The default is to send an ACK for every 2nd segment, or upon expiration of the fast timeout.

`-tcpSlowstart=<integer>`

Specifies the size of the window, in segments, at the beginning of slow start. The default is 1. This option is included for experimental purposes.

`-tcpTimer=<integer>`

Specifies a delay, in microseconds, before starting the TCP fast and slow timeouts. Without some timer delay, all hosts in a simulation will experience timeouts simultaneously – this could lead to strange simulation results. The preferred alternative to manually specifying a different delay for each host is to use `-tcpTimerAuto`.

```
-tcpTimerAuto
```

Instructs the simulator to stagger the start delay for the timers on each host. If the duration of a timer is t, and there are n hosts, the hosts' timers are started at intervals of t/n milliseconds. For example, the fast timeout fires at intervals of 200 ms. If there were four hosts, their fast timeouts would be started after delays of 0, 50, 100, and 150 ms. Thus, every timer is given a unique start delay.

```
-tcpTrace=<integer>[:<integer>]
```

Trace events of importance to TCP, such as the transmission of a segment, the size of the send window, and the firing of a timeout. The function of the arguments is the same as with the `-ethdTrace` option. (See Section 3.5.2.)

3.5.4 TCP Reno (`rtcp`)

Historically, TCP Reno superceded TCP Tahoe. Reno differs from Tahoe in that the it ACKs every other packet and uses the Fast Recovery mechanism. The arguments used with `rtcp` are identical to those used with `btcp` arguments.

```
-tcpAcknow
-tcpSlowstart=<integer>
-tcpTimer=<integer>
-tcpTimerAuto
-tcpTrace=<integer>[:<integer>]
```

See Section 3.5.3 for information on these options.

3.5.5 TCP Tahoe (`ttcp`)

The arguments used with `ttcp` are identical to those used with `btcp`, with the exception that `-tcpAcknow` has no meaning. This is because `ttcp` sends an ACK for every packet it receives.

```
-tcpSlowstart=<integer>
-tcpTimer=<integer>
-tcpTimerAuto
-tcpTrace=<integer>[:<integer>]
```

See Section 3.5.3 for information on these options.

3.5.6 TCP Vegas (vtcp)

TCP Vegas is a version of TCP, developed at the University of Arizona, that uses a congestion avoidance strategy to decrease losses and improve throughput over TCP Reno. Arguments common to the other TCPs may be used on the `args` and `pargs` lines.

```
-tcpAcknow
-tcpSlowstart=<integer>
-tcpTimer=<integer>
-tcpTimerAuto
-tcpTrace=<integer>[:<integer>]
```

See Section 3.5.3 for information on these options. In addition, several arguments are specific to TCP Vegas.

`-tcpTop=<integer>`

The α parameter used by TCP Vegas to specify the minimum number of extra router buffers to use when not in slow start. The default is 1.

`-tcpBot=<integer>`

The β parameter used by TCP Vegas to specify the maximum number of extra router buffers to use when not in slow start. The default is 3.

`-tcpExp=<integer>`

Specifies the maximum number of extra router buffers to use in slow start. The default is 1.

`-tcpFlags=[e|E][s|S][c|C][p|P]`

This option is followed by a string of up to four letters, each of which enables (lower case) or disables (upper case) the corresopnding `vtcp` option.

- `[e|E]`: Enable/disable congestion avoidance during slow start.
- `[s|S]`: Enable/disable spike prevention (experimental).
- `[c|C]`: Enable/disable congestion avoidance.
- `[p|P]`: Enable/disable prediction during slow start. Predicts the connection bandwidth, so that slow start ends sooner.

Both congestion avoidance options are enabled by default, and the others are disabled. Thus, specifying `p` as an argument is the same as `eScp`.

3.5.7 Megabyte Stream Test (megtest)

megtest is an application-level protocol, that is, it sits at the top of the protocol stack and determines the actions taken by a host during a simulation. A megtest client opens a TCP connection to a megtest server and streams data (default is one megabyte) to the server. The megtest protocol allows us to measure and compare the throughput of the various TCP protocols. The following arguments for megtest may be specified on an args or pargs line.

-buf=<integer>

Specifies the size, in kilobytes, of the TCP send and receive buffers.

-db=<filename>

Specifies the filename in which to print general information about the run. If this option is not specified, this information is not saved.

-dbId=<integer>

After the test, prints the line id: <integer> in the file specified by the -db option.

-dbStr=<string>

Specifies a string to print in the file specified by the -db option after the run has ended. This option is used to print descriptive information about the run, so that multiple runs can be output to the same file and later identified. For example, if one of the important characteristics of a simulation is that all routers use 30 buffers, then the argument, -dbStr=rbufs:30 would print the line rbufs:30 in the output file immediately following the trace information printed by the run. You can have up to 10 different -dbStr per host.

-delay=<integer>[unit]

The delay in (simulated) time units before the test starts. The optional unit can be one of us (microseconds), ms (milliseconds), or s (seconds). The default unit is milliseconds. When running megtest in the presence of background traffic, a delay of a few seconds should be specified in order to give the background connections a chance to stabilize.

-killTraffic

If given as an argument to a single host, after the megtest on *this* host is done, the traffic is killed and the simulation stops. This option is usually used on the command line after -all, where it signifies that after *all* megtest protocols have completed their streaming, all traffic processes are killed and the simulation stops.

