ORCHESTRA: A Fault Injection Environment for Distributed Systems

Scott Dawson, Farnam Jahanian, and Todd Mitton

Real-Time Computing Laboratory
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109-2122
USA
Tel 313.936.2974 Fax 313.763.1503
{sdawson,farnam,mitton}@eecs.umich.edu

ABSTRACT

This paper reports on ORCHESTRA, a portable fault injection environment for testing implementations of distributed protocols. The paper focuses on architectural features of ORCHESTRA that provide portability, minimize intrusiveness on target protocols, and support testing of real-time systems.

ORCHESTRA is based on a simple yet powerful framework, called script-driven probing and fault injection, for the evaluation and validation of the fault-tolerance and timing characteristics of distributed protocols. ORCHESTRA was initially developed on the Real-Time Mach operating system and later ported to other platforms including Solaris and SunOS, and has been used to conduct extensive experiments on several protocol implementations. A novel feature of the Real-Time Mach implementation of ORCHESTRA is that it utilizes certain features of the Real-Time Mach operating system to quantify and compensate for intrusiveness of the fault injection mechanism. In addition to describing the overall ORCHESTRA architecture and implementation, this paper also describes the experimental evaluation of two protocol implementations: a distributed group membership service on the Sun Solaris operating system, and a real-time audio-conferencing application on Real-Time Mach.

Keywords: fault injection, distributed systems, communication protocols, real-time systems.

This is a substantially expanded revision of a paper that appeared in the 26th International Symposium on Fault-Tolerant Computing (FTCS)[12].
This work is supported in part by a research grant from the U.S. Office of Naval Research, N0014-95-1-0261, and a research grant from Defense Advanced Research Projects Agency, monitored by the U.S. Air Force Rome Laboratory under Grant F30602-95-1-0044.

1 Introduction

Ensuring that a distributed system meets its prescribed specification is a growing challenge that confronts software developers and system engineers. Meeting this challenge is particularly important for applications with strict dependability and timeliness constraints. This paper reports on a software fault injection tool, called ORCHESTRA, for testing dependability and timing properties of distributed protocols. ORCHESTRA is based on a simple yet powerful framework, called *script-driven probing and fault injection*, first reported in [9, 10]. The emphasis of this 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 (e.g. [1]). Hence, the focus is on developing fault injection techniques that can be employed in studying three aspects of a target protocol: i) detecting design or implementation errors, ii) identifying violations of protocol specifications, and iii) obtaining insights into the design decisions made by the implementors.

The proposed approach is motivated by several observations. First, 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 by ordering certain concurrent events. This is further complicated by asynchronous message communication and inherent non-determinism of distributed computations. Second, when 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. Third, testing organizations often require a methodology that does not instrument the code being tested. This is particularly important when testing existing systems or when the source code is unavailable.

In the model underlying script-driven probing and fault injection, a distributed protocol can be viewed as an abstraction through which a collection of participants communicate by exchanging a set of messages, in the same spirit as the *x*-kernel [20]. Hence, no distinction is made between application-level protocols, communication protocols, or device layer protocols. In this approach, a protocol fault injection (PFI) layer is inserted below a target protocol to filter and manipulate the messages that are exchanged between participants. The fault injection layer supports the execution of deterministic or probabilistic test scripts that may probe participants or inject faults into the messages that are exchanged between participants. In particular, by intercepting messages between two layers in a protocol stack, the fault injection layer can delay, drop, reorder, duplicate, and modify messages. Furthermore, it can introduce spontaneous messages into the system to probe the participants and to orchestrate the system execution onto a particular path.

This paper describes the design of the ORCHESTRA fault injection tool, focusing on architectural features supporting portability and minimizing intrusiveness on target protocols, with explicit support for testing real-time systems. The current implementation of the ORCHESTRA fault injection tool was initially developed on the Real-Time Mach operating system and later ported to other platforms including Solaris and SunOS. This tool has been used to conduct extensive experiments on several implementations of commercial and prototype systems. This paper describes the implementation of ORCHESTRA on Real-Time Mach and Solaris operating systems for testing socket-based distributed applications. The paper also reports on the experimental evaluation of two protocol implementations: a distributed group membership service on the Sun Solaris operating system, and a real-time audio-conferencing application on Real-Time Mach.

