Delayline: A Wide-Area Network Emulation Tool

David B. Ingham and Graham D. Parrington
University of Newcastle

ABSTRACT: Programming distributed applications is already potentially difficult, although tools exist to aid in their creation. Applications developed to communicate over local area networks can expect to enjoy highly reliable communications with minimal latency. However, such applications may no longer continue to perform as expected if moved to a wide-area network environment, due to the reduced performance inherent in wide-area networking. During application development it is often impractical or inconvenient to evaluate application performance under real wide-area network conditions. In this paper we describe the design and implementation of the Delayline tool which provides a fully configurable mechanism for emulating wide-area network communications on a local area network. Delayline allows real distributed applications to be evaluated under emulated wide-area network communication characteristics.

1. Introduction

Computer networks vary widely in scale and performance, from small local networks connecting a few machines to those spanning the globe providing connectivity between millions of machines. Similarly, there are also large variations in the scale of distributed applications that are connected by such networks. Widely distributed applications must typically tolerate reduced network performance in comparison to those communicating over a local network.

From the point of view of the application programmer, the difference between wide- and local-area network programming is principally observed in two ways, namely, an increased communication latency and a higher probability of message loss. There is also a greater chance of message corruption, but this is masked from the application by the error checking in the low-level network protocols. As the scale of networks increases, there is also an increased likelihood of network partitioning, typically caused by a gateway failure. This causes the original single network to become two or more subnetworks.

This paper describes the design and implementation of the Delayline system, an application-level emulation tool, which is capable of changing the observed characteristics of the underlying network supporting an application. Using Delayline, a set of locally connected hosts, running a distributed application, can be made to appear as if they are connected over a wide-area network with user-defined topology and communication characteristics. To achieve this, Delayline provides a mechanism for describing a map of the required virtual-network, including the parameters for the interhost communications.

Delayline has been successfully used with a number of large applications based on the Arjuna distributed programming system [Shrivastava et al. 1971], where it has provided useful information for tuning the underlying remote procedure call subsystem of Arjuna.

2. System Overview

Delayline has been developed in C on a UNIX platform and is designed to operate with applications that communicate using Berkeley sockets, specifically those that use the Internet protocol suite [Comer 1988]. Delayline is nonobtrusive; that is, it has been designed to be easily incorporated into applications with the minimum effort from the programmer. To this end, Delayline requires no source-code modifications and runs on standard unmodified UNIX systems. To use Delayline, an application is simply recompiled, using a number of alternative header files, and relinked with the Delayline library provided.

Wide-area network emulation is achieved in a number of stages:

- Configuration: A mechanism is required to allow the specification of the topology and communication parameters of the network to be emulated.
- Interception: In order to perform any form of manipulation of interprocess communications, the messages themselves must be intercepted as they are transmitted from one process to another.
- Decision: Once a message has been intercepted, the decision must be made as to what action, if any, should be performed on the message.
- Manipulation: After the required action has been ascertained, this must be carried out. A major constraint here, which makes the operation more complex than is readily apparent, is ensuring that any manipulation still preserves the semantics of the Berkeley socket interface.

2.1. Configuration

Delayline provides a mechanism to allow the user to specify the type of emulation that is required. Configuration is a multistage process. First, a map of the desired network must be described. Delayline uses the concept of *host groups* for describing the desired network topology. For example, Figure 1 illustrates an example distributed system consisting of six hosts located at two physically remote sites. At each site, the hosts are interconnected using a local-area network; communication between the two sites takes place over a wide-area network, to which both of the LANs are connected. This system could be described by making hosts A, B, and C members of Delayline group 1 and hosts D, E, and F members of group 2.

Once the map has been decided upon, the next stage is to describe the type of communication that exists between the groups. To this end the user is able to create a set of named emulation styles that are described in terms of a number

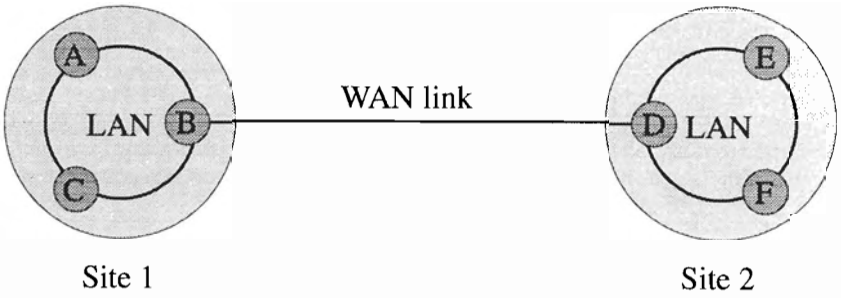

Figure 1. Example of an emulated network topology.

of user-specified performance parameters. These figures comprise the percentage of messages to be lost, the distribution type for message delay (e.g., fixed, normal, gamma, etc.) and the distribution parameters (e.g., the mean and standard deviation in the case of a normal distribution). The user then specifies the type of emulation to be applied to messages being sent from one group to another group; for example, messages from group 1 to group 2 should be subjected to a certain named emulation type.