-len=<integer>

Sets the amount of data, in kilobytes, to be transferred by the test. The default is 1024, that is, 1MB.

`-megTime=<integer>[unit]`

Stop the `megtest` after it has been allowed to run for the specified number of time units, even if the transfer has not completed. The optional `unit` can be `us` (microseconds), `ms` (milliseconds), or `s` (seconds). The default unit is seconds.

`-reset=<integer>`

Resets TCP trace information after the specified number of seconds. Often, data about transfers is only of interest after the traffic has had a chance to stabilize, which usually takes a few (simulated) seconds.

`-serverPort=<integer>`

Used if a single host is running multiple `megtest` servers, where each server instance must be assigned a port to listen on. A `megtest` client specifies which server on the host it wants to talk to, by specifying the appropriate port in the client's `-serverPort` option.

`-showTime`

If this option is given, every so often the `megtest` process prints the elapsed time.

`-thresh=<integer>`

Specifies the initial threshold variable for TCP's slow start, in 1KB units. The default is one half of the buffer size specified by the `-buf` option.

3.5.8 Background Internet Traffic (`traffic`)

Like `megtest`, `traffic` is an application-level protocol. The `traffic` protocol simulates typical Internet traffic based on *tcplib*—a library that models Internet traffic sources that is based on empirical data collected at different Internet gateways.

`traffic` starts conversations with interarrival times given by an exponential distribution. Each conversation can be of type TELNET, FTP, NNTP or SMTP, and each expects a set of parameters. For example, FTP conversations are parameterized by the number of items to transmit, control segment sizes, and the individual item sizes. All of these parameters are obtained from *tcplib* and are based on probability distributions obtained from real traffic traces. Finally, each of these conversations runs on top of its own TCP connection.

The startup distribution of `traffic` conversations is determined solely by the initial seeds given to the `traffic` protocol, and not by the lower protocols. Therefore, two `traffic` simulations started with the same seeds will start the same *tcplib* conversations at the same times. This is important for reproducing experiments, or for guaranteeing that observed behavior difference between protocols is not due to different traffic patterns.

By default, `traffic` hosts are servers and not clients. In other words, unless otherwise specified, a `traffic` host will respond to requests to open connections but will not initiate connections. Use the `-hosts` or `-rhosts` options to create `traffic` clients. A `traffic` host can be both a client and a server.

The following arguments for the `traffic` protocol may be specified on an `args` or `pargs` line.

-cit=<interval>

Specifies the average connection interarrival time (CIT) to be used at equilibrium. This time is used by the `traffic` protocol to decide when to open new connections; the `traffic` protocol should space new connections by this interval, on average. Note that a lower CIT means that connections are opened more frequently, leading to a greater overall number of open connections. See the **-begcit**, **-begdur**, and **-begint** options for opening connections at simulator startup. See Figure 5 for a diagram that illustrates the relationship among these parameters.

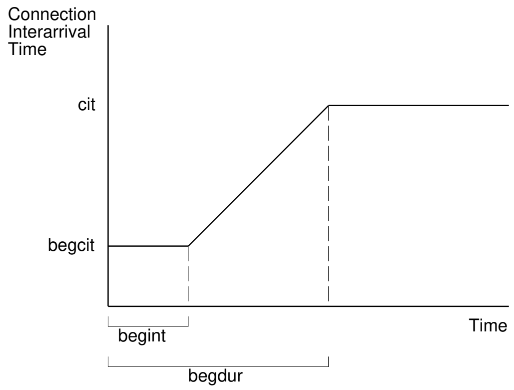

Figure 5: Interaction of CIT Parameters

-begcit=<integer>

Since there are no open connections at the beginning of a simulation, it is often desirable to open a lot of connections at once; setting this variable to a number lower than that used with the **-cit** option will allow connections to be opened more quickly. This number is used as the connection interarrival time until the interval specified with **-begint** has elapsed.

-begdur=<integer>

Specifies the number of seconds to elapse after starting a simulation before the number specified by **-cit** is used as the connection interarrival time. Between the end of the time specified by **-begint** and **-begdur**, the CIT grows linearly toward the level specified by the **-cit** option.

-begint=<integer>

Specifies the number of seconds that the CIT specified by **-begcit** should be used, before the CIT is allowed to grow linearly towards that specified with the **-cit** option.

-begkeep=<integer>

Rather than creating a new thread for each new `traffic` connection and destroying them after the connection closes, the `traffic` protocol caches the threads for later use. The number of threads in the cache will be equal to the maximum number of connections that have been open simultaneously. Since a large number of connections may be opened at the beginning of the simulation by the parameters described above, the cache may contain more threads than will be needed by the rest of the simulation. This option allows excess threads to be freed from the cache. It specifies that the simulator should only keep the specified number of threads in cache; if later it needs more, it will reallocate them. This is an optimization feature—it may improve the running time of a simulation, but will not affect its results in any way. The default value is 100.

`-db=`
`-dbStr=`

These options are the same as specified in Section 3.5.7.

`-ftpBufSize=<integer>`

Sets the send buffer size, in kilobytes, for FTP connections generated by the `traffic` protocol. The default is 20, that is, 20KB.

`-hold=[f][n][s][t]`

Tells the `traffic` host never to start a particular kind of connection. The arguments correspond to the first letters of the types of *tcplib* connections: `f` is FTP, `n` is NNTP, `s` is SMTP, and `t` is TELNET. Thus `-hold=ft` specifies that the `traffic` host should never begin an FTP or a TELNET connection.