Since ORCHESTRA is intended to be a tool for testing distributed applications and communication

protocols, portability to different platforms and the ability to insert the fault injection engine into a protocol stack are key objectives. A novel aspect of the ORCHESTRA architecture is a clean separation of the fault injection mechanism from target protocol and platform dependent code. Because most of the actions of the fault injection layer are independent of the target protocol implementation, we have been able to split our fault injection tool into two parts: a protocol independent part and a protocol dependent part. The protocol independent part consists of the actual fault injection engine and its associated data structures. It also includes the user interface for generating fault injection scripts. The fault injection engine performs the same actions no matter where it is placed in the protocol stack. The protocol dependent part consists of "glue" code to hook the engine into the protocol stack. This is the code which exports the proper interfaces from the fault injection engine to higher and lower layers, and also special routines which are invoked to get messages into and out of the engine.

The ORCHESTRA fault injection tool is designed so that the application programmer is not required to instrument the target protocol implementation, although the protocol stack may be altered. Fault injection is done at the message-level by intercepting and manipulating incoming/outgoing messages of a target protocol, or by probing a participant by injecting spontaneous messages into the system. Furthermore, the ORCHESTRA engine interprets the fault injection test scripts written in the *Tcl* language [30] or via a state-transition-based graphical user interface. Because Tcl is interpreted, one can develop relatively complex scripts to modify existing experiments or to perform new tests without re-compiling the target protocol or the fault injection tool.

Another novel architectural feature of ORCHESTRA is that it is designed to address explicitly the intrusiveness of fault injection on a target distributed system. While the intrusiveness issue is significant for both the logical correctness as well as the timing correctness of an implementation, it is particularly significant in *hard* real-time systems where timing predictability may be disturbed by the additional overhead of a fault injection mechanism. For example, in the implementation of ORCHESTRA on Real-Time Mach, we exploit operating system support, such as *scheduler feedback and capacity reservation* [27], to quantify intrusiveness on a target protocol and to compensate for it whenever possible.

The remainder of this paper is organized as follows. Section 2 presents the overall ORCHESTRA framework, covering the script-driven probing and fault injection technique, the ORCHESTRA architecture, and specification of fault injection scripts. In addition, this section discusses the issue of intrusiveness on a target protocol, and general methods of minimizing or compensating for it. Section 3 looks at the realization of the ORCHESTRA architecture on two different platforms: the Solaris[1] operating system from SUN Microsystems, and the Real-Time Mach operating system from CMU. In particular, this section looks at the implementation of the ORCHESTRA fault injection tool for testing socket-based distributed applications. Section 4 describes in detail the experimental results from studying an asynchronous fault-tolerant protocol on Solaris and a real-time audio-conferencing application on Real-Time Mach. Section 5 presents a graphical script editing tool that can be used to generate test scripts via a graphical user interface. Section 6 discusses the related work. Finally, Section 7 presents concluding remarks.

2 ORCHESTRA Framework

This section discusses the framework on which ORCHESTRA is based. The first subsection describes the overall approach for performing fault injection on messages that are exchanged between

[1] Solaris is a trademark of Sun Microsystems, Inc.

the participants in a distributed protocol. The second subsection presents the basic architecture of ORCHESTRA and a portable fault injection core that can be used to build fault injection layers on different platforms. The next subsection discusses how fault injection scripts are specified in ORCHESTRA. Finally, intrusiveness of fault injection on target protocols is covered, and methods of minimizing and compensating for intrusiveness are presented.

2.1 Approach: Script-Driven Probing and Fault Injection

In script-driven probing and fault injection, a distributed protocol can be viewed as an abstraction through which a collection of participants communicate by exchanging a set of messages, as in the *x*-kernel [20]. Each layer provides an abstract communication service to higher layers, and there is no distinction made between particular layers of the protocol stack. In our approach, a protocol fault injection layer (*PFI* layer) is inserted into a protocol stack below the *target layer*, which is the layer to be tested. The *PFI* layer can intercept messages that pass through it, and is able to manipulate these messages as the protocol participants communicate. For example, the *PFI* layer can drop messages of certain types, delay a message for a pre-specified time interval, retransmit a message, modify a message content, or introduce new messages into the protocol stack.

