Probing and Fault Injection of Protocol Implementations

Scott Dawson and Farnam Jahanian

Real-Time Computing Laboratory
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
{sdawson,farnam}@eecs.umich.edu

ABSTRACT

This paper presents a technique for probing and fault injection of distributed protocols. The proposed technique, called script-driven probing and fault injection*, can be used for studying the behavior of distributed systems and for detecting design and implementation errors of fault-tolerant protocols. The focus of this work is on fault injection techniques that can be used to demonstrate three aspects of a target protocol:* i) detection of design or implementation errors, ii) identification of violations of protocol specifications, and iii) insight into design decisions made by the implementors. *The emphasis of our approach is on experimental techniques intended to identify specific "problems" in a protocol or its implementation rather than the evaluation of system dependability through statistical metrics such as fault coverage.*

To demonstrate the capabilities of this technique, the paper describes a probing and fault injection tool, called the PFI *tool (Probe/Fault Injection Tool), and several experiments that studied the behavior of two protocols: the Transmission Control Protocol (TCP) [4, 25] and the Group Membership Protocol (GMP) [18]. The tool can be used to delay, drop, reorder, duplicate, and modify messages. It can also introduce new messages into the system to probe participants. In the case of TCP, we used the* PFI *tool to duplicate the experiments reported in [7] on several TCP implementations without access to the vendors' TCP source code in a very short time. We also ran several new experiments that are difficult to perform using past approaches based on packet monitoring and filtering. In the case of GMP, we used the tool to test the fault-tolerance capabilities of an implementation under various failure models including daemon/link crash, send/receive omissions, and timing failures. Furthermore, by selective reordering of messages and spontaneous transmission of new messages, we were able to guide a distributed computation into hard to reach global states without instrumenting the protocol implementation.*

Keywords: distributed systems, communication protocol, fault injection, protocol testing, executable specifications

1 Introduction

As software for distributed systems becomes more complex, ensuring that a system meets its prescribed specification is a growing challenge that confronts software developers and system engineers. Meeting this challenge is particularly important for distributed applications with strict dependability constraints since they must provide the required services under various failure scenarios. As we are witnessing a convergence of key technologies, emerging new applications, and market needs in this decade, we can expect that distributed systems will become more complex and that an increasing number of them will have to operate under strict availability and reliability requirements.

In this paper, we present a technique, called *script-driven probing and fault injection*, for studying the behavior of distributed systems and for testing the fault tolerance capabilities of distributed applications and communication protocols. The proposed technique is motivated by several observations:

- In testing a distributed system, one may wish to coerce the system into certain states to ensure that specific execution paths are taken. This requires the ability to orchestrate a distributed computation into "hard-to-reach" states.
- Asynchronous communication and inherent non-determinism of distributed systems introduces additional complexity. One must be able to order certain concurrent events to ensure that certain global states can be reached.
- In testing the fault-tolerance capabilities of a distributed system, one often requires certain behavior from a protocol participant that may be impossible to achieve under normal conditions. This may require the emulation of "misbehaving" participants by injecting faults into the system.
- Testing organizations often require a methodology that does not instrument the code being tested. This is particularly important for testing existing systems or when the source code is unavailable.
- Most existing testing and fault injection approaches depend heavily on probabilistic (or random) test generation. Orchestrating a distributed computation into a particular execution path requires deterministic approaches.

The remainder of this paper is organized as follows: Section 2 presents the approach for probing and fault injection of distributed systems. Section 3 described the implementation of a tool based on the proposed approach. Section 4 discusses in detail the experimental results from studying several implementations of TCP and GMP protocols. Section 5 describes related work. Section 6 presents concluding remarks and describes future directions of this work.

2 Approach

2.1 Script-Driven Probing and Fault Injection

The proposed approach views a distributed protocol as an abstraction through which a collection of participants communicate by exchanging a set of messages, in the same spirit as the *x*-Kernel [16]. In this model, we make no distinction between application-level protocols, interprocess communication protocols, network protocols, or device layer protocols. As shown in Figure 1(a), each protocol is specified as a layer in the protocol stack such that each layer, from the device-level to the application-level protocol, provides an abstract communication service to higher layers.

As mentioned earlier, determining whether or not a protocol implementation meets its prescribed specification requires orchestrating the system execution in a deterministic way. The proposed approach relies on intercepting and filtering messages between protocol participants. In particular, a probe/fault injection (*PFI*) layer is inserted between any two consecutive layers in a protocol stack. The *PFI* layer can execute deterministic or randomly-generated test scripts to probe the participants and inject various faults into the

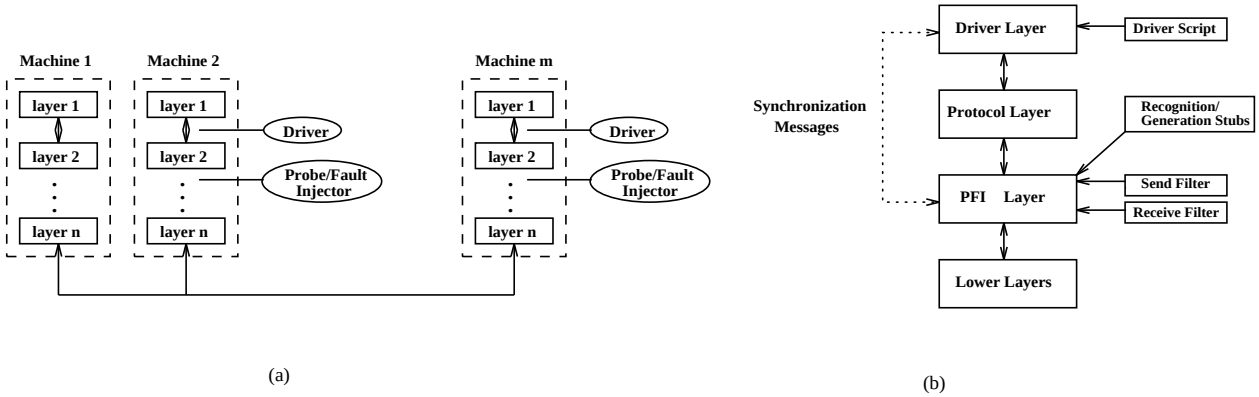


Figure 1: (a) Protocol Stacks. (b) Script Interaction.

system. By intercepting and filtering messages between two layers in a protocol stack, the *PFI* layer can delay, drop, reorder, duplicate, and modify messages. Furthermore, the *PFI* layer can introduce spontaneous messages into the system to probe the participants and to orchestrate the system execution into a particular path. Because the *PFI* layer scripts can recognize different packet types, they can perform filtering and fault injection based on message type.

Figure 1(a) illustrates how the nodes in a distributed system run a modified protocol stack to test Layer 2, which we will call the *target protocol*. The idea is that the target protocol layer is encapsulated between the driver and *PFI* layers on the protocol stack. The driver and *PFI* layers are used to examine and to manipulate the messages exchanged between the participants in the target protocol. The driver layer is responsible for generating messages and running the test. The *PFI* layer intercepts all messages coming into and leaving the target layer. The *PFI* layer can manipulate messages to/from the target layer as they pass through the protocol stack, and it can introduce spontaneous messages into the system to observe the behavior of target protocol participants on other nodes. The driver and *PFI* layers communicate with each other during the test and can coerce the system into certain states.

The reason for having a layer both above and below the target layer is to allow creation of new messages and manipulation of messages generated by other participants in a protocol. The *PFI* layer which only sits below the target layer can drop, delay, and corrupt messages, but it may not be able to generate messages directly because it cannot manipulate data structures in the target layer. The driver layer, however, can be used for doing most message generation so that data structures in the target protocol will be updated correctly.

The driver and *PFI* layers run scripts which control their actions as messages are exchanged between protocol participants. As shown in Figure 1(b), the *PFI* layer runs a script, called the *send filter*, each time a message is pushed (or sent down) the protocol stack. It runs another script, called the *receive filter*, each time a message is popped (or sent up) the protocol stack. These scripts perform three types of operations on messages:

1. **Message filtering:** for intercepting and examining a message.
2. **Message manipulation:** for dropping, delaying, reordering, duplicating, or modifying a message.
3. **Message injection:** for probing a participant by introducing a new message into the system.