`-hosts=<integer> <IP addr 1> <IP addr 2> ...`

Specifies that this host will be a `traffic` client, and lists the servers with which the client will randomly establish connections. The first argument is the number of servers, and the remaining arguments specify the IP addresses of the `traffic` servers.

`-otherBufSize=<integer>`

Sets the send buffer size, in kilobytes, for non-FTP transfers. The default is 10, that is 10KB.

`-reset=<integer>`

Resets the information accumulated about `traffic` transfers after the specified number of seconds. Often, data about transfers is only of interest after the traffic has had a chance to stabilize, which usually takes a few (simulated) seconds.

`-rhosts=<integer>`

Provides an alternative to the `-hosts` option when the user does not want to specify the IP addresses of the `traffic` servers. Like `-hosts`, this option specifies that the host is a client. At startup, the simulator randomly picks the specified number of hosts out of the pool of `traffic` hosts, and the client will establish connections to only this subset of hosts. Note that the selection occurs from all `traffic` hosts, even if they are not servers. Therefore, it is recommended that all hosts are designated as servers if this option is used.

-s

Specifies that the host is a server. This is the default. A host can be both a client and a server.

-s0

Specifies that the host is not a server.

-seed=<integer>

Specifies a seed to be used by that host to begin the random number function that determines the distribution of traffic connections. Usually, this option is not used. Instead, a global seed is specified on the command line, and this seed is automatically modified by the host name to produce one that is unique for each simulated host. See the -seed command-line option.

-showTraffic

When the simulation ends, prints some summary information about the traffic for that simulation; this is the default if there are less than 10 traffic hosts. Otherwise, this option must be specified for each host from which the summary is desired.

-trafficTime=<integer>[unit]

Stops the traffic test after it has been allowed to run for the specified number of time units. The optional unit can be us (microseconds), ms (milliseconds), or s (seconds). The default unit is seconds.

-trafficTrace=<integer>[:<integer]

Saves trace information generated by the traffic protocols. The meanings of the integer arguments are the same as those of the -ethdTrace option. (See Section 3.5.2.)

4 Another Example xsim.data File

This section presents a more complex xsim.data file that is included with the distribution. The purpose of this section is to explain this file, which can be used to test protocols in a realistic internetwork environment, as well as to give a flavor of the complex internetworks that can be simulated using *x*-sim. One word of warning: for a simulation of this complexity, it takes about one minute of real time to run 1 second of simulated time on a 200MHz workstation.

The xsim.data file on which this section is based is located in

```
/xkernel/simulator/build/Template/example/xsim.data.ANS
```

where /xkernel is the root of the *x*-kernel source tree. This canned configuration file represents a subset of the ANSnet backbone, a 45Mbps nationwide public data network.[2]

[2] More information on ANSnet can be found at: http://www.ans.net/

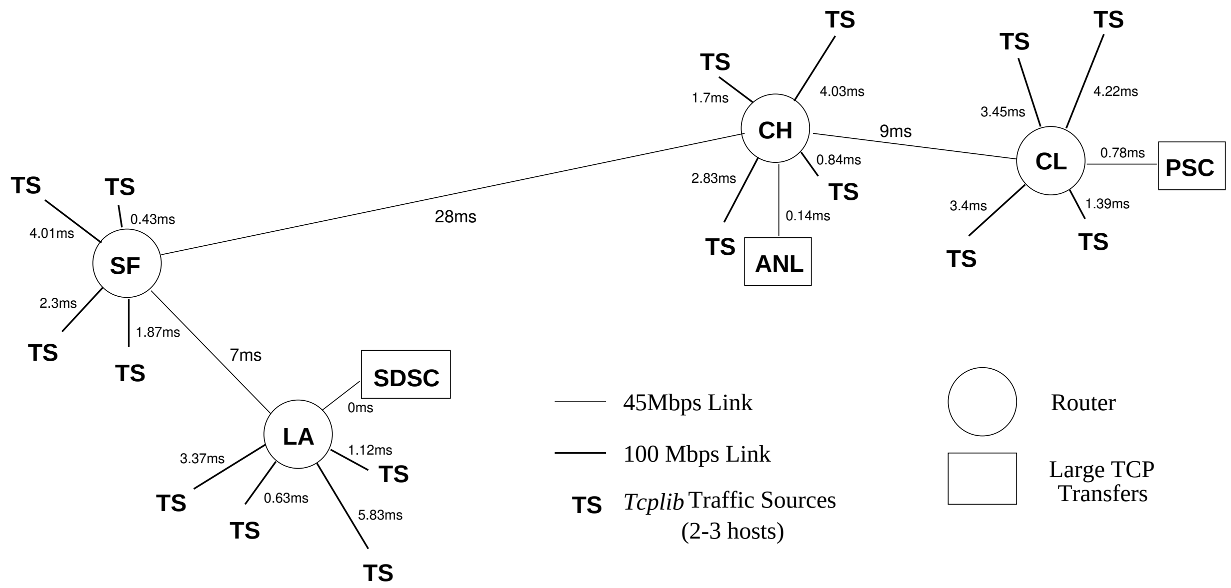

Figure 6: Subset of the ANS backbone

Figure 6 shows a diagram of the internetwork specified in the file. Four routers, marked by circles in the figure, correspond to routers located in Los Angeles, San Francisco, Chicago and Cleveland. The boxes represent the supercomputer centers in San Diego and Pittsburgh, and the Argonne National Labs site near Chicago; large "bulk data" transfers take place between these three sites. The 16 nodes marked **TS** are traffic sources, each consisting of two or three hosts. The resulting traffic consists of nearly a thousand new connections being established per simulated second, where each connection is either a TELNET, FTP, SMTP or NNTP conversation. The bulk data transfers between supercomputer centers must compete for network resources with all of these traffic conversations.