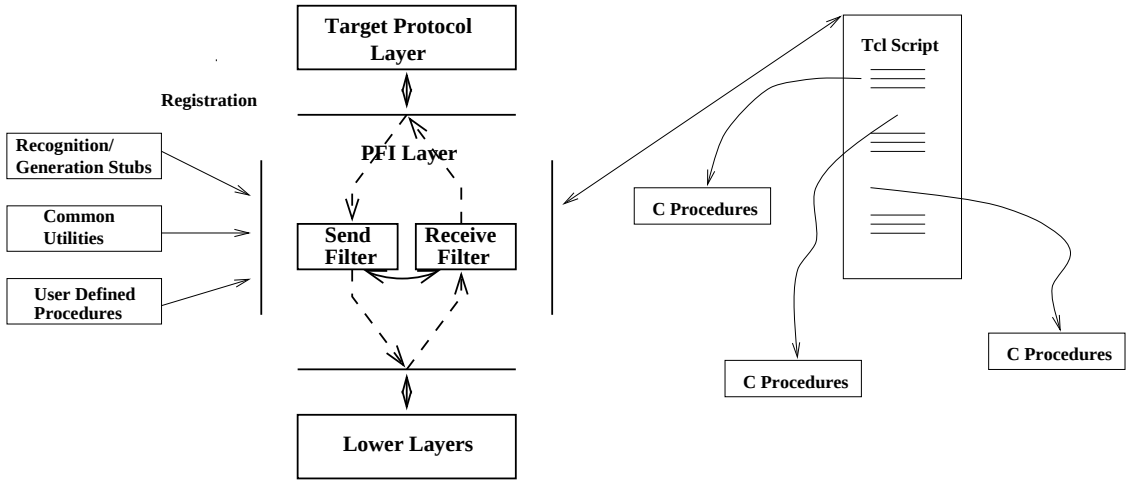

Figure 1: Protocol Stack with PFI Layer

Figure 1 shows a *PFI* layer inserted into a protocol stack in order to test the target protocol sitting above it. As messages are exchanged between protocol participants, they pass through the *PFI* layer on their path to/from the network. Each time a message is sent, the *PFI* layer runs a script called the *send filter* on the message. In the same manner, the *receive filter* is invoked on each message that is received from the network destined for the target protocol. The 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 send and receive filter scripts determine what action is to be taken on a message when it is sent or received, with possible actions being taken from the above list. Scripts may base their

decisions on message attributes such as message type or contents, on message history, or on certain data that the fault injection layer has collected (such as counters). For instance, the *PFI* layer may delay a message of a particular type if a certain sequence of messages has been received several times. An important feature of the scripts is that they may make calls to procedures that are written in C or another programming language. These utility procedures are registered with the engine that interprets the scripts. Some utility procedures are provided with the system; the user may also extend the functionality of the scripting language by providing their own routines. The various types of utility procedures that can be registered with the interpreter engine are shown in Figure 1. These procedures fall into several classes:

- *Recognition/generation procedures:* are used to identify or create different types of packets. They allow the script writer to perform certain actions based on message type (recognition), and also to create new messages of certain type to be injected into the system (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 a widely-used communication protocol such as TCP.

- *Common utilities:* are procedures frequently used by script writers for testing a protocol. Procedures that drop or log messages fall into this category. Also included are procedures that can generate probability distributions and procedures that give a script access to the system clock and timers.

- *User defined procedures:* are additional utility routines written by the user of the *PFI* tool to test his/her protocol. These procedures, usually written in C, may perform arbitrary manipulations on the messages that a protocol participant exchanges.

2.2 Portable Core Architecture

A novel aspect of the ORCHESTRA architecture is a clean separation of the fault injection mechanism from target protocol and platform dependent code. The main component of ORCHESTRA is the *PFICore* [11, 12], which is responsible for all fault injection actions. The *PFICore* is the heart of the fault injection layer, and provides the mechanism for injecting faults into protocols, regardless of where the protocol resides in the protocol stack. Using the *PFICore*, a fault injection layer is built and then inserted into the protocol stack to test the *target layer*.

To achieve independence of the *PFICore* from the messages it operates on, details of message formats are hidden from it. The *PFICore* views messages as abstract objects. The knowledge of how to interpret messages is provided by the user who builds a fault injection layer around the *PFICore* through the recognition/generation stubs described in the previous subsection. In many cases, the fault injection layer may not even define these message formats, but may leave them up to the user of the fault injection layer[2].

Figure 2 shows the basic structure of a fault injection layer built using the *PFICore*. The layer consists of two main parts. The first is the *PFICore* itself, shown in the center of the layer.