For a consistent emulation to be achieved, all of the processes that comprise the distributed application must have the same view of the network topology to be emulated. To achieve this consistency Delayline maintains its configuration information at a single host in the distributed system. A configuration server conveys configuration information to the other processes in the system that are running Delayline using remote procedure calls. The application processes contact the server the first time that a Delayline routine is executed. At this time the configuration information relevant to that particular process is received from the server and stored locally.

After a message has been intercepted by Delayline, the sending and receiving hosts are determined and the local copy of the network emulation map is referenced to determine the communication parameters to be applied to the particular message.

2.2. *Interception*

Interprocess communication using the Berkeley socket application programming interface is achieved by sending and receiving messages over sockets. There are five system calls that may be used to send data over sockets and a corresponding set of five for receiving data. By intercepting all of the application program's invocations of these system calls, the interprocess communication can be manipulated in such a way so as to provide the required network emulation.

Using compile-time switches, an application can be forced to use a set of alternative header files in preference to the standard ones. The entry points into the Delayline system are created by redefining the necessary system call-entry routines to Delayline functions. In this manner, Delayline is able to monitor the invocations of the relevant system calls and to view the parameters to the calls, which is a technique similar to that employed in the Newcastle Connection [Brownbridge 1982; Black et al. 1987]. The actual system calls may be invoked by Delayline as required. Figure 2 illustrates this integration approach.

Another possible approach to message interception would be to integrate the Delayline functionality into the UNIX kernel, which would allow applications to be subjected to Delayline emulation without requiring recompilation. Moreover, all networking applications running on the machine would be affected by Delayline. Alternatively, on host systems on which they are supported, shared libraries could be used to provide alternative versions of the relevant system-call routines for sending and receiving messages.

Although the current approach requires that the source code of the application be available, this is not thought to be too severe a restriction since the tool is designed to be used during the evaluation stages of application development. This chosen technique has the advantage of being portable between heterogeneous machines and does not rely on kernel modifications or shared library support.

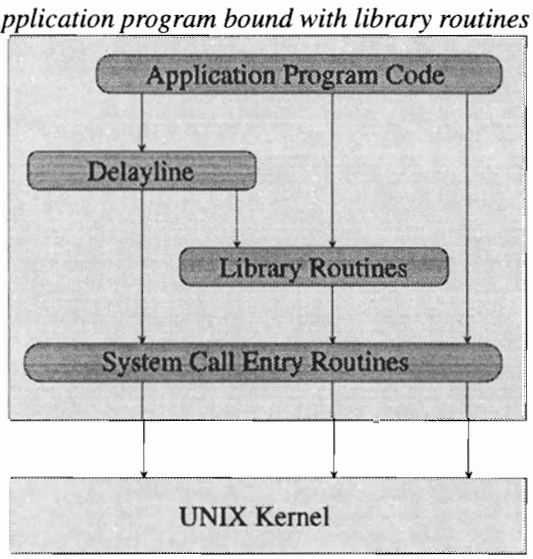

Figure 2. Integration of Delayline into an application program.

2.3. *Decision*

After intercepting a message, the source and destination hosts are determined and the network emulation map is referenced to determine the parameters to be applied to such messages. For example, the parameters for a particular link may be 5 percent probability of message loss and normally distributed message delays with mean 500 ms and standard deviation 100 ms. The decision as to what action should be taken for a particular message—lose message or delay message by 400 ms—is made by using statistical functions based on the given parameters. The protocol for the communication also affects the action to be carried out; for example, messages using the reliable TCP protocol are never lost.

2.4. *Manipulation*

The aim of the Delayline system is to mimic as closely as possible the interprocess communication performance of the desired wide-area network. To achieve this goal the system is able to introduce message loss and delay into the communications. The main constraint in performing this manipulation is the requirement that the semantics of the standard socket interface must be preserved. The following sections describe the techniques that are used by Delayline to provide the required message manipulation.

3. *Emulation of Message Loss*

As stated previously, one of the primary differences between local and wide-area network communications is a higher probability of message loss. Delayline achieves message-loss emulation by manipulating the application program's invocations of the sending system calls. Consider the C language prototype of the send system call:

```
int send(int sockfd, char *buff, int nbytes, int flags)
```

The first parameter indicates the socket descriptor to use, the address of the data to be sent is specified by `buff`, and the length of the data is given in `nbytes`. The final argument, `flags`, allows certain transmission options to be set. The call returns the number of bytes that have been transmitted.