The `xsim.data` file specifying this network is large, but easy to understand. This section gives only snippets of that file; see the actual file for the complete story. There are only a few types of components being configured: 45Mbps and 100Mbps links, routers, bulk transfer hosts, and traffic hosts. Each type of component resembles others of its type; in other words, all 100Mbps links look almost the same. The remainder of this section gives examples showing how an individual component of each type is specified in the `xsim.data` file.

4.1 Networks

Figure 6 shows 100Mbps links connecting the traffic sources to the routers, and 45Mbps links connecting the routers to each other and to the supercomputer sites. The 45Mbps links running between routers are configured as point-to-point links; the 45Mbps links connecting the supercomputing centers to the routers are parameterized Ethernets. We call these networks "parameterized" Ethernets because the parmeters we specify in the `xsim.data` file change the characteristics of these networks from those of the default 10Mbps Ethernet.

The time associated with each link in the figure represent latency times for packets traveling across the link; this latency is proportional to distance. From San Francisco, it is much farther to

Chicago than to Los Angeles, so the latency to Chicago (28ms) is four times as great as that to L.A. (7ms).

We now give an example specification for each kind of link, as it appears in `xsim.data`. The entries specifying networks are found at the end of the file. First, the 45Mbps point-to-point link from San Francisco to Chicago:

```
set prate = 45Mb;
set pdelay1 = 28ms;

net PP 192.0.51.0;
  connections = SF 0.0.0.1, CH 0.0.0.2;
  args = rate=$prate, delay=$pdelay1;
```

This entry is straighforward. The first line identifies it as a point-to-point link with network address 192.0.51.0; the two routers SF and CH attach at addresses 192.0.51.1 and 192.0.51.2, respectively. The bandwidth and latency of the link are set by the `rate` and `delay` parameters.

Next, we show how the parameterized Ethernets that connect the supercomputer centers to the routers are configured. This one connects the router in Cleveland to the Pittsburgh Supercomputing Center:

```
set erateBig = 45Mbps;
set edelay42 = 780us;
set emaxpkt = 1500;
set ehdrsize = 12;
set coll = off;

net ETH 192.0.42.0;
  connections = CL 0.0.0.100, h1n42 ethd 0.0.0.1;
  args = rate=$erateBig, delay=$edelay42, maxPacket=$emaxpkt, hdrSize=$ehdrs
         collisions=$coll;
```

This network is identified as an Ethernet, with net address 192.0.42.0. We change its bandwidth from the default 10Mbps to 45Mbps, and specify a latency of 780 microseconds. The CL router connects at address 192.0.42.100; host `h1n42`, a host at the PSC that will engage in bulk transfers, has address 192.0.42.1.

Finally, each of the traffic sources is connected to a router by a 100Mbps parameterized Ethernet. Here is an entry for a network that connects a traffic source consisting of three hosts to the router in L.A.:

```
set erate = 100Mbps;
set edelay1 = 1120us;
set emaxpkt = 1500;
set ehdrsize = 12;
set coll = off;

net ETH 192.0.1.0;
```

```
  connections = LA 0.0.0.100, h1n1 ethd 0.0.0.1, h2n1 ethd 0.0.0.2,
            h3n1 ethd 0.0.0.3;
  args = rate=$erate, delay=$edelay1, maxPacket=$emaxpkt, hdrSize=$ehdrsize,
         collisions=$coll;
```

This entry is almost the same as the one given above for the 45Mbps parameterized Ethernet. The three hosts `h1n1`, `h2n1`, and `h3n1` will generate traffic connections during the simulation; together they make up a single "traffic source" in Figure 6.

4.2 Routers

There are four routers shown in Figure 6. Below is the entry that defines the router at San Francisco:

```
set rbufs = 60;

router FCFS SF;
  rtable = 192.0.5.0 192.0.5.0, 192.0.6.0 192.0.6.0, 192.0.7.0 192.0.7.0,
           192.0.8.0 192.0.8.0, 192.0.1.0 192.0.50.1, 192.0.2.0 192.0.50.1,
           192.0.3.0 192.0.50.1, 192.0.4.0 192.0.50.1, 192.0.40.0 192.0.50.1,
           0.0.0.0 192.0.51.2;
  args = buf=$rbufs, timeInc=10000, shortTrace=20000:4,
         traceQueue=192.0.50.2, traceInOut=192.0.50.2, traceOut=192.0.50.2,
         traceQueueLen=192.0.50.2, traceQueue=192.0.51.1, traceInOut=192.0.51.1,
         traceOut=192.0.51.1, traceQueueLen=192.0.51.1;
```

The routing table given on the `rtable` line is large but simple. It specifies that

- Packets destined for hosts on networks 192.0.5.0, 192.0.6.0, 192.0.7.0, and 192.0.8.0 are sent to the appropriate host. The router is directly connected to these networks.
- Packets destined for networks 192.0.1.0, 192.0.2.0, 192.0.3.0, and 192.0.4.0 are sent out on network 192.0.50.0 to address 192.0.50.1 (this is the L.A. router).
- All other packets are sent out on network 192.0.51.0 to address 192.0.51.2 (this is the Chicago router).