[2]This is in fact how the socket implementation of ORCHESTRA described in Section 3 works. The reason is that the socket fault injector is intended for testing application level protocols. Because packet formats will differ from one protocol to another, message formats cannot be fixed as they could if the fault injection layer were intended to be used to test a specific protocol.

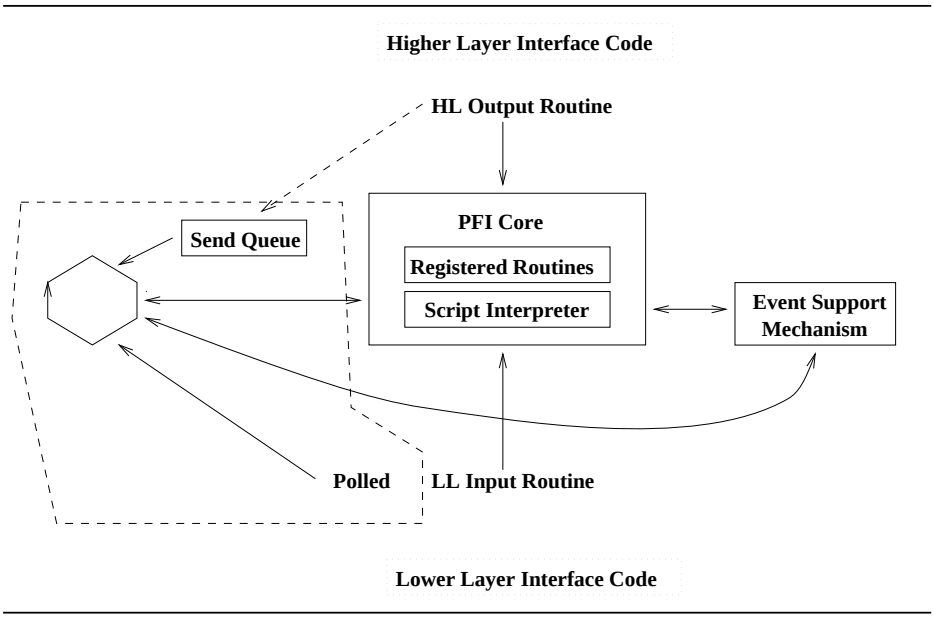

Figure 2: The Orchestra Architecture

It consists of the script interpreter and any routines that have been registered with the script interpreter.

The right side of the figure pictures event support. Event support is provided by the fault injection layer builder, and consists of routines to post and track events. These routines are registered with the *PFICore* so that it can post events such as delayed messages. When the *PFICore* needs to post an event, it makes a call to the event posting routine that has been registered with it. One of the parameters to the event posting is the routine that should be called when the event is ready. The reason that event support is not provided with the *PFICore* is that the event support implementation depends on the system that the *PFICore* is being used in. The fault injection layer designer must use existing support or provide their own support for events.

Also shown in the figure is an optional thread that a *PFI* layer builder may be required to provide. In some cases, the fault injection layer may be placed into a protocol stack that does not provide upcalls when messages arrive, such as in an implementation above the socket layer. In these implementations, the *PFI* layer must poll for arriving messages. Furthermore, in these cases it is usually also true that no event mechanism exists that the fault injection layer designer can use. The separate thread in the *PFI* layer can then be used for polling and event support. A simple form of event support in this case might consist of event queues that the thread maintains.

In order to build a fault injection layer using the *PFICore*, a designer must first address two issues. The first is whether the target protocol stack has upcalls for message arrivals. The second is the type of event support (if any) that is provided in the protocol stack. If a thread is required either for message polling or for event support, it must be provided by the *PFI* layer designer. In addition, if the protocol stack does not contain event support, the *PFI* layer designer must provide their own. If event support exists, wrappers can be written around existing event support and registered with the *PFICore*. Finally, the *PFI* layer builder must provide the interface code for higher and lower layers in the protocol stack, so that as messages are sent and received they will pass through the *PFI* layer. Once this is done, the fault injection layer can be inserted into the

protocol stack.