The header files provided by Delayline redefine all application program instances of the system calls used for sending data (e.g., `send`, `sendto`, etc.) to

Delayline versions (e.g., `dl_send`, `dl_sendto`, etc.). The code segment to follow illustrates, in a simplistic fashion, how message loss is implemented in the Delayline sending routines.

```
int dl_send(int dl_sockfd, char *dl_buff, int dl_nbytes, int dl_flags)
{
   ...
   if (action == LOSE_MESSAGE)
     return(dl_nbytes);
   else
     return( send(dl_sockfd, dl_buff, dl_nbytes, dl_flags) );
}
```

The application program receives the expected return value but the actual send is not performed. The result is semantically equivalent to the message being lost during its journey from source to destination. A slight additional complexity arises from the use of the `writev`, `readv`, `sendmsg` and `recvmsg` system calls which use scatter/gather buffers to provide the ability to read into or write from noncontiguous buffers. For these routines, the total number of bytes in each of the buffers is calculated and returned to the application program.

4. *Emulating Increased Latency*

Introducing latency into interprocess communication in a semantically correct manner is somewhat harder to achieve than the emulation of message loss. Consider two communicating processes, A and B. The diagram in Figure 3 shows the situation in which process A is attempting to send a message to process B.

When process A executes the `send` system call, the data to be sent is copied into kernel buffer space and queued to be sent. At this point, the kernel returns control to the application program, allowing it to continue. When process B executes the `recv` call, control is passed to the kernel and, under normal circumstances, the application is blocked until the data arrives, at which point the data received is passed to the application and control is returned.

The time taken for the message to travel from process A to process B is influenced by a number of factors including the input/output scheduling policy of the operating system. However, the primary dependency is the type of communication link connecting the two hosts. If the two processes are running on hosts connected by a LAN, the communication will typically be very fast, in the order of milliseconds (dotted line in diagram), while a wide-area connection could result in a much slower delivery, typically in the order of seconds. In order to emulate

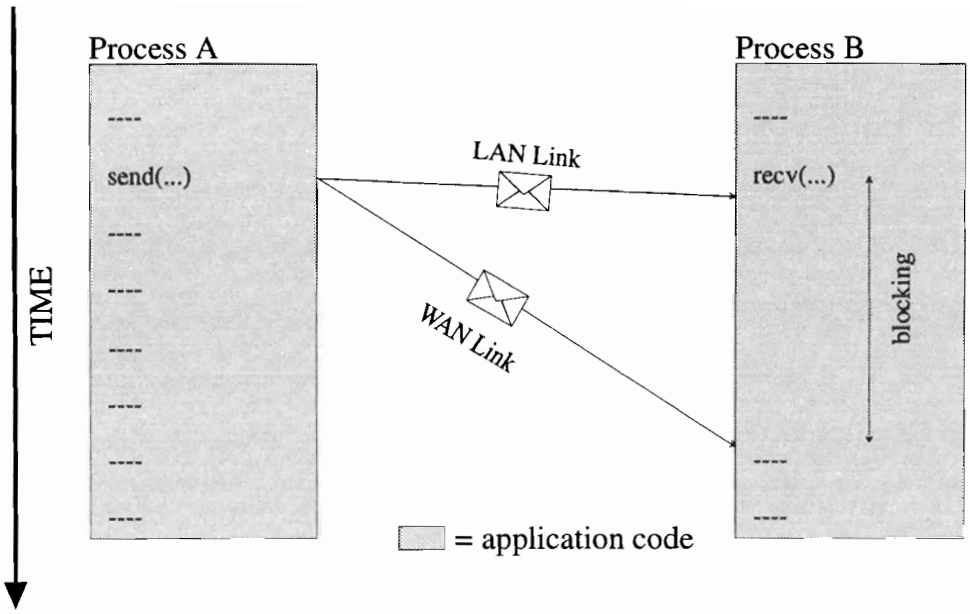

Figure 3. Interprocess communication.

wide-area communications, this increased communication delay must be inserted by the Delayline software, while preserving the correct semantics of the Berkeley socket interface.

In the current version of the Delayline software, latency is introduced by intercepting messages in both the sending and receiving phases of the interprocess communication. As messages to be sent are intercepted by Delayline, they are *signed*, indicating that they should be delayed on reception. The process of signing the messages involves prepending the messages with a fixed-length header containing a *Delayline signature* and the required delay in milliseconds. The revised buffer is sent to its destination using the appropriate system call with modified parameters, indicating the differing size of the buffer. The return value from the system call is modified to take into account the additional header before being passed back to the application program. This technique allows a semantically correct flow of control in the sender, because control can be returned to the application as soon as the sending-system call returns from the kernel. These control-flow semantics are described further later.