The first arguments listed configure the router with 60 buffers; the `qmin` and `qmax` arguments have no effect at the present time, but will be included in a future release. The rest of the arguments are trace options. They specify that the router should trace activity on the network interfaces with addresses 192.0.50.2 and 192.0.51.1; packets sent out on these interfaces are destined for the routers in L.A. and Chicago. Information that will be traced for each interface includes the minimum, maximum, and average send queue size in packets and bytes; the average number of bytes/second forwarded to the interface; and the average number of bytes/second sent out on the interface. Also, between seconds 5 and 8 of the simulation, trace information for each packet sent on interface 192.0.51.1 will be saved.

All of the other routers resemble this one. All routers trace queue size and average bytes/second for the interfaces connected to the point-to-point links; the packets sent on these interfaces are destined for another router. However, only the SF router uses the `pktTrace` option to trace individual packets.

4.3 Hosts

There are three types of hosts in the internetwork: hosts initiating a bulk transfer, hosts receiving a bulk transfer, and hosts generating traffic connections.

The hosts engaged in bulk transfers are at the supercomputer sites. Below, host `h1n40` is at the SDSC and will initiate a transfer; host `h1n42` is at the PSC and will receive the data sent by host `h1n40`.

```
set ip = ip, ip vnet, vnet eth arp, arp eth, eth ethd, ethd sim;
set tcpMegS = vtcp;
set tcpMegR = btcp;
set buf1 = 400;
set len1 = 1024;
set tcpTrace1 = -tcpTrace=350000:5;
set thresh1 = 200;
set ed = 0;
set delay1 = 10s;
set megTime1 = 12s;
set db = db/junk -tcpTimerAuto ;
set dbStr1 = -dbStr=rbufs:$rbufs;
set dbStr2 = -dbStr=group:$group;
set dbStr3 = -dbStr=qmin:$qmin;
set dbStr4 = -dbStr=qmax:$qmax;

host h1n40;
  protocols = megtest $tcpMegS, $tcpMegS $ip;
  args = -c192.0.42.1 -buf=$buf1 -len=$len1 $tcpTrace1 -thresh=$thresh1
         -megTime=$megTime1 -delay=$delay1 -dbStr=delay:$delay1
         -db=$db -timeInc=10000 -ethdTraceOut -ethdTraceIn $dbStr1 $dbStr2
         $dbStr3 $dbStr4;

host h1n42;
  protocols = megtest $tcpMegR, $tcpMegR $ip;
  args = -s -len=$len1 -buf=$buf1 -timeInc=10000 -ethdTraceIn
         -ethdTraceOut;
```

The protocol stacks for both hosts are straightforward; note, however, that `h1n40` will use TCP Vegas as its transport protocol, while `h1n42` will run Big Window TCP.

Let's look at the more significant arguments for the hosts in detail. For explanations of arguments not discussed, see Section 3.5.

- The `-c192.0.42.1` parameter indicates that `h1n40` will initiate the bulk transfer to the specified address. If you look at the entry for network 192.0.42.0, you will see that this is the address of `h1n42`, which is given the `-s` argument.

- `-buf` sets the maximum size of the TCP send buffer to 400KB for both hosts.

- `-len` specifies that 1024KB of data should be transferred.
- `-thresh` sets the initial threshold for TCP Slow Start at 200KB.
- `-delay` specifies that the host should wait 10 seconds before starting the transfer, and `megTime` says that the transfer should be cut off at 12 seconds. Thus, the transfer is given 2 seconds to run.
- `-ethdTraceOut` and `-ethdTraceIn` trace the average number of bytes sent and received, respectively, by the host.
- `-tcpTrace` will trace significant TCP events, such as the transmission of a segment, the size of the send window, and the firing of a timeout.

The other kind of host on our internetwork generates traffic connections. Each of these hosts is both a client and a server.

```
set ip = ip, ip vnet, vnet eth arp, arp eth, eth ethd, ethd sim;
set tcpTraff = btcp;
set begcit = 25;
set begdur = 7 -begint=2;
set cit2 = 90;
set db = db/junk -tcpTimerAuto ;
set ed = 0;
set time = 15;
set ethdTrace = ;
set dbStr1 = -dbStr=rbufs:$rbufs;
set dbStr2 = -dbStr=group:$group;
set dbStr3 = -dbStr=qmin:$qmin;

host h1n1;
  protocols = traffic $tcpTraff, $tcpTraff $ip;
  args = -hosts=3 192.0.5.1 192.0.9.1 192.0.13.1 -s1 -begcit=$begcit
         -begdur=$begdur -begkeep=200 -cit=$cit2 -db=$db -trafficTime=$time
         -ethdDelay=$ed $ethdTrace $dbStr1 $dbStr2 $dbStr3 -reset=10;
```

There is nothing surprising about the protocol stack on this host. Big Window TCP is used as the transport protocol. Again, let's examine the arguments.