The packet *recognition/generation stubs* shown in Figure 1(b) are invoked to determine the message type whenever a message in intercepted by the *PFI* layer. An interface from the send/receive scripts to the stubs exists so that the scripts can determine what types of packets they are operating on. The send/receive scripts can also use the packet generation stub to generate messages of certain types at the *PFI* layer. The packet stubs are written by people who know the packet formats of the target protocol. A packet stub may

be written by the protocol developer for an application-level protocol, or it may be supplied by the system for a popular protocol such as TCP whose packet formats are known.

Message generation can only be done if state of the target protocol doesn't have to be updated for the generated message. For example, when generating a spurious `ACK` message in TCP, no data structures need to be updated. The message can simply be generated and sent. However, when generating a data message in TCP, the sequence number would need to be used and updated, and the message would have to be kept track of in case retransmissions were needed. This type of message generation cannot be done by the *PFI* layer because it does not have access to the data structures of the target protocol (in this case, TCP). In such cases, the message can be generated by the driver layer.

2.2 Failure Models

Testing the fault-tolerance capabilities of a protocol implementation requires the emulation of misbehaving participants by injecting various types of faults into the system. Hence, techniques that exercise the fault-tolerance capabilities of a distributed system must take into consideration the various ways in which a protocol implementation may fail.

A protocol participant is faulty if it deviates from its prescribed specification. A model of failures specifies in what way a protocol participant can deviate from its correct specification. The fault injection approach introduced earlier can test the fault-tolerance capabilities of protocol implementations under various failure models commonly found in the distributed systems literature including: *process crash failures, link crash failures, send omission failures, receive omission failures, timing/performance failures,* and *arbitrary/byzantine failures.* Although a formal treatment of different failure models is beyond the scope of this presentation [14], a brief outline of various failure assumptions that can be tested by our technique is described below.

Process crash failures: A process/processor fails by halting prematurely and doing nothing from that point on. Before stopping, however, it behaves correctly.

Link crash failures: A link fails by losing messages, but it does not delay, duplicate, or corrupt messages. Before ceasing to transport messages, however, it behaves correctly.

Send omission failures: A process fails by intermittently omitting to send messages it was supposed to send.

Receive omission failures: A process fails by intermittently omitting to receive the messages that were sent to it.

General omission failures: A process fails by suffering a send or receive omission failure, or both. Similarly, a link fails by intermittently omitting to transport messages sent through it.

Timing/performance failures: A process fails by violating the bound on the time required to execute a step. A link fails by transporting messages faster or slower than its specification.

Arbitrary/byzantine failures: A faulty link or faulty process can exhibit arbitrary behavior including:

- generate spurious messages,
- claim to have received a message,
- modify/corrupt message content, and
- reorder messages.

The models presented above can be classified in terms of severity. Model B is more severe than model A if the set of faulty behavior allowed by A is a proper subset allowed by B. Thus, a protocol implementation that tolerates failures of type B also tolerates those of type A. The failure models above were presented in the order of severity.

2.3 Script Specification

Scripts are at the heart of this approach. Scripts are instructions that are executed by the driver and the probe/fault injection (*PFI*) layer to orchestrate the system computation into a particular state and to inject various kinds of faults into a system. A system designer must be able to specify sufficiently powerful scripts for manipulating the messages exchanged in a distributed computation. We must emphasize that the scripts serve a dual purpose. They are used for:

- specification of the instructions to orchestrate a distributed computation into a desired state, and
- specification of the fault(s) to be injected into the system once a certain state is reached.

We believe that inventing a new scripting language is not the solution. Instead, modifying and supporting a popular interpreted language with a collection of predefined libraries gives the user a very effective tool which allows him/her to write most scripts. It also eases the burden of learning a new language for users already familiar with whatever interpreted language is chosen. Furthermore, if a script written in this interpreted language can invoke user-defined procedures which can modify the internal state of the protocol, then the system designer has the ability to write scripts which can perform complex actions. This is a powerful tool because changing the scripts to perform new or different tests does not require re-compiling the *PFI* layer. The only time a re-compilation is required is when the library routines are changed. (As mentioned in Section ??, we plan to use *Tcl* as the scripting language in the implementation of our tool. Tcl allows users to define their own extensions, usually written in C, to the scripting language.)

Our experience during the last few months supports the view that a rich set of predefined library routines can help the system designer to develop powerful scripts in a very short time. In particular, predefined procedures can be used for:

- filtering messages based on the header or content,
- dropping, delaying, modifying, and duplicating messages,
- introducing spontaneous or probe messages to observe the response from another participants,
- reordering events in a run to ensure that certain global states are reachable.
- injecting various fault types into the system as described before,
- synchronizing scripts executed by *PFI* layers running on different nodes, and
- setting and manipulating timers and clock variables.

3 Probe/Fault Injection (*PFI*) Tool

In order to demonstrate the effectiveness of the approach presented in Section 2, we developed a tool based on the concept of script-driven probing and fault injection. We also performed extensive experimental studies of several commercial and prototype distributed protocols. This section introduces a brief overview of the tool; the next section presents a detailed discussion of our experiments.

The *Probing and Fault Injection* (*PFI*) tool was initially developed on the *x*-Kernel running on Mach 3.0 and later ported to SunOS. The tool is meant to be inserted into a protocol stack as a separate layer of the stack below a target protocol. Figure 2 illustrates the various components of the *PFI* tool: *send/receive scripts, filters (or interpreters), recognition/generation stubs, user-defined procedures,* and *system utilities.*

Send/receive scripts are the instructions for orchestrating a computation into a particular path and for injecting faults into a system. We chose Tcl [24] as the language for writing the scripts. *Filters* are the interpreters that execute the scripts as messages pass through the *PFI* layer. In Tcl, an interpreter is simply an object which contains some state about variables and procedures which have been defined. In order to run a script, the user *evaluates* the script in the context of the interpreter. The call might look something like `Tcl_Eval(interp, script)`, where `interp` is a handle on the interpreter object in which the script is evaluated, and `script` is a character string. Each time a message passes into the *PFI* layer, the appropriate

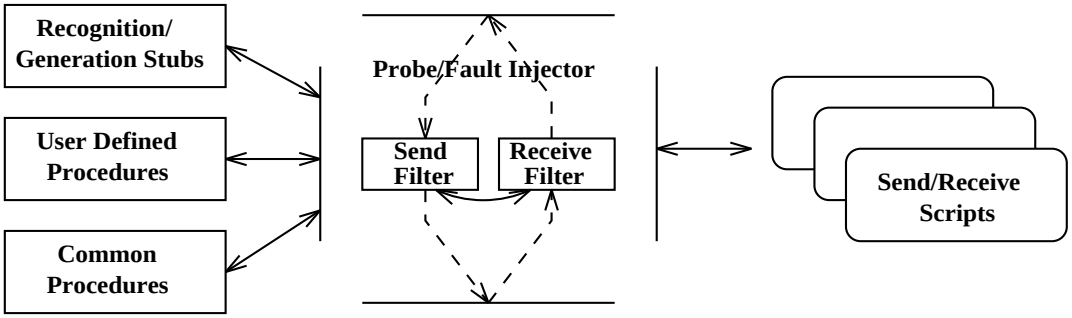


Figure 2: Probe/Fault Injection Tool

(send or receive) script is interpreted in the appropriate interpreter. A handle on the current message (called *cur_msg*) is also created so that the script can perform operations on this message.

When a script is interpreted, it may do several things. The most common action is to invoke the operation `msg_type x`, which returns the type of message `x`. Usually this operation is performed on the current message in order to determine whether to perform some action on the message or not. A simple example of a script written in Tcl follows. This script simply drops all acknowledgement (`ACK`) messages.

```
# Message types are ACK, NACK, and GACK.
# This script drops all ACK messages.
set ACK  0x1
set NACK 0x2
set GACK 0x4

# Print out a banner and then the contents of the current message.
puts -nonewline "receive filter: "
msg_log cur_msg

# Get the type of the message and drop it if it's an ack.
set type [msg_type cur_msg]
if {$type == $ACK} {
    xDrop cur_msg
}
```

Scripts can also set up state which can be used later in the test. For example, a script may keep a running count of the number of messages which have passed through the *PFI* layer. Since state of variables is stored in the interpreter object, the value of this count is persistent across messages.