The system calls for receiving data through sockets are also remapped to Delayline functions so as to allow the examination and removal of the prepended signatures before passing data to the application program. When these calls are executed by an application, control is passed to the kernel and the application is

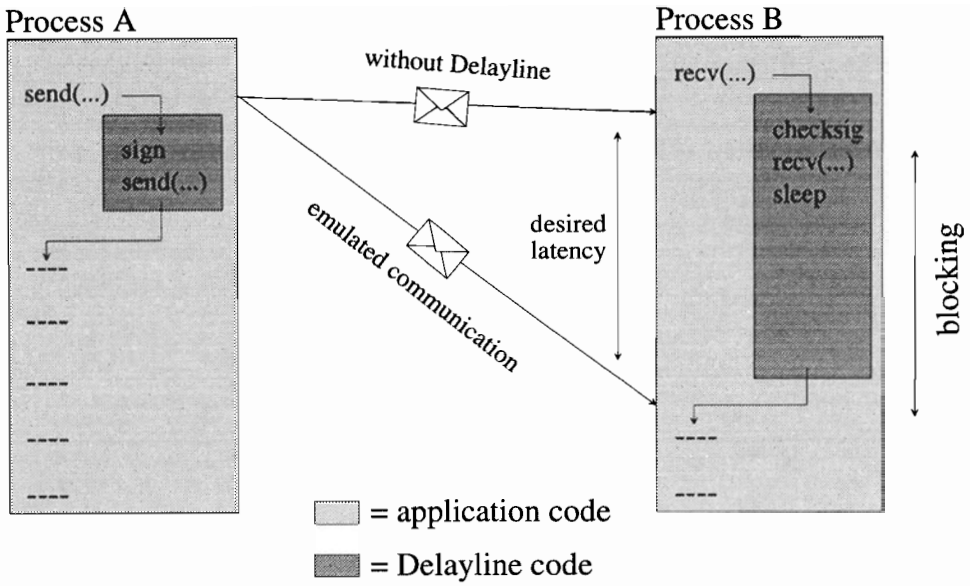

Figure 4. Emulating increased latency.

blocked until the data arrives on the socket (exceptions to these semantics are discussed in Section 5.2).

When data is received over the socket the first stage of processing is to check whether the data received originated from a process running the Delayline software. This check is a necessary inclusion so as not to exclude the possibility of communication with processes that are running Delayline and those that are not. With this assumption in mind, it was necessary to be able to check for the existence of a signature without affecting the data in the cases where a signature was not found. The signature checking was achieved by performing a generic receive from the socket, using an additional option flag, known as `MSG_PEEK`, which allows data to be read from the socket without removing the contents from the kernel buffer. If a signature is found in a received message, the delay time is extracted and the buffer is stripped of the signature.[1] Instead of returning this received data directly to the application, the specified delay is introduced at this stage before control is returned to the application. This technique provides a good emulation of a message arriving later than it actually did. This processing is illustrated in the diagram in Figure 4.

A simpler method of emulating increased latency would be to introduce the

1. The signature technique used by Delayline operates using the simple assumption that it is very unlikely for a non-Delayline process to generate a message containing the Delayline signature; it does not provide a 100% success guarantee.

necessary delay in the sending phase of the communication, that is, sleeping for the desired time before passing the message to the kernel to be transmitted. Indeed, from the point of view of the receiving process, this approach would have the same effect in terms of creating the illusion of increased transmission latency. It would have the advantage of not requiring interception of the application calls for receiving messages, and consequently the construction of the Delayline signatures would also not be required during the sending phase. However, this approach was not used as it breaks the semantics of the Berkeley socket interface. The problem is that by introducing the delay in the sending routines, control flow is held in the Delayline system for the duration of the sleep, rather than being returned to the application as soon as the data is copied into kernel buffer space.

Using the adopted approach, the unwanted latency in the sending process is reduced to the time taken for Delayline to decide the fate of the message and to add the required signature.

5. *Additional Complexities*

Although the technique described in the previous section was found to be a good mechanism for introducing the additional communication latency, the Berkeley socket interface provides additional functionality that causes complications for Delayline. The most significant of these is the `select` system call, which can be used to multiplex input/output requests among multiple sockets or files.

5.1. *Select*

Certain application programs, typically server programs, require communication over a number of sockets. Issuing a receive call on one of the sockets typically will cause the process to block even though there may be data waiting to be serviced on one of the other sockets. A number of techniques may be used to program this scenario, including nonblocking sockets and asynchronous input/output, both of which are discussed in subsequent sections. The most convenient approach for implementing such an application is to use the `select` system call, which allows the programmer to specify a number of descriptors (either sockets or files) to *watch* for activity. When data is detected on any of these descriptors, the call returns indicating the active descriptor. The prototype for the `select` call is as follows:

```
int select(int width, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout)
```

The `fd_set` arguments represent the sets of descriptors of interest. These are divided into three classes: `readfds` identifies the descriptors on which the caller wishes to be able to receive data; `writefds` identifies the descriptors to which data is to be written; and the third set is for descriptors for which exceptional conditions are pending. The call returns the number of active descriptors and modifies the data sets to indicate the descriptors that are ready for processing. The `timeout` parameter is used to specify the time after which the call should return if no activity is detected. When a `select` call returns and indicates that data is available to be read from a particular descriptor, then the application can invoke a receive call knowing that it will not be blocked.

