Probing and Fault Injection of Protocol Implementations

Scott Dawson  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
Ensuring that a distributed system with strict dependability constraints meets its prescribed specification is a growing challenge that confronts software developers and system engineers. This paper presents a technique for probing and fault injection of fault-tolerant 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 implementers. 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 a summary of several extensive experiments that studied the behavior of two protocols: the Transmission Control Protocol (TCP) [4, 22] and a group membership protocol (GMP) [17].

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 nondeterminism 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 [14]. 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.

![Figure 1: (a) Protocol Stacks. (b) Script Interaction.](image)

Determining whether or not a protocol implementation meets its prescribed specification requires orchestrating the system execution in a deterministic manner. 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 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. PFI layer scripts have the ability to recognize different message types, which allows them to 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 of the protocol stack, which we will call the target protocol. The protocol stack is modified so that the target protocol layer is encapsulated between the driver and PFI layers. The driver and PFI layers are then used to examine and to manipulate the messages exchanged between participants in the target protocol. The driver layer is responsible for generating messages and running the test. The PFI layer intercepts all messages entering and leaving the target protocol. The PFI layer can manipulate messages to/from the target protocol layer as they pass through the protocol stack, and can introduce spontaneous messages into the system. These spontaneous messages can be used to probe target protocol participants on other nodes and observe their behavior. During the test, the driver and PFI layers may communicate with each other and are able to coerce the system in particular states.

The reason for having layers 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. Since the PFI layer sits below the target layer, it can drop, delay, and corrupt messages, but it may not be able to generate messages because it cannot manipulate data structures in the target protocol. Generating messages above the target layer ensures that data structures in the target protocol are updated properly.

The driver and PFI layers interpret 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 sent down the protocol stack. It runs another script, called the receive filter, each time a message is 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 propping a participant by introducing a new message into the system.
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. A formal treatment of these failure models is beyond the scope of this presentation.

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 types 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 3, we 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/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 z-Kernel running on Mach 3.0 and later ported to SunOS. The tool may be inserted into a protocol stack as a separate layer below a target protocol. Figure 2 illustrates the components of the PFI tool: send/receive scripts, filters (or interpreters), recognition/generation stubs, user-defined procedures, and common procedures.

![Figure 2: Probe/Fault Injection Tool](image)

Send/receive scripts are the instructions for orchestrating a computation into a particular path and for injecting faults into a system. Tcl is the language used for writing PFI layer scripts. Filters are the interpreters that execute the scripts as messages pass through the PFI layer. Each time a message passes into the PFI layer, the appropriate (send or receive) script is interpreted (or run) and is able to inject faults into the system by manipulating the message.

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

```tcl
# Message types are ACK, NACK, and GACK.
set ACK 0x1
set NACK 0x2
set GACK 0x4

# Print out message contents
puts -nonewline "receive filter: ": msg_log cur_msg

# Get msg type and drop if ACK
set type [msg_type cur_msg]
if {type == "$ACK"} {
    xDrop cur_msg
}
```

Scripts are able to maintain state across receipt of multiple messages. For example, a script may keep a running count of the number of messages which have passed through the PFI layer, and use the count to decide when to begin dropping messages.

Another component of the tool (besides script filters) is a set of utility procedures. These utility procedures are used by the scripts, and can perform several types of functions. They are:

- Recognition/generation procedures: are used to recognize and generate different types of packets. They allow the script writer to perform different actions based on message type, and also to create new messages to be injected into the system.
- Common procedures: are commonly used procedures used by scripts written for testing any protocol. Procedures which drop or log messages fall into this category. Also included are procedures which can generate probability distributions.
- Communication procedures: allow the different filters (send and receive) to communicate information to each other. These allow one filter script to set variables or state in the other filter.
- User defined procedures: specified by the user of the PFI tool to test his/her protocol. These procedures are typically written in C and are linked into the tool. The user can then write scripts which use the procedures to perform special functions during testing of their protocol.