The other components of the tool is a set of *utility procedures* which the script can call. Several of these were used in the above script, such as `msg_log`, `msg_type`, and `xDrop`. There are several categories of utilities. One is the *recognition/generation stubs*. These allow the script writer to determine the type of a message (recognition), and also to create messages of certain types (generation). The stubs are written by anyone who understands the headers or packet format of the target protocol. They could be written by the protocol developers or the testing organization, or even be provided with the system in the case of popular and well known protocols such as TCP.

Another category of utilities is a set of *common procedures* which might be frequently invoked by scripts writers in testing different protocols. These procedures might perform actions such as logging or dropping

messages, as in `msg_log` and `xDrop`. Of course, `msg_log` depends on the contents of the message, so it would have to be tailored by someone who knows the header format of the target protocol. Also included in this category of utilities is a set of procedures which allow the user to generate probability distributions. For example, a call such as `dst_normal mean var` will produce numbers with a normal distribution around *mean* with variance *var*. In this way, it is possible for the script writer to perform actions on messages in a probabilistic manner if desired. Other things that fall into this category are procedures which give the script access to system time and procedures which allow the send filter script to change state in the receive filter and vice versa. Changing the state of the other interpreter is used for cross-interpreter communication. For example, the send filter might set a variable in the receive interpreter which tells the receive filter to start dropping messages. Lastly, there is the category of user defined procedures. These are procedures which may be needed by whomever is performing the test. These procedures can be written in C and linked into the tool. The scripts then have the ability to call the new procedures, allowing the tester to perform more powerful tests.

The basic idea behind *script driven probing and fault injection* is that testing different failure scenarios and creating different tests is accomplished simply by invoking different scripts. Since these scripts are inputs into the *PFI* tool, changing the scripts does not require recompilation of the tool. Once all of the utility procedures are in place, the tool is compiled. After that, barring changes to the utility procedures, the tool remains static. This reduces the time required to run different kinds of tests compared to a system which might require some type of re-compilation in order to run different tests.

4 Experimental Results

This section describes the results of extensive experiments on several commercial implementations of TCP and a prototype implementation of a Group Membership Protocol (GMP). These experiments were conducted to demonstrate the capabilities of the *PFI* tool described in Section 3. Three aspects of the target protocols were demonstrated: *i) detection of design or implementation errors, ii) identification of violations of protocol specifications, and iii) insight into design decisions made by the implementors.* The fault injection experiments and their results are summarized in the following two subsections.

4.1 Testing of TCP

The Transmission Control Protocol (TCP) is an end-to-end transport protocol that provides reliable transfer and ordered delivery of data. TCP is connection-oriented protocol and it uses flow-control between protocol participants to operate over network connections that are inherently unreliable. Because TCP is designed to operate over links of different speeds and reliability, it is widely used on the Internet. TCP was originally defined in RFC-793 [25] and was updated in RFC-1122 [4]. In order to meet the TCP standard, an implementation must follow both RFCs.

For testing TCP, we modified an *x*-Kernel protocol stack to include a layer which incorporates the *PFI* tool described in Section 3. We call this layer the *PFI* layer. The *PFI* layer sits directly between the TCP layer and the IP layer. The resulting *x*-Kernel protocol stack is shown in Figure 3. In the figure, there is one machine running Mach and a modified *x*-Kernel protocol stack. It is connected to the network which also has machines running vendor TCP implementations. In the tests, connections are opened between the vendor TCP implementations and the *x*-Kernel TCP.

Four vendor implementations of TCP were tested using the *PFI* tool. They were the native TCP implementations of SunOS 4.1.3, Solaris 2.3, AIX 3.2.3, and NeXT Mach, which is based on Mach 2.5. The SunOS, AIX, and NeXT Mach implementations were all very similar, and seemed to have been based on the same release of BSD unix. Solaris, which is based on an implementation of System V, behaved differently than the others in most experiments. Five experiments were performed on each of the TCP implementations, as described below. Results of these experiments are summarized in a tabular form for ease of reference.

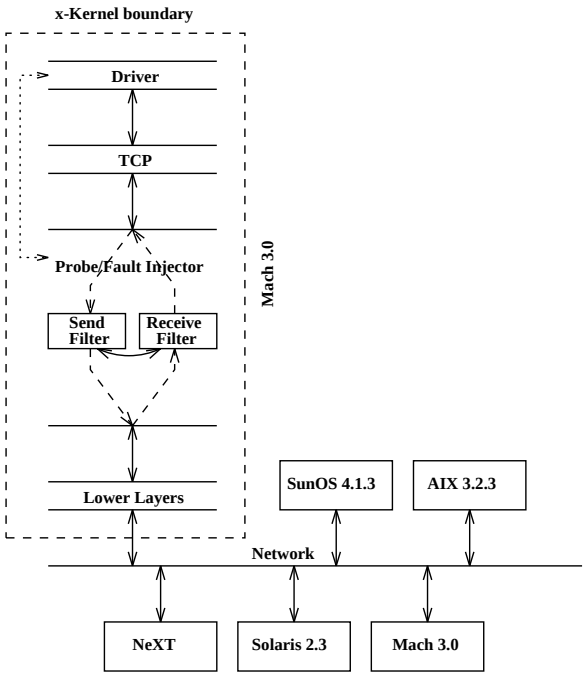


Figure 3: TCP Test Platform.

Experiment 1: TCP retransmission intervals

This experiment examines how different implementations of TCP retransmit dropped data segments. TCP uses timeouts and retransmission of segments to assure reliable delivery of data segments. Each time the sender sends a data segment, a timeout for the segment is set. If an acknowledgement is not received before the expiration of the timeout, the data is assumed lost and is retransmitted. When the data is retransmitted, another timeout is set to keep track of when the acknowledgement is expected. The TCP specification states that for successive retransmissions of the same segment, the retransmission timeout should increase exponentially. It also states that an upper bound on retransmissions may be imposed.

This experiment tested how vendor TCP implementations retransmitted segments. In the experiment, a connection was opened to *x*-Kernel machine from the each vendor machine. The receive filter script of the *PFI* layer was configured such that after allowing thirty packets through without dropping or delaying their `ACK`s, all incoming packets were dropped. In order to monitor the retransmission behavior of the SunOS 4.1.3, Solaris 2.3, AIX 3.2.3, and NeXT Mach implementations, each packet was logged with a timestamp by the receive filter script before it was dropped. When the *PFI* layer started dropping messages, no further data was received by the TCP layer of the *x*-Kernel machine and so no `ACK`s were sent. The results of the experiment are summarized in Table 1.

The SunOS 4.1.3 machine retransmitted the packet twelve times before sending a TCP reset and closing the connection. The timeout on the retransmissions increased exponentially until it reached 64 seconds, at which point it leveled off. This was the upper bound on the retransmission timeout. In most cases, the Sun sent the next segment in the sequence space soon after the initial transmission of the dropped segment. This is so that if the original segment was only delayed in the network, the `ACK` was delayed or dropped, or the receiving TCP was using delayed `ACK`s, data is not retransmitted because it need not be. By transmitting the next segment in the sequence space, the sending TCP was simply eliciting an `ACK` for both segments at the same time. After not receiving the `ACK` for either segment, the original segment was retransmitted until the connection was timed out and dropped.

[1] One possible reason for this is to increase performance on high speed local area networks. The RFC states that a lower bound of 1 second is probably inadequate for high speed LANs.

	Results	Comments
SunOS 4.1.3	Retransmitted segment 12 times before sending TCP reset and closing connection. Retransmissions increased exponentially. Used an upper bound on retransmissions of 64 seconds.	
AIX 3.2.3	Same as SunOS	
NeXT Mach	Same as SunOS	
Solaris 2.3	Retransmitted segment 9 times before closing connection abruptly. No reset segment was sent. Retransmissions increased exponentially as in other implementations.	No upper bound was established because the connection closed before stabilizing at one. Also, there was a very short lower bound on retransmissions (330 milliseconds as opposed to 1 second used in other implementations[1].

Table 1: TCP Retransmission Timeout Results

Behavior on the RS/6000 running AIX 3.2.3 and the NeXT machine running Mach was essentially the same as that of the SunOS 4.1.3 machine. The segment was retransmitted twelve times before a reset was sent and the connection was dropped. The timeout on the retransmissions increased exponentially until it reached an upper bound of 64 seconds. In all cases, both machines transmitted the next segment soon after the segment which was dropped, but when no `ACK` was received, they started transmitting the original segment until the connection was timed out and dropped.