In order for Delayline to maintain the correct semantics of the Berkeley socket interface when introducing latency into communications, it also intercepts application program invocations of the `select` call. The Delayline select routine (`dl_select`) invokes the standard `select` system call with the same parameters that it has received. If, upon return, the system call indicates that a socket has data ready, then a few bytes of data are read from the socket, without removing the data from the kernel's buffer. This data is examined to determine whether the message originated from a Delayline process and, if so, what latency is required to be injected. If the message does contain a Delayline signature, then the latency is introduced in `dl_select` rather than in the receive call. In order to record this fact, Delayline maintains a descriptor set, similar in operation to the sets previously described. When a message is manipulated in the `dl_select` routine, the corresponding socket is marked in the set. This can be examined by later invocations of either receive calls or further `dl_select` calls. After Delayline has made the message visible, the relevant socket is unmarked.

The action to be performed on messages intercepted during the `dl_select` routine depends on the relationship between three time values: (a) the delay to be introduced, T_{delay}, (b) the timeout value supplied to the `select` call, $T_{timeout}$, and (c) the elapsed time between the `select` call being invoked and the message arriving, $T_{elapsed}$. The three different timing scenarios to be considered are

- $T_{timeout}$ = indefinite: The delay can be introduced after the `select` system call returns and before control is passed back to the application. The socket is marked so that no further delay will be introduced during the receive call.
- $T_{delay} < (T_{timeout} - T_{elapsed})$: Processed in the same manner as for the infinite timeout case.
- $T_{delay} > (T_{timeout} - T_{elapsed})$: This scenario reflects the situation in which no messages have arrived within the specified timeout period. In order to emulate this, Delayline waits until the timeout expires and then returns to

the application, indicating that no activity has been detected, which involves specifying a return value of zero and massaging the descriptor sets. At the same time the emulated arrival time of the message is determined, which is known to be T_{delay} time units after the message was detected to have arrived. This is the time after which the message is to be made visible to the application. Delayline maintains a list of these visibility times for each of the active sockets in an application.

After an application has returned from `select` due to timeout expiry, it will typically invoke `select` again soon afterwards. If the Delayline `dl_select` routine intercepts a message over a socket that has already been marked as processed in the file descriptor set, then the visibility time table is referenced to determine whether this particular message should be now passed to the application.

When the Delayline receive routines intercept a message, the corresponding socket is referenced against the file descriptor set to determine whether the receive call was preceded by a `dl_select` call; if so, the visibility time table is referenced to determine whatever extra delay may be required before the message is passed to the application. At this point the socket is removed from the file descriptor set.

5.2. Nonblocking Sockets

It was previously stated that when a process invokes a receive system call over a socket the normal behavior is for the process to block until data arrives on that socket. However, UNIX provides a mechanism to override this behavior to allow programs to poll sockets rather than block awaiting data. This is achieved by using the `fcntl` or `ioctl` system calls to apply the `FNDELAY` (for `fcntl`) or `FIONBIO` (for `ioctl`) options to the required sockets. Once applied, an input/output request that cannot complete over such a nonblocking socket is not done. Instead, return is made to the caller immediately and the global `errno` is set to `EWOULDBLOCK`.

Naturally, allowing Delayline to introduce arbitrary delays into receive operations over nonblocking sockets would break the semantics of the Berkeley interface. To overcome this problem, a scheme is proposed which uses the visibility time table that was introduced during the discussion of the `select` call. When a message that is to be delayed arrives on a nonblocking socket, the visibility time is calculated, and the call returns, indicating that the operation would block. Further invocations reference the visibility time table to determine whether the message should be made available to the application.

5.3. Asynchronous Input/Output

All of the previous discussion about socket communication has assumed synchronous I/O. It is also possible to program interprocess communication applications using asynchronous I/O, which allows a process to instruct the kernel to notify it when a specified descriptor is ready for I/O. In order to achieve this, a process must perform a number of steps.

Using signal-driven I/O, an application is able to perform a receive operation on a socket knowing that data is ready to be read. Again, as in the `select` case, Delayline should not introduce delays into such receive calls because they would break the semantics of the asynchronous I/O.

Ideally, this problem could be solved by creating alternative signal handlers within the Delayline system to intercept the `SIGIO` signals. These handlers could mask the signal from the application for as long as required. Unfortunately, UNIX does not allow the nesting of signal handlers that would be required to implement this scheme, and at present Delayline does not provide a mechanism to suitably cater for this scenario.

5.4. Message Reordering

The current prototype implementation of Delayline has a limitation concerning the reordering of messages. Consider the following situation in which three hosts, A, B and C, connected by the same local-area network, are cooperating in an application. On host A, a typical server process, P_a, is listening on a well-known port for client requests. After receiving an initial message, the server creates another process to deal with the request and then continues listening on the well-known port. At each of the hosts B and C, processes P_b and P_c are required to utilize the service provided by A.