- The `-hosts=3 192.0.5.1 192.0.9.1 192.0.13.1` argument specifies that this traffic host will act as a client, and open traffic connections with the three hosts specified. Note that host 192.0.5.1 is near San Francisco, 192.0.9.1 is near Chicago, and 192.0.13.1 is near Cleveland; thus, packets sent to these hosts will have to traverse the point-to-point links between routers. The `-s1` parameter tells the host to accept connections initiated by other hosts.
- The `-begcit`, `-begdur`, `-begint`, and `-cit` arguments all have to do with how often *tcplib* opens connections during the simulation. See section 3.5 for details.

- `-trafficTime` specifies to stop the traffic host after it has run for 15 seconds.
- `-reset` says to reset the trace information after 10 seconds. The bulk transfers start running then, and that's when the traces will get interesting.

5 Building *x*-Sim

This section describes the procedures that should be followed to install, configure, build the *x*-kernel, and build and run *x*-Sim as a Unix process. We assume the following throughout the rest of this note.

- You will be installing the *x*-kernel in directory `/usr/xkernel`, and this directory is reachable (via NSF, for example) from all machines on which you want to run *x*-Sim. You may choose some root directory other than `/usr/xkernel`. If you do select another root directory, change the steps given below accordingly.
- You will be running *x*-Sim on a Solaris (Sparc) workstation. If you are building *x*-Sim on some other Unix platform—SunOS (on Sparc) or OSF/1 (on Alpha)—then substitute `sunos` or `osf1` for `solaris`, respectively, in the steps that follow.

5.1 The Steps

1. Create directory `/usr/xkernel` and use anonymous FTP to copy file `xkernel.tar.Z` from `cs.arizona.edu/xkernel` into this directory. Then type:

   ```
   cd /usr/xkernel
   uncompress xkernel.tar.Z
   tar xf xkernel.tar
   ```

   This will create several subdirectories in `/usr/xkernel`. A description of the *x*-kernel directory hierarchy can be found in `/usr/xkernel/README`. Note that you can skip this step if you are not the first person to install and use the *x*-kernel or *x*-Sim at your site.

2. Create a directory in which you will build an instance of *x*-Sim. This is called your build directory, and in general, each different configuration of the *x*-kernel and *x*-Sim has its own build directory.

   ```
   cd /usr/xkernel/simulator/build
   mkdir solaris
   ```

   If directory `/usr/xkernel/simulator/build/solaris` already exists—because someone else has already followed this procedure—then you have two choices.

   (a) Select another name for your build directory (e.g., `mydir`) and create a directory with this name in directory `/usr/xkernel/simulator/build`. The catch to this option is that you have to either have write permission in directory `/usr/xkernel/simulator/build`, or you have to have someone that has write permission create your build directory for you.

(b) Create a build directory in your own home directory; e.g., `/usr/llp/xsim/solaris`. This is the preferred approach if a lot of different people are going to be building the *x*-kernel or *x*-Sim.

Substitute the name of your build directory, as appropriate, in the rest of the steps.

3. Copy configuration files into your build directory.

```
cd /usr/xkernel/simulator/build/solaris
cp ../Template/Makefile.solaris Makefile
cp ../Template/example/* .
mkdir db
chmod 664 *
```

   Note that if your build directory is not in `/usr/xkernel/simulator/build`, then you will have to use the full path names for these various files rather than the relative path names given above (i.e., `/usr/xkernel/simulator/build/Template/example` instead of `../Template/example`).

4. Edit the "`XRT =`" line of the `Makefile` you just copied into your build directory to reflect the root of your *x*-kernel source tree. Assuming your tree is at `/usr/xkernel`, this means changing

```
XRT = ../../..
```

   to

```
XRT = /usr/xkernel
```

   Note that this step is necessary only if your build directory is underneath your home directory. It is not necessary if your build directory is in `/usr/xkernel/simulator/build` since from there, `../../..` already points to the root of the *x*-kernel source tree.

5. Put `/usr/xkernel/bin/solaris-sparc` and `/usr/xkernel/bin` in your search path. They should appear before `/bin` and `/usr/bin` in order to pick up GNU `make` before the standard Unix `make`. If you are building *x*-Sim on a platform other than Solaris, then substitute `sunos-sparc` or `osf1-alpha` for `solaris-sparc`, as appropriate. Also, make sure the GNU C compiler (`gcc`) is in your search path. (At Arizona, `gcc` can be found in directory `/usr/local/bin`.)

6. Build libraries necessary for your *x*-Sim. This could take quite a while.

```
cd /usr/xkernel/simulator/build/solaris
make system
```

   Note that you can skip this step if you are not the first person to build an *x*-Sim for this platform.

7. You are now ready to build your *x*-Sim.

```
cd /usr/xkernel/simulator/build/solaris
make compose
make depend
make
```

This sequence should result in the creation of the binary file `xkernel` in your build directory. Note that you may see what appear to be error messages about missing files (e.g., `Makefile.local` and `DEPS/Makedep.*`) when you run `make compose`. These warnings can be ignored.