The Solaris 2.3 implementation behaved differently than the others. The connection was dropped abruptly without stabilizing at an upper bound as in other implementations. This occurred after nine retransmissions of the packet. The reason that no upper bound was reached was that there was a very short lower bound on retransmissions in Solaris (an average of 330 milliseconds over 30 runs). Therefore, exponential backoff of the RTO started from around 330 milliseconds, and the ninth retransmission occurred an average of only 48 seconds after the eighth. When the connection was dropped, no reset segment was sent, presumably because no one would be waiting to receive it. Table 1 summarizes the results of the above experiment.

Experiment 2: RTO with three and eight second `ACK` delays

This experiment examines how different implementations of TCP adjust the retransmission timeout value in the presence of network delays. The retransmission timeout value (RTO) for a TCP connection is calculated based on measured round trip time (RTT) from the time each packet is sent until the `ACK` for the packet is received. RFC-1122 specifies that a TCP must use Jacobson's algorithm [17] for computing the retransmission timeout coupled with Karn's algorithm [20] for selecting the RTT measurements. Karn's algorithm ensures that ambiguous round-trip times will not corrupt the calculation of the smoothed round-trip time.

We ran two variations on the same experiment. The experiment was to delay acknowledgements of incoming segments in order to check the response of the sending TCP to this apparent network delay. One variation used an `ACK` delay of three seconds, the other used a delay of eight. The send script of the fault injection layer was set up to delay each outgoing `ACK` for 30 `ACK`s in a row. After doing this, the receive filter started dropping all incoming packets. Each incoming packet was logged. Approaches which depend on monitoring and filtering packets [7, 21] cannot perform tests like this one because they do not have the ability to manipulate messages. For example, they do not have the ability to direct the system to perform a task such as "delay all `ACK` packets."

We expected that the RTO value of the sender would be adjusted to account for apparent network delays, and that the first retransmission of a segment would occur more than three (or eight) seconds after the initial transmission of the segment. We also expected that subsequent retransmissions of the same segment would occur with timeouts increasing exponentially from the value of the initial RTO to 64 seconds at which point the RTO would level off. We had not yet established an upper bound for Solaris 2.3 TCP, and hoped to do

so in the course of this experiment.

In the SunOS 4.1.3 experiment, when the x-Kernel machine started dropping packets, the first retransmission occurred about 6.5 seconds after the initial transmission of the segment. Additional retransmissions increased exponentially from 6.5 seconds until they leveled off at 64 seconds. In AIX 3.2.3 the initial retransmission was at eight seconds and retransmissions backed of exponentially as well. The NeXT started at five seconds and increased exponentially. In all experiments, the next segment was transmitted soon after the first segment as in the previous no delay experiment.

The behavior of the Solaris implementation was different than the others. The first retransmission of the dropped segment occurred at an average of 2.4 seconds. The second retransmission was seen an average of 1.2 seconds later, and exponential backoff started from there. It seems that Solaris 2.3 TCP either did not use Jacobson's algorithm, or did not select RTT measurements in the same way as other implementations. It was not nearly as adaptable to a sudden slow network as the other implementations. Because of the low RTO value, all connections timed out before stabilizing at an upper bound. In most of the runs (out of 30 runs), only seven retransmissions occurred. Only one run had nine retransmissions.

In an effort to find out why connections were being dropped before nine retransmissions (which was how many occurred in the no `ACK` delay test performed previously), another experiment was run. It was suspected that Solaris TCP was keeping a global fault counter and dropping the connection when this counter reached some threshold. This suspicion was raised when the following sequence of events was observed in an earlier experiment. In the sequence, `A` and `B` are machines, and `A` is sending messages to `B`. `B` responds with `ACK` messages which are delayed by the *PFI*. At some point, the fault injector on `B` starts dropping incoming messages. After this point, the TCP on `B` does not receive any more messages, and so no `ACK`s are sent. `A` eventually times out the connection. The interesting thing is that `A` times out the connection after fewer retransmissions of message `m2` than expected (six as opposed to nine).

$A \overset{m1}{\rightarrow} B$
$B \overset{ACKm1}{\rightarrow} A$ `(delayed)`

`PFI layer on` B `started dropping incoming messages`

$A \overset{m1}{\rightarrow} B$ `(retransmit)`
$A \overset{m1}{\rightarrow} B$ `(retransmit)`

A `now receives ACK which was delayed`

$A \overset{m2}{\rightarrow} B$
$A \overset{m2}{\rightarrow} B$ `(retransmit)`
$A \overset{m2}{\rightarrow} B$ `(retransmit)`
$A \overset{m2}{\rightarrow} B$ `(retransmit)`
$A \overset{m2}{\rightarrow} B$ `(retransmit)`
$A \overset{m2}{\rightarrow} B$ `(retransmit)`
$A \overset{m2}{\rightarrow} B$ `(retransmit)`

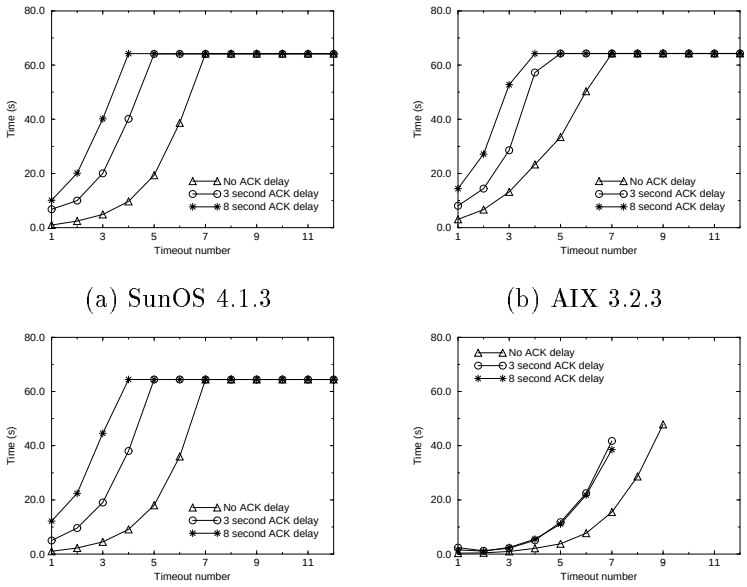
A `drops connection`

Based on this behavior, a new experiment was constructed which attempted to increase the number of times which `m1` was retransmitted, and decrease the number of retransmissions for `m2`. In this experiment, thirty packets were allowed through, and then the initial transmission of a new segment (`m1`) was `ACK`ed with a 35 second delay. All subsequent incoming packets were dropped by the *PFI* layer on `B`. The hope was that multiple retransmissions of `m1` would be seen before the `ACK` arrived at the sender. Then, only several retransmissions of `m2` would occur before the connection was dropped, confirming the global counter theory.

This was exactly what happened. `M1` was retransmitted six times before the original `ACK` arrived. `M2` was then transmitted three times before the connection was dropped.

The active probing method proposed by Comer & Lin [7] cannot discover behaviors such as this one. In their approach, only crash failures are generated. Long delays of specific messages are not possible and so a situation where the initial packet is retransmitted multiple times before the `ACK` is received does not occur[2]. Because of this, connections are not observed to close after only several retransmissions of a packet, leaving interesting implementation details such as the global error counter undiscovered.

The results for the eight second delay variation of this experiment were essentially the same as the three second delay case. The three BSD derived implementations behaved as expected by adjusting their RTO values to account for apparent network slowness. The Solaris RTO seemed to be unaffected by the increased `ACK` delays. This was simply more evidence that Solaris either does not use Jacobson's algorithm or is selecting the RTT measurements in a way that it different from the other implementations. Graphs of the three second, eight second, and no `ACK` delay experiments are shown below. Table 2 also summarizes the results of this experiment.