Delayline is being used with this application to emulate the scenario where host C is physically remote from the other two hosts. In this situation, it is expected that messages from P_c to P_a would suffer from a greater latency compared with those from P_b to P_a. For the sake of illustration say that Delayline subjects all messages from P_c to P_a to a fixed delay of 10 units, while messages from P_b to P_a are left untouched. At some point in the application's operation, P_c sends a message (M_c) to P_a, and shortly afterward P_b also sends a message (M_b) to P_a. When P_a receives M_c, Delayline introduces the required delay of 10 units before passing the message to the application. During this delay period message M_b arrives on the same well-known port at which P_a has just received M_c. This message should be passed directly to the application. However, since Delayline is currently processing M_c, then M_b is not read from the socket until M_c has been passed to

the application. It is apparent that a problem arises in this situation. Although M_c arrived at P_a before M_b, the messages should be passed to the application in the reverse order. This problem is also visible in certain situations when using the Delayline `dl_select` routine. A general solution to this message re-ordering problem is currently being prepared for a future version of the Delayline system.

6. Network Partitioning

Providing an efficient and accurate emulation of network partitioning is more difficult to achieve than the other wide-area network characteristics previously described. The basic illusion that one host is partitioned from another host is logically straightforward to create and could be implemented by losing all messages sent between hosts on opposite sides of the emulated partition.

The major difficulty is due to the necessity that all of the processes in the distributed application must have a consistent view of the emulated network. If a partition occurs in a real network, separating one group of hosts from another, then typically all of the processes will be affected by this partition when inter-group communication is attempted. The emulation of increased latency and message loss only requires that all of the processes have the same set of configuration parameters, whereas emulating a network partition requires a technique by which a change of network state can be propagated to all of the application processes at a specified time.

Effective partition emulation means that Delayline must be aware of the connectivity of the emulated network, that is, the Delayline groups that would be affected if the network partitioned at a particular point. Describing this connectivity requires that the current configuration information be augmented by parameters indicating the effect of the various partition possibilities. Additionally, the configuration system must allow the various partition options to be described in terms of the probabilities of their occurrence and the distributions of their recovery times.

The current version of Delayline does not support network partition emulation. A more detailed study of the problem is currently being carried out, with a view to incorporating this additional functionality into future releases.

7. Evaluation

Delayline performance can be evaluated in two ways. Of primary importance is the accuracy of the emulation. However, even a perfect emulation would not be

acceptable if Delayline imposed too large a processing overhead on the application. These two factors are discussed in the remainder of this section.

7.1. Emulation Accuracy

In order to test the operation of Delayline, a small utility program (`timer`) was developed to produce round-trip time statistics between two hosts. This program operates in two modes, server or client mode. In server mode, the program listens for messages on a specified port and immediately echoes them back to the originator. In client mode, the program measures the time taken between sending a message to a specified server and receiving the reply. The program can send the messages using either UDP or TCP protocols.

In order to evaluate the emulation abilities of Delayline, an experiment was carried out to attempt to make two locally connected hosts appear as if they were physically located on opposite sides of the world—one in Newcastle, England, and one in Adelaide, Australia.

When Delayline is used to emulate the communication characteristics of a particular wide-area network connection, some information about the performance of the connection to be emulated is needed. In some circumstances it may be possible, by using a program such as the `timer` utility, to measure the actual performance. However, there are other situations where it is impractical or even impossible to obtain real, measured performance figures. In these cases, an approximation is required. For this experiment the UNIX `ping` program (Packet InterNet Groper) was used to determine an approximation of the communication characteristics of the connection between Newcastle and Adelaide. The `ping` program uses the Internet Control Message Protocol (ICMP) [Postal 1981] to send echo request messages to a specified host and waits for a reply. ICMP messages are encapsulated in IP datagrams and therefore do not rely on upper-layer protocols. However, `ping` statistics give a reasonable indication of the performance achievable using UDP or TCP.

The `ping` program was used to send 5,000 test messages from Newcastle to Adelaide, the resulting statistics produced are as follows:

```
----cruskit.aarnet.edu.au PING Statistics----
5000 packets transmitted, 4050 packets received, 19% packet loss
round-trip (ms)  min/avg/max = 1143/1524/6547
```

The corresponding statistics between the two locally connected hosts are

```
----glororan.ncl.ac.uk PING Statistics----
2000 packets transmitted, 2000 packets received, 0% packet loss
round-trip (ms)  min/avg/max = 2/3/20
```