One of the main benefits of script driven probing and fault injection is that testing different failure scenarios is accomplished simply by using different scripts. Changing scripts does not require recompilation of the tool. This reduces the time required to run multiple of different tests, compared to a system in which some re-compilation is necessary.

## 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 a connection-oriented protocol and 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 [22] and was updated in RFC-1122 [4]. In order to meet the TCP standard, an implementation must follow both RFCs.

To test vendor TCP implementations, we modified an x-Kernel protocol stack to include a layer which incorporates the PFI tool described in Section 3. The PFI layer sits directly between the TCP and IP layers of the protocol stack. The resulting protocol stack is shown in Figure 3. The figure shows one machine running Mach with the modified x-Kernel protocol stack. This machine is connected to a network which has machines running vendor TCP implementations. For these experiments, connections are opened between the vendor TCP implementations and the x-Kernel TCP. Faults are then injected into the system from the x-Kernel protocol stack and the behavior of the vendor implementations is monitored.

We ran experiments on four different vendor implementations of TCP; implementations from SunOS 4.1.3, Solaris 2.3, AIX 3.2.3, and NeXT Mach, which is based on Mach 2.5. The results were similar for the SunOS, AIX, and NeXT Mach implementations, which are based on BSD Unix. Solaris, which is a System V implementation, behaved somewhat differently than the others in most experiments. A brief description of five experiments and their results follows. A more detailed description can be found in [9].

**Experiment: TCP retransmission intervals**

This experiment examines how different implemen-
tations 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 acknowledgment is not received before the expiration of the timeout, the data is assumed lost and is retransmitted. The TCP specification states that for successive retransmissions of the same segment, the retransmission timeout (RTO) should increase exponentially, and that an upper bound on retransmissions may be imposed.

In this experiment, vendor TCP implementations were tested in order to determine how they respond to a broken network connection. A connection was opened from the vendor TCP to the z-Kernel TCP, and 30 packets were allowed through the PF1 layer. After 30 packets had been sent, the PF1 layer started dropping packets. At this point, the vendor TCPs believed the connection dead. They retransmitted the packet some number of times, and closed the connection. For the SunOS, AIX 3.2.3, and NeXT Mach implementations, the number of retransmissions was fixed at 12. For the Solaris implementation, the connection was dropped based on elapsed time since an acknowledge (ACK) packet was last received; the actual number of retransmissions varied. In all implementations, the retransmission timeout increased exponentially, and the BSD based implementations had an upper bound on retransmissions of 64 seconds. In all runs, the Solaris machine closed the connection before any upper bound on retransmissions was reached.

**Experiment: RTO with ACK delays**

This experiment examines how different implementations of TCP adjust the retransmission timeout value in the presence of network delays. The RTO value 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 [16] for computing the retransmission timeout coupled with Karn's algorithm [18] 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.

In the experiment, ACK messages for data segments were delayed. Two variations of the experiment were run. In the first, ACKs were delayed by two seconds; in the second, the delay was eight seconds. The vendor TCP implementation opened a connection to the z-Kernel TCP. The first 30 packets sent by the vendor TCP were delayed, after which the PF1 layer started dropping packets. The result was the same with all vendor TCP implementations. The TCP adjusted for the apparent network delay by raising the retransmit timeout (RTO). The Solaris TCP took more packets to adapt to the network delay. (In order to get the Solaris TCP, a setup period of around 200 packets was used.) The reason for this is that Solaris has a very short lower bound on the RTT estimate because it is tuned for high speed LANs. The short lower bound results in a slower convergence to a good RTO value because Karn's algorithm discards many of the RTT estimates. This experiment also determined an upper bound on the RTO value for Solaris of 56 seconds. The RTO started at a higher point because of the delayed ACK packets, so the upper bound was reached before the connection timed out.