(a) SunOS 4.1.3 (b) AIX 3.2.3

(c) NeXT Mach (d) Solaris 2.3

Figure 4: Retransmission timeout values

Experiment 3: Keep-alive Test

This experiment examines the sending of keep-alive probes in different TCP implementations. There is no provision in the TCP specification for probing idle connections in order to check whether they are still active. However, many TCP implementations provide a mechanism called keep-alive which sends probes periodically that are designed to elicit an `ACK` from the peer machine. If no `ACK` is received for a certain number of probes in a row, the connection is assumed dead and is reset and dropped. The TCP specification states that by default keep-alive must be turned off, and that the threshold time before which a keep-alive is sent must be 7200 seconds or more.

In this experiment, the receive filter of the *PFI* layer was configured to drop all incoming packets. The sending machine (the machine for which keep-alive was being tested), opened up a connection to the machine running

[2]Comer & Lin did show that for a crash failure, a packet is retransmitted nine times before the connection is dropped. We duplicated this result in a previous section

	Results	Comments
SunOS 4.1.3	Retransmissions started from 6.5 seconds for three second delay case, accounting for apparent network delays.	
AIX 3.2.3	Started retransmitting at 8 seconds.	
NeXT Mach	Started retransmitting at 5 seconds	
Solaris 2.3	Started retransmitting at 1.2 seconds.	Did not use Jacobson's algorithm for computing the RTO estimate or did not use Karn's algorithm for selecting RTT measurements. Used a global error counter to determine when to drop the connection.

Table 2: TCP Retransmission Timeouts with Delayed `ACK`s

the *x*-Kernel machine and turned on keep-alive. The receive filter script in the *PFI* layer was configured to log all incoming packets with a timestamp and then drop them.

The SunOS 4.1.3 machine sent its first keep-alive 7202 seconds after the connection was opened. The packet was dropped and was retransmitted 75 seconds later. After retransmitting the keep-alive a total of eight times at 75 second intervals, the Sun sent a TCP reset and dropped the connection. No further traffic was observed from the connection after this time. The format of the Sun keep-alive packet was SEG.SEQ = SND.NXT - 1 with 1 byte of garbage data. That is to say, the sender sent a sequence number of one less than the next expected sequence number, with one byte of garbage data. Since this data has already been received (because the window is past it), it should be acked by any TCP which receives it. The byte of garbage data is used for compatability with older TCPs which need it.

The AIX 3.2.3 machine sent the first keep-alive 7204 seconds after the connection was opened. The keep-alive packet was dropped, and eight keep-alives were then retransmitted at 75 second intervals, all of which were dropped. After not receiving `ACK`s for any of the keep-alives, the sender sent a TCP reset and dropped the connection. The format of the AIX keep-alive packet was SEG.SEQ = SND.NXT - 1 with 0 bytes of data. The NeXT Mach implementation had the same behavior and used the same type of keep-alive probe as the RS/6000. Note that the keep-alive sent by NeXT and AIX did not contain the one byte of garbage data.

The Solaris 2.3 implementation performed differently than the others. The Solaris machine sent the first keep-alive 6752 seconds after the connection was opened. The keep-alive was dropped, and the Solaris TCP retransmitted it almost immediately and it was dropped again. Keep-alive probes were retransmitted with with exponential backoff, and the connection was closed after a total of seven retransmissions. It should be noted that by sending the initial keep-alive packet at 6752 seconds after the connection was opened, the Solaris TCP violated the TCP specification which states that the threshold must be 7200 seconds or more.

In a variation on this experiment, the incoming keep-alive packets were examined to determine the interval between keep-alive probes. The probes were not dropped by the *PFI* layer, so the connections stayed open for as long as the experiments ran. The SunOS 4.1.3, AIX 3.2.3, and the NeXT Mach machine continued transmitting keep-alive packets at ~7200 second intervals as long as the keep-alives were `ACK`ed. Solaris sent probes at 6752 second intervals. The test was run for eight hours (four keep-alives) on the SunOS 4.1.3 machine, 14 hours (seven keep-alives) on AIX 3.2.3, 20 hours (10 keep-alives) on NeXT Mach, and 112 hours (60 keep-alives) on the Solaris 2.3 machine. The results of this experiment are summarized in Table 3.

Experiment 4: Zero window probe test

This experiment examines the sending of zero window probes in different TCP implementations. The TCP specification indicates that a receiver can tell a sender how many more octets of data it is willing to receive by setting the value in the window field of the TCP header. If the sender sends more data than the receiver is willing to receive, the receiver may drop the data (unless the window has reopened). Probing of zero (offered) windows MUST be supported [4, 25] because an `ACK` segment which reopens the window may be

	Results	Comments
SunOS 4.1.3	First keep-alive arrived at about 7200 second mark. When dropped, the keep-alive was retransmitted eight times before the connection was dropped. When the keep-alives were not dropped, they continued to be sent at 7200 second intervals.	
AIX 3.2.3	Same as SunOS.	
NeXT Mach	Same as SunOS.	
Solaris 2.3	Sent first keep-alive at 6752 seconds. When dropped, the keep-alive was transmitted with exponential backoff seven times before the connection was dropped. When keep-alives were not dropped, they were sent at 6752 second intervals indefinitely.	Specification was violated by the sending of keep-alives at < 7200 second intervals.

Table 3: TCP Keep-alive Results

lost if it contains no data. The reason for this is that `ACK` packets which carry no data are not transmitted reliably. "If zero window probing is not supported, a connection may hang forever when an `ACK` segment that re-opens the window is lost."

This test determined how the SunOS 4.1.3, Solaris 2.3, AIX 3.2.3, and NeXT implementations of TCP perform zero window probing. The machine running the *x*-Kernel was configured such that when the driver layer received data, it did not reset the receive buffer space inside the TCP layer. The result was a full window after several segments were received. Incoming zero-window probes were `ACK`ed, and retransmissions of zero window probes was logged. On all implementations except Solaris 2.3, the retransmission timeout of zero-window probes exponentially increased and leveled off at 60 seconds. Solaris used a 56 second upper bound for the timeout value. As long as the probes were acked, they continued to be sent.

A variation on the zero window probe experiment was also performed. It was the same as the original experiment, except that as soon as x-injector advertised a zero window, the receive filter started dropping incoming packets. The expectation was that the connection would eventually be reset by the sender because no `ACK`s were received for the probes.

Even though the zero-window probes were not `ACK`ed, the SunOS, AIX, and NeXT Mach machines all continued sending probes at 60 second intervals and appeared as if they would do so indefinitely. Solaris did the same at 56 second intervals. The test was allowed to continue for 90 minutes on all machines. This behavior could be a problem because if a receiving TCP which has advertised a zero window crashes, the sending machine could stay in a zero-window probing state until the receiving TCP starts up again and sends a RST in response to a probe. In order to make sure whether this was in fact the case, the same experiment was performed, but once a steady state of sending probes was established, the ethernet was unplugged from the x-injector machine. Two days later, when the ethernet was reconnected, the probes were still being sent by all four machines. The results of this experiment are summarized in Table 4.

Experiment 5: Reordering of messages

This experiment examines how different TCP implementations deal with messages which are received out of order. When a TCP receives segments out of order, it can either queue or drop them. The TCP specification in RFC-1122 states that a TCP should queue out of order segments because dropping them could adversely affect throughput. In this test, the send filter of the fault injection layer was configured to send two outgoing segments out of order, and the subsequent packet exchange was logged. In order to make sure that the second segment would actually arrive at the receiver first, the first segment was delayed by three seconds

	Results	Comments
SunOS 4.1.3	Zero window probes were retransmitted with exponential backoff until a 60 second upper bound was reached. Then they were transmitted at 60 second intervals indefinitely, whether they were ACKed or not.	While not a specification violation, it seems that transmitting zero window probes forever even when they are not ACKed could pose a problem.
AIX 3.2.3	Same as SunOS.	
NeXT Mach	Same as SunOS.	
Solaris 2.3	Same as SunOS except that the upper bound on retransmissions was 56 seconds.	This is interesting, because the ratio of Solaris/Other Vendors for this upper bound is the same as it was for the Keepalive interval[3].

Table 4: TCP Zero Window Probe Results

and any retransmissions of the second segment were dropped.

The result was the same for the Suns running Solaris 2.3 and SunOS 4.1.3, the RS/6000 running AIX 3.2.3, and the NeXT running Mach. The second packet (which actually arrived at the receiver first), was queued. When the data from the first segment arrived at the receiver, the receiver acked the data from both segments.