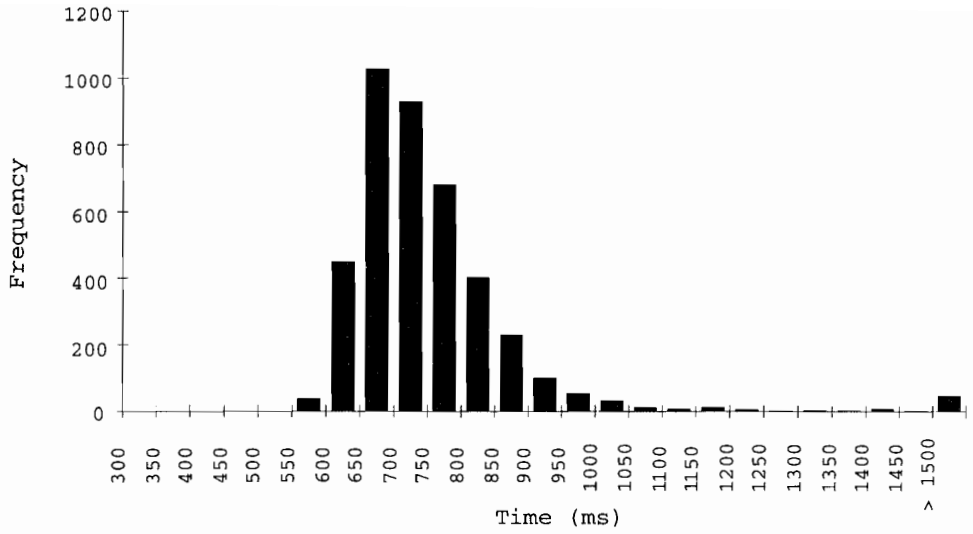

Figure 5. Observed Newcastle to Adelaide timings.

Comparing the two sets of results shows that communications between the two local machines is, as expected, fast (average round-trip time of 3 ms) and very reliable (0 percent packet loss). Communication between Newcastle and Adelaide is considerably slower (average round-trip time of 1,524 ms) and less reliable (19 percent packet loss). Using the `traceroute` program to examine the route taken shows that the packets pass through a total of 13 hops en route from source to destination, traveling via America to reach their destination in Adelaide.

The individual round-trip times, produced by `ping`, between Newcastle and Adelaide were halved to provide an approximation to the one-way times. These results were analysed and a distribution of the results is shown in Figure 5. Although the return routes taken by the messages may not be the same as the outgoing routes, it is thought reasonable to assume approximately equal performance.

The parameters for the distribution are given in Figure 6. A number of distribution functions were used with these parameters to find the optimum emulation of the observed distribution. The best emulation results were produced using the two-sided gamma distribution provided by Delayline. Standard statistical techniques were used to calculate the required parameters for the distribution.

Delayline was configured so that communication between the two local hosts was subjected to 9.5 percent message loss and message delays distributed with the aforementioned gamma distribution.

Two versions of the `timer` program were developed, one compiled without Delayline and one with. These two versions were used to time 2,000 round trips

Mean	762.34
Standard Deviation	203.09
Variance	41245.9
Minimum value	571.5
Maximum value	3273.5

Figure 6. Newcastle to Adelaide distribution parameters.

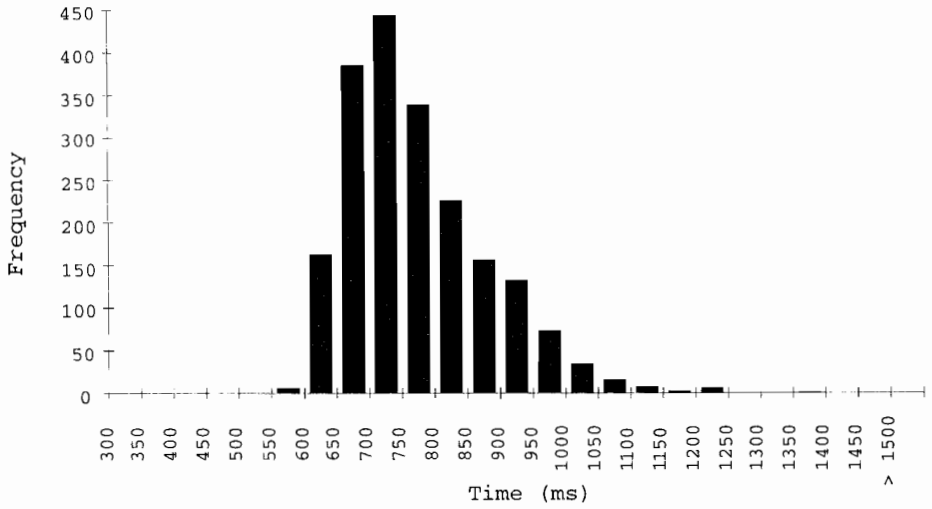

Figure 7. Emulated Newcastle to Adelaide timings.

between the two locally connected hosts using UDP. The summary results produced by the version of the `timer` compiled without Delayline are

```
2000 roundtrips from glororan.ncl.ac.uk have been analysed.
Min = 1 ms, avg = 1 ms, max = 40 ms.
```

The corresponding results from the timer compiled with Delayline are

```
2000 roundtrips from glororan.ncl.ac.uk have been analysed.
Min = 1160 ms, avg = 1541 ms, max = 2720 ms.
```

As with the `ping` statistics between Newcastle and Adelaide, the individual round-trip times were halved to get an approximation of one-way times. These results were analysed and a distribution of the results is shown in Figure 7. It can be seen that the emulated distribution is a close approximation to the actual distribution shown in Figure 5, thus validating the hypothesis that Delayline is able to effectively emulate wide-area networking performance using a local-area network.