**Experiment: 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, keepalive was turned on for connections between the vendor TCPs and the z-Kernel TCP. There were two variations on the experiment. In
the first variation, all incoming packets were dropped in order to observe TCP behavior when keepalive are dropped. In the second, keepalives were not dropped, allowing the time interval between keepalive packets to be established. The BSD implementations behaved similarly, sending keepalive packets at about two hour intervals. Solaris used a 6750 second interval, which is about 1 hour and 52 minutes. The BSD implementations retransmitted non-ACKed keepalives eight times at 75 second intervals before dropping the connection. Solaris retransmitted the keepalives with exponential backoff and then dropped the connection, using the same method for timing out connections as in the non-keepalive case.

Behavior of the Solaris keepalive mechanism differed based on when keepalive was turned on for the connection (before or after connecting the socket). When keepalive was turned on before connecting, the first keepalive was sent two keepalive periods after the connection was opened (3 hours and 44 minutes). When keepalive was turned on after connecting, the first keepalive was sent after one keepalive period (which was the expected behavior). In both cases, subsequent keepalives continued at 1 hour and 52 minute intervals. This difference in behavior is probably an implementation bug.

**Experiment: 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. Probing of zero (offered) windows MUST be supported [4, 22] because an ACK segment which reopen the window may be lost if it contains no data, since 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.”

In this test, vendor TCPs opened connections to the z-Kernel TCP. A zero window condition was allowed to occur in the z-Kernel TCP (by not receiving data from the TCP). The BSD implementations responded by sending zero window probes every 60 seconds. Solaris sent probes every 56 seconds. An interesting fact

---

1 The fact that a difference in behavior existed was pointed out by Jerry Toporek at Mentat Corporation. Tests confirmed this and determined what the difference was.

2 It is interesting to note that the ratio of inter-arrival times between Solaris and the BSD implementations is the same here as in the keepalive test. This suggests that the implementors may be depending on certain timing behavior which is not being provided faithfully.
was that even if probes were not ACKed, they continued to be sent. In a variation on this experiment, the network was disconnected after a zero window occurred. Two days later, when the network was reconnected, the vendor TCPs were still sending probes. Although not a violation of the TCP specification, this doesn’t make much sense. It would seem that if probes are not ACKed, the connection should be timed out as any other connection would be.

**Experiment: 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 and any retransmissions of the second segment were dropped.

The result was the same for all vendor implementations. 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 was to test the fault-tolerance capabilities of a prototype implementation of the *strong group membership protocol* [17] 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, departs, or 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., [7, 20, 23]. A detailed exposition of this problem is beyond the scope of this presentation.

Informally, the strong group membership protocol, as described in [17], 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. 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 ACKs or NAKs 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.

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. We then performed tests which interrupted (dropped or delayed) packets such as heartbeats or `MEMBERSHIP_CHANGE`es, simulated network partitions, and tested specific functionality of the gmp daemons. While performing these tests, we found several implementation errors and protocol specification violations which went unnoticed during testing by the implementors. An in-depth explanation of the tests and their results appears in [9].

### 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 re-
search closely related to this work 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 [21], a more recent work on BSD Packet Filter (BPF) which uses a register-based filter evaluator [19], and the Mach Packet Filter (MPF) [26] 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 [6] 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 NetMetrax 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 [6], 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 z-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 (e.g. [1, 25]) and simulation approaches for injecting hardware failures (e.g. [8, 12]) 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, 24]. However, fault-injection and testing dependability of distributed systems has received very little attention until recently [3, 10, 11, 13]. 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 [13] 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. The Delaysine tool presented in [15] allows the user to introduce delays into user-level protocols. The tool is used mainly for emulating a wide-area network in a local network development environment and allows the user to specify delays on certain paths which the application is using. This work and that reported in [10] are 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.

Acknowledgment

We are very grateful to Jerry Toporek at Mentat for valuable discussions on the TCP experiments reported in this paper. His insightful comments helped us in improving this work. We also wish to thank the three graduate students in EECS 682 who provided a stable copy of the GMP protocol for us to test. In addition, Stuart Sechrest pointed out that 6752/7200 ≈ 56/60!

References