NAME

troll – network simulator / datagram forwarder

SYNOPSIS

troll [*options*] **port**

```
#include <sys/types.h>
#include <netinet/in.h>
#include "troll.h"
```

DESCRIPTION

Troll is an application designed to simulate the various quirks and vagaries of a network environment. The troll forwards UDP datagrams between processes, possibly dropping, delaying, garbling, and/or duplicating those datagrams.

COMMAND LINE ARGUMENTS

port The UDP port number on which the troll will listen for incoming datagrams. This argument is mandatory, and must be an integer between 1024 and 65535.

The following options override various default parameters. They can also be set by interactive commands, as described below. Options specified as *percent* must be an integer in the range from 0 through 100.

–b Normally, the troll runs in an interactive mode, as described in the following section. The **–b** option disables the interactive mode, and the troll can thus be run more conveniently in the background. For example, *''troll -b 26784 -s0 -x0 &'* will start the troll in the background, listening on UDP port 26784 and neither delaying nor dropping datagrams.

–f*file* The distribution of datagram delays is to be taken from the named file rather than computed from the *delay* parameter. See the description of the **delay file** interactive command below.

–g*percent* The percentage of datagrams that should be garbled (default is 0%).

–m*percent*
The percentage of datagrams that should be duplicated (sent multiple times; default is 0%).

–r Allow reordering. Without this option, delays are all interpreted as relative: Datagrams are queued first-in-first out, with delays between datagrams calculated according to the delay parameter (see *–s*, *–se*, and *–f*). With the *–r* option, delays are absolute. Since each datagram is independently delayed, datagrams may be sent out in a different order than they arrived. See further discussion under **TECHNICAL DETAILS** below.

–s*number*

–se*number*
The separation of datagrams in milliseconds (default is 100; see also *–f*). If **e** is specified, the delay is an exponential random variable with mean *number* . Otherwise, it is a constant.

–x*percent* The percentage of datagrams that should be dropped (default is 10%).

INTERACTIVE MODE

After being started, the troll normally runs in an interactive mode, and will prompt for commands with the message

```
Speak to the troll:
```

A command may be abbreviated to any unambiguous prefix.

drop *percent* Sets the percentage of datagrams that will be dropped (i.e., will not be forwarded).
Note that the troll can buffer only a limited number of datagrams; if a datagram arrives and no buffer space is available, the datagram is dropped. Setting **delay** or **duplicate** to a suffi-

ciently high value may result datagrams being dropped even if the **drop** parameter is zero.

delay *milliseconds*

delay exponential *mean*

delay file *filename*

These commands control the **delay** distribution. Depending on the setting of the **reorder** option, the **delay** parameter controls the delay of datagrams or separation between them. The first form sets **delay** to a specific value. The second sets it to an exponentially distributed random variable with the indicated mean. (Other distributions may be provided by future releases of the troll). The third form sets it to a random variable with a distribution indicated by the histogram contained in the named file. The third form takes the distribution from a file. The first line of the file should contain an integer indicating the precision, in milliseconds. The remainder of the file should have have one line for each possible delay value (in multiples of the precision). The length of each line (not including the newline character at its end) controls the relative frequency of corresponding delays. (The actual contents of the lines are ignored). For example, if file contains

```
100
<two empty lines>
xx
xxxxxx
xxx
x
x
```

then 2/13 of the datagrams will be delayed 200ms, 6/13 will be delayed 300ms, 3/13 400ms, 1/13 500ms, 1/13 600ms, and no datagrams will be delayed less than 200 or more than 600 milliseconds. As a special case, a file with only one non-empty line after the first will generate a deterministic distribution (all datagrams delayed the same amount of time). *Warning:* Although delays are specified in milliseconds, the Unix timing facilities are usually less precise, so delays may actually be longer than specified.

garble *percent* Sets the percent of forwarded datagrams that will be garbled (i.e., have one or more bits changed).

duplicate *percent*

Sets the percent of forwarded datagrams that will be duplicated (i.e., transmitted more than once); of those, *percent* will be duplicated again, etc. Each duplicate of a particular datagram is independently delayed and garbled.

source *filename*

Reads "interactive" commands from *filename*.

The remaining commands do not take any argument.

reorder Toggles the **reorder** option (initially *off* unless set by the **−r** command-line option). If **reorder** is off, the **delay** distribution is used to compute the delay *between* datagrams sent out. If **reorder** is on, the **delay** distribution is used to compute the *total* delay for each datagram (or duplicate). If the **reorder** option is on and **delay** is not a constant, datagrams may not be forwarded in the same order they were received.

parameters Prints the current values of the reliability parameters.

statistics Prints statistics about the number of datagrams dropped, garbled, duplicated, etc. These statistics are also printed on termination.

clearstats Clears all statistics to zero (the startup value).

background (Alias **zap**). Tells the troll to stop accepting interactive commands. This command is useful for putting the troll into the background. You can then type "`^Z`" and "`bg`" to actually put the troll in the background. (See also **−b**).

quit Causes the troll to print statistics and terminate. The same effect is produced by an interactive "command" of `control-D` or by a keyboard interrupt (or `kill -INT`).

trace Toggles the **trace** option. When tracing is enabled, the **troll** prints a message starting with "<" on each datagram arrival, and a message starting with ">" on each datagram transmission.