7.2. Delayline Overhead

Each time a message is intercepted, processing is carried out to determine what action, if any, should be performed. This processing naturally introduces an overhead, even when no message manipulation is required. Delayline uses a number of techniques, including caching, to minimize this overhead.

In order to quantify the unwanted latency introduced by the system, the version of the `timer` program compiled with Delayline, described earlier, was used with configuration settings indicating no message loss and no additional delays. The program was again run between the two local workstations, and the results are

```
2000 roundtrips from glororan.ncl.ac.uk have been analysed.
Min = 2 ms, avg = 2 ms, max = 123 ms.
```

The maximum value of 123 ms is due to the processing required to contact the configuration server the first time the process executes a Delayline routine. The remaining values are all located close to the mean of 2 ms. Comparing these results to those produced by the version of the timer compiled without Delayline indicates an average round-trip overhead of 1ms. This equates to an overhead of 500 μs per message interception. This additional overhead incurred is thought to be small enough not to adversely effect the performance of the application being tested.

8. Conclusions

This paper has shown that Delayline has the ability to change the characteristics of the underlying network supporting a distributed application. Experience in using Delayline with large distributed systems has shown the usefulness of such a technique for tuning applications to run in a wide-area network environment.

There are a number of other systems available that are loosely described as network emulation tools. Nest [Bacon et al. 1988] is a tool for prototyping distributed algorithms and systems. Nest provides its own communications interface to construct prototype distributed applications. The system provides a simulation of the entire distributed application within a single process. NET [Baclawski 1987] provides a software simulation of a computer network that allows distributed applications to be modeled as a number of processes that execute on a single host. Systems like Network II.5 [CACI 1993] and Starlite [Cook 1987] allow simulations of distributed computer systems to be carried out by providing modules for representing devices such as clocks, disks, and networks. Delayline differs from

all of these simulation tools in that it operates with real distributed applications rather than providing a simulation of a distributed system.

The most challenging aspect of the Delayline implementation was providing the message manipulation techniques needed to obtain a good emulation while maintaining, as closely as possible, the correct semantics of the socket programming interface. The current version of Delayline incorporates functionality that achieves this successfully in most cases. However, further work is necessary to effectively handle asynchronous I/O using signals.

Another area for continued study includes experimentation with the other possible methods of message interception outlined in Section 2.2. Using alternative shared libraries to provide the entry points into the Delayline software would allow the system to be used with applications for which the source code is not readily available.

The network partitioning emulation, introduced in Section 6, requires more research in two areas: first, an investigation of the types of partitioning problems that are visible in real wide-area networks so as to provide the basis for accurate emulation and second, an investigation of how to effectively integrate partition emulation into the Delayline software.

Another quite different development would be to use the Delayline infrastructure to provide a fault injection facility for distributed applications. Currently, Delayline pays no regard to the content of the messages that are intercepted. A scheme has been conceived that could use the existing techniques provided by Delayline and extend them to perform user-defined actions based on the content of the intercepted messages. This approach would make it possible to transparently inject faults into a distributed system at a certain point in the communications. For example, a host could be made to appear as if it had crashed, from the point of view of the other nodes in the system, by masking all further communications at a certain user-defined stage in the processing.

Acknowledgments

This work has been supported in part by grants from the UK MOD, the Science and Engineering Research Council (Grant number GR/H81078), and ESPRIT basic research project 6360 (BROADCAST).

References

1. K. Baclawski. A Network Emulation Tool. In *Proceedings of the Symposium on the Simulation of Computer Networks*, pp.198–206, August 1987.
2. D. F. Bacon, A. Dupuy, J. Schwartz, and Y. Yemini. Nest: A Network Simulation and Prototyping Tool. IBM T. J. Watson Research Center, 1988.
3. J. P. Black, L. F. Marshall, and B. Randell. The Architecture of UNIX United. In *IEEE Special Issue on Distributed Database Systems*, pp. 709–718, 1987.
4. D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle Connection or UNIXes of the World Unite! *Software Practice and Experience*, 12(12):1147–1162, December 1982.
5. D. E. Comer. *Internetworking with TCP/IP: Principles, Protocols, and Architecture*. New York: Prentice-Hall, 1988.
6. R. P. Cook. StarLite: A Network-Software, Prototyping Environment. In *Proceedings of the Symposium on the Simulation of Computer Networks*, pp. 178-184, August 1987.
7. J. B. Postel. Internet Control Message Protocol. Technical Report RFC762, Network Information Center, SRI International, September 1981.
8. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of the Arjuna Distributed Programming System, *IEEE Software*, pp. 66–73, January 1991.
9. CACI Products Company. *A Quick Look at Network II.5*, 1993.