4.2 Testing of GMP

The objective of the experiments described in this subsection is to test the fault-tolerance capabilities of a prototype implementation of the *strong group membership protocol* [18] using the probe and fault injection technique presented earlier. In a distributed environment, a collection of processes (or processors) can be grouped together to provide a service. A server group may be formed to provide high-availability by replicating a function on several nodes or to provide load balancing by distributing a resource on multiple nodes. A group membership protocol (GMP) is an agreement protocol for achieving a *consistent* system-wide view of the operational processors in the presence of failures, i.e., *determining who is up and who is down*. The membership of a group may change when a member joins, a member departs, or a member is perceived to depart due to random communication delays. A member may depart from a group due to a normal shutdown, such as a scheduled maintenance, or due to a failure. The group membership problem has been studied extensively in the past both for synchronous and asynchronous systems, e.g., [8, 22, 26]. A detailed exposition of this problem is beyond the scope of this presentation.

Informally, the strong group membership protocol, as described in [18], ensures that membership changes are seen in the same order by all members. In this protocol, a group of processors have a unique leader based on the processor id of each member. When a membership change is detected by the leader of the group, it executes a 2-phase protocol to ensure that all members agree on the membership[4]. The leader sends a `MEMBERSHIP_CHANGE` message when a new group is being formed. A processor, upon receiving this message, if the message is from a valid leader, removes itself from its old group. At this point, the group of this processor is said to be in a `IN_TRANSITION` state, i.e. it is a member in transition from one group to another. This processor then sends an `ACK` message to the leader. The leader, after collecting either `ACK`s or `NAK`s from all the members, or when it has timed out waiting, determines what the membership of the new group will be. It then sends out a `COMMIT` message containing the group membership to all the members. The important aspects of this protocol are that the group changes are acknowledged, and that for some period of time, all the members that will be in a new group are in transition.

[3] That is to say, $56/60 \approx 6752/7200$, suggesting that the Solaris implementation has somehow scaled its upper bounds for retransmissions. One suggestion for why this is happening is that timers which the TCP implementation is depending on are not quite correct. For example, the TCP could rely on seeing 7200 ticks of a one second timer between sending keep-alives. If this one second tick actually occurred every .938 seconds, keep-alives would be sent at 6752 second intervals.

[4] The protocol is deceptively simple, but it has a number of subtle properties which are beyond the scope of this presentation

The implementation of the group membership protocol which we tested was developed by a group of three graduate students as part of a project in a course on distributed systems in the Fall Term, 1993. The students were already familiar with SunOS and socket-level programming on TCP/IP. Furthermore, as part of the course project, they performed several extensive tests by instrumenting their code. The implementation of the GMP was written as a user-level server which ran on SUN machines on top of UDP. A Reliable communication layer was implemented using retransmission timers and sequence numbers. In order to test the *group membership daemon* (gmd), we inserted the *PFI* tool into the communication interface code where udp send and receive calls were made. The change is shown in Figure 5. The experiments and results follow. As in TCP, a summary of results appears in a table after the discussion on each experiment.

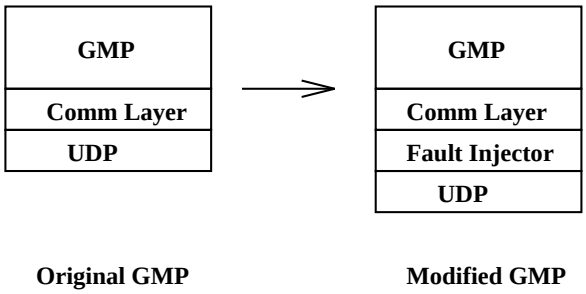


Figure 5: Modified Group Membership Daemon

Experiment 1: Packet interruption

This set of tests involved three machines and various types of packet interruption. The group membership daemons (gmds) were tested for resiliency to delayed or dropped heartbeats, dropped `ACK`s of `MEMBERSHIP_CHANGE` messages, and dropped `COMMIT` messages. The results are presented below.

Group membership daemons normally send heartbeats to each other in order to keep track of who is up and running. If a gmd does not receive heartbeats from another gmd for a period of time, it will declare to the group leader that the other machine is down. As a simple test of this behavior, the send filter script on one of the machines was configured to oscillate between two states. In the first state, heartbeats which the gmd sends were actually sent. In the second state, all outgoing heartbeats were dropped.

Because of some code instrumentation by the implementors of gmp, an error turned up when heartbeats to the local machine were dropped. What happened was that when the local machine did not receive heartbeats from itself, it sent out a message to the other members of the group saying that it had died! However, it did not update its own local state very well and instead of forming a singleton group (a group containing only itself), it stayed in the old group but simply marked itself as down. After this, if someone sent it a `PROCLAIM` message, it would forward it to the group leader, but there was a bug in the code which forwarded the message. A routine was being called with the wrong type of parameter, which resulted in the packet not being forwarded at all. Even though they had instrumented the code, the implementors of the gmp did not find this error. The reason was that they did not perform this type of test. Even when this bug was fixed, because the local gmd did not correctly update its state when it believed itself "dead", it would continue to send bad information to the other gmds. The other machines were not resilient to this type of failure, which was a serious implementation problem. Timing failures on the local machine could actually cause the machine to go "haywire" and have a detrimental effect on the global state of the entire system. The implementors should have coded for the case in which the machine that has "died" is the local machine. Identical behavior was observed when a gmd was suspended for 30 seconds[5]. When it was un-suspended, it's timers had expired and the same bugs were observed.

Since the group membership daemons could not handle dropped heartbeats to themselves, another test was performed. The test was the same as the previous test, but instead of dropping all heartbeats in the second

[5] This was done by sending a SIGTSTP to the running program by typing a <Ctrl>-Z in the shell where the program was running. It was put back into the foreground 30 seconds later by typing fg into the shell.

state, only heartbeats to other members of the group were dropped. The result was that the machine which was dropping heartbeats kept getting kicked out of the group even though it was still active. The machine would then form a singleton group, and then would try to join the others again. It would be admitted to a new group containing all machines, and would remain in the group until it started dropping heartbeats again. When it started dropping heartbeats, the cycle would repeat with the "faulty" machine being kicked out of the group again.

A similar experiment was performed in which heartbeats were delayed by ten seconds. The results were exactly the same because delayed heartbeats are like dropped ones. This is because the heartbeat expect timer expires before the delayed heartbeats arrive, having the same effect as dropped packets.

When a new group is formed, there is a two phase commit process. First, the group leader sends a `MEMBERSHIP_CHANGE` message to all prospective members of the new group. It waits for `ACK` messages from the members, and then sends a `COMMIT` message to all machines that it received `ACK`s from. If a machine does not send an `ACK` message in reply to the `MEMBERSHIP_CHANGE` message from the leader, it should never receive a `COMMIT` message and will not be part of the new group. In this test, the receive filter script of the group leader was configured to drop `ACK` messages from one of the machines (compsun1). Expected behavior was that compsun1 never would never get committed into any group.

In the experiment, gmds were started on two machines and allowed to form a group. Then, the gmd on compsun1 was started. It sent `PROCLAIM` messages to the other two machines and received a `PROCLAIM` from the group leader. It replied with a `JOIN` message, and the group leader initiated the change to a new group by sending `MEMBERSHIP_CHANGE` messages to everybody. The `ACK` from compsun1 was dropped by the fault injector on the group leader, and the group leader did not send a `COMMIT` to compsun1. The two original machines formed a group with only themselves in it, and compsun1 stayed in a transitional state. Some time later, compsun1's `MEMBERSHIP_CHANGE` timer expired and it sent out `PROCLAIM` messages to the others and the whole process repeated. Compsun1 was never admitted to a group.

In a variation on the previous test, the receive filter script of compsun1 was configured to drop incoming `COMMIT` packets. The expectation was that a group would be formed containing all machines, but compsun1 would not ever accept the view of the group (because it would not see the `COMMIT`). Since compsun1 would not view itself as in the group, it would not send heartbeats to the other members and would be kicked out of the group.