8. You are now ready to run your *x*-Sim.

```
cd /usr/xkernel/simulator/build/solaris
xsim
```

5.2 Debugging

There are some debugging aids built in to *x*-Sim. If you type

```
xsim -h
```

x-Sim will print the following help documentation:

```
Help for debugging xsim:

  xsim  -xsimdbg  E:<Event codes>  S:<Simulator codes>  T:<Test codes>

    where code strings are 2-letter, event specific codes

         Event codes:
           IQ  show event queue inserts
           RQ  show event queue removals
           SE  show stub executions
           ES  show evSchedule calls
           CI  show clock interupt handler execution
           SQ  show full queue in IQ and RQ
           FN  use function names when listing
           AL  equivalent to all Event codes

         Simulator codes:
           ED  show ethd stuff
           L0  show very detailed simulator stuff
           L1  show detailed simulator stuff
           L2  show coarse simulator stuff
           PC  show packet collision stuff
           AL  equivalent to all Simulator codes
```

```
Test codes:
  ET  show ethtest stuff
  MG  show megtest stuff
  TR  show traffic stuff
  AL  equivalent to all Test codes

Examples:     xsim -xsimdbg E:ESCIFN S:AL T:MG
          or  xsim -xsimdbg E:ES,CI,FN S:AL T:MG
```

6 Output

The `megtest` and `traffic` protocols print some general information to the display while the simulator is running. The simulated time elapsed is shown, along with messages from individual hosts about their states and error conditions. At the completion of a run, summary information is printed about the `megtest` transfer or `traffic` conversations.

The most interesting and important output generated by a simulation, however, takes the form of trace files. The protocols and routers included with the simulator have had trace statements inserted at places in the code where significant events occur; each trace statement writes a few bytes to a trace file, indicating the event and recording relevant state. See the "Data Trace Library" section of the *x*-kernel Programmer's Manual for information on the this tracing facility [NSR96]. Also, see the code for the TCP Vegas protocol in `/xkernel/simulator/protocols/vtcp` for an example of how to augment a protocol with trace statements. The file `README.trace` in this directory lists all of the protocol's trace statements, and explains where each occurs and what information it saves. For an example of how to add trace statements to a driver protocol, see `/xkernel/simulator/sim/ethd`.

Two graphical display tools have been developed to convert the trace files generated by a simulation into readable PostScript graphs. They are `ptcp` and `extrace`, and both can be found in `/xkernel/bin/solaris-sparc`, `/xkernel/bin/sunos-sparc` or `/xkernel/bin/osf1-alpha`, depending on the system you are using. The `ptcp` tool generates TCP-specific graphs, displaying such information as the size of the send window and the time when a packet was retransmitted. Graphs produced by `extrace` show changes in queue sizes on the routers. For an explanation of how to interpret the output from these two tools, see Section 7 and Section 8.

7 Ptcp

Ptcp is a graphical analysis tool used to analyze trace output from *x*-Sim simulations. The ptcp can be used as follows.

Invoking the "ptcp" tool:

```
ptcp [arguments] <plot types> trace-filename
```

plot types:

```
win
```

Plot different windows (congestion, etc.)

rate

Plot the sending rate.

vegas

Plot Vegas information

rtt

Plot RTT.

drtt

Plot RTT differences.

seq

Plot sequence numbers.

Args:

-t

Starts graphs at time 0 (when the simulator started). The default is to start graphing at the time the connection is opened.

-pn

Create page information so postscript browsers, like ghostview, can jump to any page.

-db=<file-name>

To append file with data about the transfer. The entries are in ascii. For further information refer to the database documentation (to be written later).

-sid=<sid>

Used to specify the session (connection) id when there were more than one connection.

-xtime=<time>

Time in ms covered by each graph. If the transfer lasts longer than this time, more than one graph will be created.

-xtics=<time>

Show tic marks in the time axis every ¡time¿ ms.

-xbeg=<time>

Beginning time, in ms, of first plot.

-xend=<time>

Ending time, in ms, of last plot. Xbeg and xend specify the time range to plot.

`-journal`

Create plots suitable for publication. The plots show less features.

`-noEvents`

Don't show events such as packets with old acks, packets with RST, SYN, FIN, timeout KEEP events, etc.

`-journal=<#>`

Similar to -journal, except that it will create an encapsulated postscript file of the right size to hold # plots.

`-header`

Insert header information at the beginning of the file. One can run ptcp multiple times and have each run's plot appended to the same file (see the -a flag). The first time ptcp is ran it should have the -header flag, the last time it is ran it should have the -tail flag).

`-tail`

Include necessary commands at the end of the postscript file.

`-allplot`

This flags is equivalent to the -header and -tail flags.

`-a`

Append this plot to the default output file (trace.ps) or to the file specified with the -o argument.

`-o <filename>`

Use filename as the output file containing the postscript commands to do the plots.

`-scale`

Scale the output by .8 in the y direction

`-n <comment>`

Add the comment at the top of the 1st page.

`-big`

Create big plots suitable for slides. The time axis in each plot will only show 1/2 the time specified with -xtime.

`-maxRate=<rate>`

Set the y axis to maxRate when plotting the sending rate.

`-rateTic=<#>`

Plot tic marks at multiples of # when plotting sending rate.

`-arn=<#>`

How many entries to use when doing running average of sending rate.

`-maxWin=<#>`

Similar to -maxRate, but refers to window plot.

`-winTic=<#>`

Similar to -rateTic, but refers to window plot.