One example of building a fault injection layer using the portable *PFICore* is an *x*-kernel implementation of the *PFI* layer. Because the *x*-kernel already provides support for events, all that is necessary is to provide a routine for registering events that makes calls to the *x*-kernel event registration routine. This routine is then registered with the *PFICore* so that it can post events such as delayed messages. Because the *x*-kernel has upcalls when messages are received, and no thread is needed for event support, it is not necessary for the *x*-kernel *PFI* layer builder to provide an extra thread for events or message polling. All that is necessary is to write the code for the input and output operations of the *x*-kernel *PFI* layer so that the layer can be inserted into an *x*-kernel protocol stack. These operations simply call the *PFICore* with the message as an argument. If the *PFICore* returns a handle to the message, then the message is sent on to the next layer in the protocol stack. If not, the input or output routine simply returns, and the message becomes the responsibility of the *PFICore*. If it is being delayed or otherwise detained, it will be sent up or down the protocol stack later when its conditions have been met.

2.3 Specification of Fault Injection Scripts

A protocol participant is faulty if it deviates from its prescribed specification. A fault model specifies in what way a protocol participant can deviate from its correct specification. The fault injection approach presented in this section can test the fault-tolerance capabilities of protocol implementations under various 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* [3].

At the heart of our approach are scripts which are executed by the *PFI* layer in order 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 which are exchanged in the course of carrying out a distributed computation.

One can view a fault injection script as a state machine which operates on a message. The state machine has access to the message type and contents, and the ability to track message histories and keep counters or other state. When a new message enters the *PFI* layer, a handle to the current message is set up in the interpreter, and the state machine (i.e., the script) is run through the interpreter. Usually, this message starts off in the initial state of the state machine. In some cases, however, the previous message may have left the state machine in another state, in order to perform a more complex function. Regardless of which state the message begins in, it travels through the state machine, making transitions from one state to another based on information contained in the state machine, such as message type, number of messages seen in a sequence, or recent message history. The message exits the state machine when the execution of the script completes, usually resetting the state to the initial state before doing so. At this point, several things may have happened to this message. For example, the message may have been delayed or dropped which means the interpreter simply moves on to receive the next message. Alternatively, the message may be passed to the next layer in the protocol stack which means the interpreter puts puts the message in the receive queue of the next layer.

We believe that inventing a new language for specifying fault injection scripts is not the best 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

[3] A formal treatment of these failure models is beyond the scope of this presentation [17].

also eases the burden of learning a new language for users already familiar with whatever interpreted language is chosen. As mentioned previously, we use *Tcl* [30] 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. Since a script written in the *Tcl* language can invoke user-defined procedures which can modify the internal state of the protocol, 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.

In the ORCHESTRA fault injection environment, a typical Tcl script which encodes a state machine might look like this:

```
if { [catch {set state}] == 1} {
    set state 0
}

while { 1 } {
    if { $state == 0 } {
        msg_delay 5.0 1
        return
    } elseif { $state == 1 } {
        set state 0
        return
    }
}
```

The first three lines of the script set the state to the initial state the first time the script is interpreted (that is, the first time a message is received or sent through the filter). The operation `msg_delay` delays the message by 5 seconds, and instructs the filter to resume the script in state 1. In this manner, a message can be delayed, and made to resume in its next logical state. It is also possible to name a message and store the name in a list (without sending the message). The script may accumulate a certain number of messages before sending them out to the network or putting them on the application's receive queue. In addition to grouping and delaying messages, the script can also manipulate groups of messages by dropping or reordering messages within the group. Because the script can condition state transitions on message type information, it may also perform actions such as reordering sets of acknowledgment messages. This would be done by transitioning on the message type, and queuing acknowledgment messages, while allowing non-acknowledgment messages to pass normally. The acknowledgment messages can be subsequently reordered and sent on to the next layer in the protocol stack.

In order to make script generation and specification more straightforward, we have also developed a state-transition-based graphical script editor which generates test scripts. In the editor, states are represented by various icons, while state transitions are denoted by arrows. Each state may have a specific action, such as delaying or dropping a message, or updating a state variable. Transitions between states may have conditions attached to them, meaning that the transition is taken only if the condition is true. A state may maintain a counter, for example, and the transition to the next state may be based on the value of the counter. We have found that this editor makes specifying common tests fairly easy, and that tests can be generated in a short time using the tool. The user may also specify Tcl code which can be invoked within each state. This allows the user more flexibility in generating test scripts with the editor by allowing the user to add additional code to tailor activities which occur within a state. Figure 3 shows a simple script specified in the editor. Section 5 provides more detailed information on the ORCHESTRA graphical script editor.