In the experiment, gmds were started on two machines and allowed to form a group. When compsun1 started running, it sent `PROCLAIM` messages and received a `PROCLAIM` from the group leader in response. Compsun1 then sent a `JOIN` message to the leader. The leader sent out `MEMBERSHIP_CHANGE` messages to all machines and all responded with `ACK`s. The leader then sent `COMMIT`s to everybody. Compsun1 dropped its `COMMIT` message and stayed in the `IN_TRANSITION` state. The other two machines adopted the new group view which contained everybody. After not receiving any heartbeats from compsun1, the leader declared compsun1 dead and formed a new group which did not contain compsun1. Again, some time later, compsun1's `MEMBERSHIP_CHANGE` timer expired and it sent out `PROCLAIM` messages to the others and the process repeated. The results of this experiment are summarized in Table 5.

Experiment 2: Network partitions

The group membership protocol is designed to tolerate network partitions. If a partition occurs, the result should be that separate but non-overlapping groups are formed. In order to test whether this worked or not, several tests were run in which messages between group members were dropped.

In the first test, the send filter scripts were configured to oscillate between two states. In the first state, all outgoing messages that the gmd sends were actually sent. In the second state, the messages were dropped based on destination address. Five machines were involved; they were compsun{1-5}. In the second state, compsun{1-3} could only send messages to each other, and compsun{4,5} were similarly isolated.

When the machines started dropping messages, two active but disjoint groups were formed. One consisted of compsun{1-3}, and the other had compsun{4,5}. After a while, the machines entered the original state again and started transmitting to each other. At this time, a group was formed which contained all machines. A while later, the machines entered the second state and the process repeated.

	Results	Comments
Drop all heartbeats/ suspend gmd	Gmd believes it has died because it does not receive heartbeats from itself. There was also a parameter passing bug in the gmd.	Implementors should have coded for the special case in which the local machine has "died"
Drop most heartbeats	Machine which was dropping outgoing heartbeats kept getting kicked out of the group. When it started sending heartbeats again, it would be re-admitted, only to be kicked out again when it started dropping heartbeats.	Behaved as specified.
Drop `ACK`s of `MEMBERSHIP_CHANGE` messages	The machine dropping the `ACK`s was never admitted to a group	Behaved as specified.
Drop `COMMIT`s	The machine which drops the `COMMIT` packet stayed in the `IN_TRANSITION` state. Everyone else committed it into their view of the group, but since it did not send heartbeats, it got kicked out of the group.	Behaved as specified.

Table 5: GMP Packet Interruption

	Results	Comments
Partition into two groups	Two separate but disjoint groups were formed, and then when heartbeats were again allowed between all machines, a single group formed again. When the heartbeats were again dropped, the cycle repeated.	Behaved as specified.
Leader/CrownP separation	Depending on the timing of dropped heartbeats, there were two possible paths to the same end state. In the end state, the original leader was again the leader, and the original crown prince was not in the group.	Behaved as specified.

Table 6: Network Partition Experiment

In another test, the leader and crown prince were configured to stop sending messages to each other. The crown prince is the machine which is next in line to be the leader if the leader fails. There were two courses of action, but the result was the same for both. In the end, the crown prince was in a singleton group by itself, and everyone else was in a group with the leader. The two possible courses of action were dependent on the ordering of concurrent events.

If the leader sent out the `MEMBERSHIP_CHANGE` for the new group before the crown prince, everybody but the crown prince became part of a new group. The crown prince was never admitted to the new group because it was not able to send a `JOIN` message to the leader.

If the crown prince sent out the `MEMBERSHIP_CHANGE` for the new group first, everybody but the leader joined a group with the crown prince as the new leader. Soon after, however, the original leader sent a `PROCLAIM` to everybody which was received by all machines except for the new leader (the former crown prince). Since the original leader had a lower IP address than the new leader, each machine responded to the original leader with a `JOIN` message. A group was formed which consisted of all machines except for the crown prince. The crown prince was never admitted to this group because it was not able to send a `JOIN` or `PROCLAIM` message to the leader. The results are summarized in Table 6.

	Results	Comments
Proclaim forwarding	When a proclaim was sent to a non leader machine, it was forwarded to the leader. However, instead of the leader responding to the original sender, it responded to the machine which forwarded the message. This caused a proclaim loop.	There was a bug in the proclaim forwarding code. This bug was fixed.

Table 7: Proclaim Forwarding Experiment

Experiment 3: Proclaim forwarding

In the group membership protocol, machines which desire to be in a group send `PROCLAIM` messages to potential members of the group. These messages are either responded to if received by the leader, or forwarded to the leader if received by another group member. When the leader receives a `PROCLAIM`, it should respond to the originator of the `PROCLAIM` with either a `PROCLAIM` of its own or a `JOIN` message (depending on which machine has a lower IP address). In this test, a machine sent a `PROCLAIM` to a machine which was not the group leader. In order to do this, the send filter script of the machine compsun1 was configured to drop `PROCLAIM`s to the group leader so that only the `PROCLAIM` to non-leader machines were actually sent. The expectation was that the `PROCLAIM` would be forwarded to the leader, who would then respond to the `PROCLAIM` originator (compsun1).

The gmds on the two machines were started and allowed to form a group. Compsun1 was then started, and sent `PROCLAIM`s to the other two machines, but the one to the leader was dropped by the send filter script. The crown prince received the `PROCLAIM` and forwarded it to the leader, who responded to the crown prince instead of the original sender with a `PROCLAIM` of its own. Of course, the crown prince simply forwarded the `PROCLAIM` right back to the leader, who responded with a `PROCLAIM`. This created a vicious cycle of `PROCLAIM` sending between the forwarder (in this case the crown prince), and the leader. The original sender of the `PROCLAIM` (compsun1) never received a `PROCLAIM` in response to its `PROCLAIM`, which was a serious problem. The code was fixed so that the group leader always responds to `PROCLAIM` originator instead of the `PROCLAIM` sender, because the sender may only be forwarding the message. The results of this experiment are summarized in Table 7.

Experiment 4: Timer test

The group membership protocol uses timers extensively. There are timers set for sending and receiving heartbeats, sending `PROCLAIM` messages, joining groups, and preparing to commit new groups, among others. It is important that during some phases of the protocol, all timers be unset. For example, it doesn't make sense to time out waiting for a heartbeat message when you are waiting for the `COMMIT` message for a new group. This test exercised the code which unsets the timers when a machine receives a `MEMBERSHIP_CHANGE`. In the test, the receive filter for compsun1 was configured such that it was allowed to join one group. After that, when it received a second `MEMBERSHIP_CHANGE` (when another group was formed) it started dropping all incoming `COMMIT` and heartbeat packets.

To begin the test, compsun1 and the group leader were started and formed an initial group. When a third machine was started later, compsun1 received a second `MEMBERSHIP_CHANGE` and went into a state in which incoming heartbeat and `COMMIT` messages were dropped. Soon after, compsun1 was still in a transitional state, when no timers (except for the `MEMBERSHIP_CHANGE` timer) were supposed to be set. However, compsun1 timed out waiting for a heartbeat message from the leader. This means that the heartbeat expect timer for the leader was not unset when the `IN_TRANSITION` state of the protocol was entered.

It turned out that there was an error in the code which unregisters timeouts. In the procedure, if an argument is NULL, all timeouts of the same type are unregistered. If the argument is non-null, only the first is unregistered. It worked the opposite of how it should have because of a logic error, and was fixed. The results of the experiment are shown in Table 8.

	Results	Comments
Timer test	When a machine enters the `IN_TRANSITION` state, it should unset all timers. By dropping incoming `COMMIT` packets, it was ensured that the machine would stay `IN_TRANSITION` for a long time. During this time, the heartbeat expect timer expired, which should not have occurred. This was an error in the code which unsets timers.	Behaved as specified.

Table 8: GMP Timer Test

5 Related Work

Numerous approaches have been proposed in the past for evaluation and validation of system dependability including formal methods, analytical modeling, and simulation and experimental techniques. Past research closely related to the work presented here can be classified into two areas: (a) network monitor and filter-based approaches; and (b) fault injection techniques.