Note: If using only 1 sid, then always use -sid=232

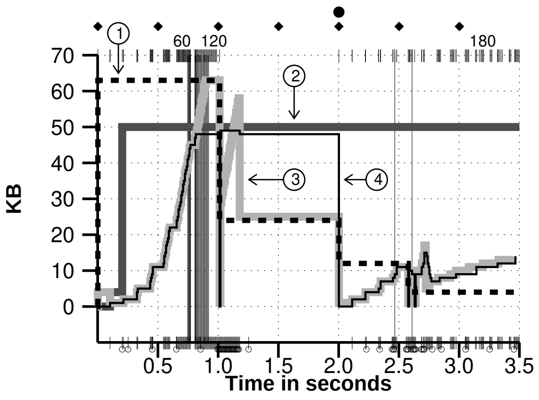

Figure 7: TCP Windows Graph.

To assist the reader develop a better understanding of the graphs produced by ptcp and to gain a better insight of TCP's behavior, we describe in detail one of these graphs. Figure 7 is a trace of Reno when there is other traffic through the a router. The numbers in parenthesis refer to the type of line in the graph.

In general, output is allowed while the UNACK-COUNT (4) (number of bytes sent but not acknowledged) is less than the congestion window (3) and less than the send window (2). The purpose of the congestion window is to prevent substained congestion. The send window is used for flow control; it prevents data from being sent when there is no buffer space availabe at the receiver.

The threshold window (1) is set to the maximum value (64KB) at the beginning of the connection. Soon after the connection is started, both sides exchange information on the size of their respective receive buffers, and the send window (2) is set to the minimum of the sender's send buffer size and the receiver's advertized window size.

The congestion window (3) increases exponentially while it is less than the threshold window (1). At 0.75 seconds, losses start to occur (indicated by the tall vertical lines). More precisely, the vertical lines represent segments that are later retransmited (usually because they were lost). At around 1 second, a loss is detected after receiving 3 duplicate ACKs and Reno's Fast Retransmit and Fast Recovery mechanisms go into action. The purpose of these mechanisms is to detect losses before a retransmit timeout occurs, and to keep the pipe full (we can think of a connection's path as a water pipe, and our goal is to keep it full of water) while recovering from these losses.

The congestion window (3) is set to the maximal allowed segment size (for this connection) and the UNACK-COUNT is set to zero momentarily, allowing the lost segment to be retransmitted. The threshold window (1) is set to half the value that the congestion window had before the losses (it is assumed that this is a safe level, that losses won't occur at this window size).

The congestion window (3) is also set to this value after retransmitting the lost segment, but it increases with each duplicate ACK (segments whose acknowledgement number is the same as previous segments and carry no data or new window information). Since the receiver sends a duplicate ACK when it receives a segment that it cannot acknowledge (because it has not received all previous data), the reception of a duplicate ACK implies that a packet has left the pipe.

This implies that the congestion window (3) will reach the UNACK-COUNT (4) when half the data in transit has been received at the other end. From this point on, the reception of any duplicate ACKs will allow a segment to be sent. This way the pipe can be kept full at half the previous value (since losses occurred at the previous value, it is assumed that the available bandwidth is now only half its previous value). Earlier versions of TCP would begin the slow-start mechanism when losses were detected. This implied that the pipe would almost empty and then fill up again. Reno's mechanism allows it to stay filled.

At around 1.2 seconds, a non-duplicate ACK is received, and the congestion window (3) is set to the value of the threshold window (1). The congestion window was temporarily inflated when duplicate ACKs were received as a mechanism for keeping the pipe full. When a non-duplicate ACK is received, the congestion window is reset to half the value it had when losses occurred.

Since the congestion window (3) is below the UNACK-COUNT (4), no more data can be sent. At 2 seconds, a retransmit timeout occurs (see black circle on top), and data starts to flow again. The congestion window (3) increases exponentially while it is below the threshold window (1). A little before 2.5 seconds, a segment is sent that will later be retransmitted. Skipping to 3 seconds, we notice the congestion window (3) increasing linearly because it is above the threshold window (1).

8 Extrace

Extrace is a graphical analysis tool used to analyze trace output from x-Sim simulations. Extrace is used specifically for analyzing router buffers and ethernet driver traces.

Invoking the "extrace" tool:

```
extrace [arguments] [plot-types] trace-filename
```

plot-types:

```
queue
port0
port1
```

args: Same as in ptcp:

```
-allplot
-header
-tail
-a
-o
-xbeg=
-xend=
-xtime=
-xtics=
-maxRate= (can use T1, Kb and Mb)
-maxTics=
-journal
-pn
-n
-big
-auto1
-auto2
```

new:

`-group=<#>`

Group events into groups of size #. For example, if the queue information of a router was written every 10ms, a grouping of 5 would be equivalend to writing the information every 50ms.

`-bucket=<#>`

Used for the histograms printed out to the screen. Specifies the bucket size of the histogram.

`-reset=<time>`

Reset gathering of statistics, so the numbers shown only represent statistics after time (in ms).

`-db=<filename>`

Writes information to the specified (db) file.

`lenDb[r|w]`

To be documented.

References

[CDJM91] Ramon Careres, Peter B. Danzig, Sugih Jamin, and Danny J. Mitzel. Characteristics of wide-area tcp/ip conversations. In *Proceedings of the SIGCOMM '91 Symposium*, pages 101–112, September 1991.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, August 1993.

[NSR96] Network Systems Research Group, Department of Computer Science, University of Arizona. *x-kernel Programmer's Manual (Version 3.3)*, January 1996.