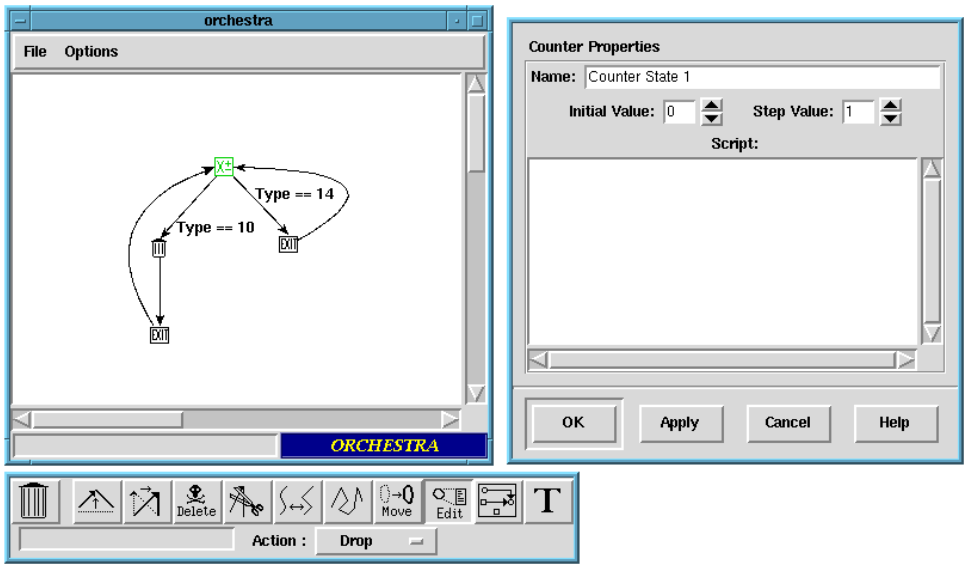

Figure 3: Script Generation Using the GUI

2.4 Intrusiveness on a Target Protocol

An important issue in the experimental evaluation and validation of the fault-tolerance capabilities of distributed and real-time protocols is the intrusiveness of the approach on the target system. The inherent necessity of an instrumentation may introduce a "Heisenberg effect" on the execution of the target system. This is particularly important in real-time systems where timing predictability may be disturbed by the additional overhead of a fault injection mechanism. Two key features of the *script-driven probing and fault injection* approach attempt to minimize the intrusiveness on a target protocol:

- *Message-level fault injection and probing-* Because the fault injection layer is inserted between two levels in the protocol stack, the code of the target layer is not modified.
- *Probing and monitoring from a non-target platform-* Just as passive monitors are used to observe network traffic, our approach can be used to instrument one node in the system. This node can be used to collect data about target participants by probing.

While the intrusiveness issue is significant for both the logical correctness as well as the timing correctness of an implementation, it is particularly significant in *hard* real-time systems where timing predictability may be disturbed by the additional overhead of a fault injection mechanism. One must ensure that unintended errors (including timing faults) are not introduced into the system by the fault injection mechanism during an experiment. Furthermore, the timing predictability of the target system must be preserved.

In a nutshell, in applying the proposed framework to fault injection of real-time protocols, we exploit operating system support to quantify precisely the intrusiveness of a fault injection experiment on the timing behavior of the target system and to compensate for it whenever possible. For example, the implementation of ORCHESTRA on the Real-Time Mach operating system depends on specific kernel support to deal with the intrusiveness issue. More detail on the Real-Time Mach implementation and the exploitation of operating system support to handle timing intrusiveness is provided in Section 3.3.

3 Implementation of ORCHESTRA for Socket-Based Applications

This section describes an implementation of a fault injection tool based on the ORCHESTRA architecture. This tool allows the user to test applications and application level protocols that use UNIX[4] sockets for inter-process communication. The socket-based implementation of ORCHESTRA was developed on both the Solaris and Real-Time Mach operating systems, and was built using the *PFICore* described in Section 2.2. Subsection 3.1 provides an overview of sockets and how they are used for communication on both Solaris and Real-Time Mach. Subsection 3.2 describes the implementation of the ORCHESTRA socket tool, built using the *PFICore*. Subsection 3.3 discusses how specific features of Real-Time Mach are exploited in the ORCHESTRA implementation to deal with timing intrusiveness of fault injection on that platform.