Packet Monitoring and filtering:

To support network diagnostics and analysis tools, most Unix systems have some kernel support for giving user-level programs access to raw and unprocessed network traffic. Most of today's workstation operating systems contain such a facility including NIT in SunOS and Ultrix Packet Filter in DEC's Ultrix. To minimize data copying across kernel/user-space protection boundaries, a kernel agent, called a *packet filter*, is often used to discard unwanted packets as early as possible. Past work on packet filters, including the pioneering work on the CMU/Stanford Packet Filter [23], a more recent work on BSD Packet Filter (BPF) which uses a register-based filter evaluator [21], and the Mach Packet Filter (MPF) [29] which is an extension of the BPF, are related to the work presented in this paper. In the same spirit as packet filtration methods for network monitoring, our approach inserts a filter to intercept messages that arrive from the network. While packet filters are used primarily to gather trace data by passively monitoring the network, our approach uses filters to intercept and manipulate packets exchanged between protocol participants. Furthermore, our approach requires that a filter be inserted at various levels in a protocol stack, unlike packet filters that are inserted on top of link-level device drivers and below the listening applications.

Another closely related work is the *active probing* approach proposed in a recent paper by Comer and Lin [7] to study five TCP implementations. *Active probing* treats a TCP implementation as a black box, and it uses a set of user-level procedures to probe the black box. Using the NetMetrix protocol analyzer and monitor tools, trace data is gathered and analyzed to reveal characteristics of various TCP implementations. In addition to repeating TCP experiments similar to those reported in [7], our approach allows other tests that are not possible with techniques that are based primarily on monitoring and gathering trace data. In particular, our approach differs from the active probing technique in four major aspects. First, using a fault injection layer below the TCP layer in the *x*-Kernel protocol stack, we are able *to intercept and manipulate* the TCP packets without having access to the SunOS, AIX, NeXT Mach, or Solaris TCP source code. The manipulation of TCP packets allows various operations such as delay, reorder, and selective message loss. Second, our script-driven approach makes writing complex test scripts relatively easy in a short time. A protocol developer can use a combination of predefined filters and user-defined procedures written in C to develop complex scripts. Third, while an approach based on passive monitoring or active probing can simulate crash failures, more complicated failure models such as omission and timing failures are nearly impossible to test using these methods. A richer set of failure models can be tested using the approach presented here. Finally, although our approach can be more intrusive than active probing, it is intended to be the basis for a tool that can be applied to testing application-level services as well as communication protocols. Our experience in testing the fault-tolerance capabilities of the Group Membership Protocol (GMP), as described in Section 4, seems to support this view.

Fault injection approaches:

Various techniques based on fault-injection have been proposed to test fault-tolerance capabilities of systems. Hardware fault-injection [1, 12, 28] and simulation approaches for injecting hardware failures [6, 9, 13] have received much attention in the past. Recent efforts have focused on software fault-injection by inserting faults into system memory to emulate errors [5, 27]. Others have emulated fault-injection into CPU components [19], typically by setting voltages on pins or wires. However, fault-injection and testing dependability of distributed systems has received very little attention until recently [3, 10, 11, 15]. Most of the recent work in this area have focused on evaluating dependability of distributed protocol implementations through statistical metrics. For example, the work reported in [2] calculates fault coverages of a communication network server by injecting physical faults, and it tests certain properties of an atomic multicast protocol in the presence of faults. Other work can be characterized as probabilistic approaches to test generation [3, 10]. The work reported in [15] focuses on CPU and memory fault injection into a distributed real-time system; this approach also allows inducing communication faults with a given statistical distribution that is specified by the system implementor. Finally, the work reported in [10] is closest to the approach proposed here.

Rather than estimating fault coverages for evaluating dependability of distributed systems, this work focuses on techniques for identifying violations of protocol specifications and for detecting design or implementations errors. The proposed research complements the previous work by focusing on deterministic manipulation of messages via scripts that can be specified by the protocol developer. The approach is based on the premise that injecting faults into a protocol implementation requires orchestrating a computation into hard-to-reach states. Hence, deterministic control on ordering of *certain* concurrent messages is a key to this approach.

6 Conclusion

This paper presented a technique for probing and fault injection of distributed protocols. To demonstrate the capabilities of this approach, experiments were performed that tested several implementations of a transport layer communication protocol, TCP, and an application-level protocol, GMP. The advantages of the proposed approach include: portability to different platforms; uniform treatment of network communication and application-level protocols; support for deterministic and probabilistic testing; and support for user-defined test scripts. Ongoing activities on this project are currently focused on three related paths: (i) development of a more elaborate tool with a graphical user interface; (ii) automatic generation of test scripts from a protocol specification; and (iii) experimental studies of other commercial and prototype distributed protocols.

Acknowledgement

We wish to thank the three graduate students in EECS 682 who provided a stable copy of the GMP protocol for us to test. Although our experiments uncovered several subtle implementation errors and violations of protocol specification, their final grades remained the same. Thanks also to Stuart Sechrest for pointing out that $6752/7200 \approx 56/60$.

References

[1] J. Arlat, Y. Crouzet, and J.-C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 348–355, June 1989.

[2] J. Arlat et al. Experimental evaluation of the fault tolerance of an atomic multicast system. *IEEE Trans. Reliability*, 39(4):455–467, October 1990.

[3] D. Avresky, J. Arlat, J.C. Laprie, and Yves Crouzet. Fault injection for the formal testing of fault tolerance. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 345–354. IEEE, 1992.

[4] R. Braden. RFC-1122: Requirements for internet hosts. *Request for Comments*, October 1989. Network Information Center.

[5] R. Chillarege and N. S. Bowen. Understanding large system failures — a fault injection experiment. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 356–363, June 1989.

[6] G. Choi, R. Iyer, and V. Carreno. Simulated fault injection: A methodology to evaluate fault tolerant microprocessor architectures. *IEEE Trans. Reliability*, 39(4):486–490, October 1990.

[7] Douglas E. Comer and John C. Lin. Probing TCP implementations. In *Proc. Summer USENIX Conference*, June 1994.

[8] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, (4):175–187, 1991.

[9] E. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behaviour. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 236–243. IEEE, 1990.

[10] K. Echtle and Y. Chen. Evaluation of deterministic fault injection for fault-tolerant protocol testing. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 418–425. IEEE, 1991.

[11] Klaus Echtle and Martin Leu. The EFA fault injector for fault-tolerant distributed system testing. In *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 28–35. IEEE, 1992.

[12] G. Finelli. Characterization of fault recovery through fault injection on ftmp. *IEEE Trans. Reliability*, 36(2):164–170, June 1987.

[13] K. Goswami and R. Iyer. Simulation of software behaviour under hardware faults. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 218–227. IEEE, 1993.

[14] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*. Addison Wesley, 1993. Second Edition.

[15] Seungjae Han, Harold A. Rosenberg, and Kang G. Shin. DOCTOR: An integrateD sOftware fault injeCTOn enviRonment. Technical Report CSE-TR-192-93, The University of Michigan, December 1993.

[16] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Trans. Software Engineering*, 17(1):1–13, January 1991.

[17] Van Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM*, pages 314–329, August 1988.

[18] Farnam Jahanian, Ragunathan Rajkumar, and Sameh Fakhouri. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, pages 2–11, Princeton, New Jersey, October 1993.

[19] G.A Kanawati, N.A. Kanawati, and J.A. Abraham. FERRARI: A tool for the validation of system dependability properties. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 336–344. IEEE, 1992.

[20] Phil Karn and Craig Partridge. Round trip time estimation. In *Proc. SIGCOMM 87*, Stowe, Vermont, August 1987.

[21] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Winter USENIX Conference*, pages 259–269, January 1993.

[22] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. A membership protocol based on partial order. In *Second Working Conference on Dependable Computing for Critical Applications*, February 1990.

[23] J. Mogul, R. Rashid, and M. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proc. ACM Symp. on Operating Systems Principles*, pages 39–51, Austin, TX, November 1987. ACM.

[24] John K. Ousterhout. Tcl: An embeddable command language. In *Winter USENIX Conference*, pages 133–146, January 1990.

[25] Jon Postel. RFC-793: Transmission control protocol. *Request for Comments*, September 1981. Network Information Center.

[26] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, Montreal, Quebec, August 1991.

[27] Z. Segall et al. Fiat – fault injection based automated testing environment. In *FTCS-18*, pages 102–107, 1988.

[28] K. G. Shin and Y. H. Lee. Measurement and application of fault latency. *IEEE Trans. Computers*, C-35(4):370–375, April 1986.

[29] Masanobu Yuhara, Brian N. Bershad, Chris Maeda, and J. Eliot B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter USENIX Conference*, January 1994. Second Edition.