PROGRAMMING INTERFACE

The data contained in every UDP datagram sent to the troll must begin with a `sockaddr_in` structure, as defined in `<netinet/in.h>`:

```
struct sockaddr_in {
      short sin_family;
      u_short     sin_port;
      struct      in_addr sin_addr;
      char  sin_zero[8];
};
```

The **troll** forwards the datagram to the socket addressed by this field. Before forwarding the datagram, it overwrites this field with the address of the socket from which it came.

Note: All fields must be in network byte order. This this convention differs from system calls (such as `sendto(2)` and `recvfrom(2)`) that expect the `sin_family` field to be in host byte order.

For example, if a program sends datagrams using code similar to

```
struct sockaddr_in dest;
char buffer[BUFFER_SIZE];

dest.sin_family = AF_INET;
dest.sin_addr = /* Internet address of remote machine */;
dest.sin_port = htons(PORT);
      .
      .
      .
/* fill buffer with data */

result = sendto(socket, buffer, sizeof buffer, 0,
                     (struct sockaddr *)dest, sizeof dest);
```

it can be transformed to use the **troll** as follows:

```
struct sockaddr_in dest, troll;
struct {
      struct sockaddr_in header;
      char body[BUFFER_SIZE];
} message;

message.header.sin_family = htons(AF_INET);
message.header.sin_addr = /* Internet address of remote machine */;
message.header.sin_port = htons(PORT);

troll.sin_family = AF_INET;
troll.sin_addr = /* Internet address of machine running the troll */;
troll.sin_port = htons(TROLL_PORT);

/* fill buffer with data */

result = sendto(socket, (char *)&message, sizeof message, 0,
                     (struct sockaddr *)troll, sizeof troll);
```

When the datagram arrives, the header is in the correct format for a reply. Thus a simple "echo server" has the outline

```
for(;;) {
      struct sockaddr_in trolladdr;
      int len = sizeof trolladdr;
      NetMessage message;

      /* read in one message from the troll */
      n = recvfrom(sock, (char *)&message, sizeof message, 0,
                  (struct sockaddr *)&trolladdr, &len);
      if (n<0) {
            perror("fromtroll recvfrom");
            exit(1);
      }
      printf("troll [%s,%d]",
            inet_ntoa(trolladdr.sin_addr), ntohs(trolladdr.sin_port));
      printf("source [%s,%d]0,
            inet_ntoa(message.msg_header.sin_addr),
            ntohs(message.msg_header.sin_port));
      n = sendto(sock, (char *)&message, sizeof message, 0,
                              (struct sockaddr *)&trolladdr, len);
      if (n!=sizeof message) {
            perror("fromtroll sendto");
            exit(1);
```

`troll.c` , as well as test programs `totroll.c` and `fromtroll.c`, are available on the course home page.

TECHNICAL DETAILS

Whenever a datagram arrives, the troll examines the header to see whether the `sockaddr` there meets minimal sanity checks (it requires that `ntohs(header.sin_family)` is `AF_INET` and that `htohs(header.sin_port)` is in the range `1024...65535` ; it does not check whether `header.sin_addr` is reasonable). If not, it increments a counter and drops the datagram. Otherwise, it records the destination address from the header and overwrites it with the socket address of the sender.

Next, the troll flips a coin, and with probability **drop** it throws away the datagram (incrementing a counter). Otherwise, it flips another coin and with probability **garble** garbles the datagram. The garbling algorithm chooses 5 to 10 bytes of the datagram and XORs them with a random pattern. The bytes are chosen randomly, except that one garbled byte is guaranteed to be among the first 10 bytes of the datagram. The return address is not garbled.

Next, the troll calculates when to transmit the datagram and schedules a timer event to send it. A random value is computed according to the current **delay** settings. If the **reorder** parameter is *on*, the dispatch time is calculated by adding the random value to the to the current time. Otherwise, it is calculated to adding the random value to the latest dispatch time any datagram currently queued for delivery. The timer queue is maintained such that requests to send two or more datagrams at the same time are satisfied first-come-first-served.

After the troll queues a datagram for delivery, it flips a coin and with probability **duplicate** makes a copy of the datagram. The copy is treated as an independent arrival and duplicated, delayed, garbled, etc. as described above.

SURGEON GENERAL'S WARNING

Since UDP does no flow control, datagrams will be dropped if you send them to the troll too fast, or if you don't receive them fast enough from the troll. Thus datagrams may be dropped even if the troll parameters are set for ''perfect reliability'' (all probabilities 0).

BUGS

There should be more convenient ways to specify other distributions of delay (such as normal or hyper-exponential) and to specify distributions for other random variables. There should be some way to make the delay depend in complicated ways on the number of datagrams currently queued, to simulate congestion in the network.

AUTHORS

The troll was originally written by Mitchell Tasman. It has been extensively modified and enhanced by Marvin Solomon.