3.1 Socket Structure

Many applications and application level protocols use UNIX sockets to communicate with each other. Sockets provide an interface in which one application may send data to another by specifying an address/port combination. For network sockets, when reliable transmission is not critical, UDP (unreliable datagram protocol) is used. For connection-oriented (reliable) sockets, TCP (transmission control protocol) is used. Applications using sockets invoke several library routines which set up and send/receive data on the socket. These library routines call the kernel or protocol manager through a system call interface. Routines available to applications using sockets include `socket()`, `bind()`, `connect()`, `send()`, `recv()`, etc.

In Real-Time Mach[5], protocol stack operations are provided by the UNIX server process, called the UX server. The UX server keeps track of state such as which address/port combinations are being used by TCP and UDP connections in the system. Socket calls are exported to the user through the system C library, `libc.a`. In addition to having this "kernel" support for sockets, Real-Time Mach also provides a user level socket library, called `libsockets.a`. The purpose of the user level socket library is mainly to increase the speed of send and receive operations. It does this by creating an extra thread in the user task to handle socket operations, and by keeping the protocol stack code within the user application space. For complex operations such as connection setup and teardown, the UX server is still contacted, so that system wide state such as address/port combinations that are in use can be kept track of, but send and receive operations are faster because the UX server is not part of the message path.

For applications which are simply sending and receiving data, significant speedup (over the UX server implementation) has been achieved by using `libsockets.a`. In Real-Time Mach, user level sockets have been used in conjunction with processor reserves to achieve predictable protocol processing [27]. Protocol processing is made predictable by binding the protocol processing thread to a processor reserve so that sufficient time is set aside for communication. In addition, because the send/receive path is contained within the application, protocol operations take place at the priority of the process, alleviating problems with priority inversion due to FIFO scheduling within the UX server.

In Solaris, the protocol stack resides in the kernel. The socket interface is provided by a library, called `libsocket.a`[6], which uses system calls to get data into and out of the kernel.

Figures 4(a) and 4(b) show the structure of the Solaris and Real-Time Mach socket implemen-

[4] UNIX is a registered trademark of UNIX System Laboratories, Inc.

[5] Mach 3.0 socket structure is identical to Real-Time Mach.

[6] Not to be confused with the Real-Time Mach `libsockets.a`.

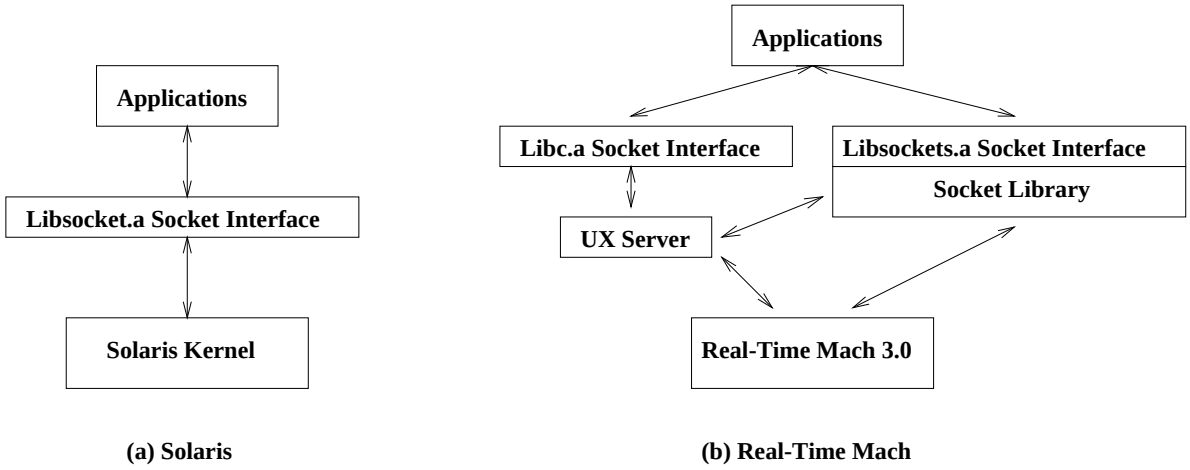

Figure 4: Socket Structure

tations. As the figure shows, on Solaris, applications use the `libsocket.a` interface to access the protocol stack in the Solaris kernel. On Real-Time Mach, applications may use socket routines provided by either the standard C library or by the socket library, `libsockets.a`. On Real-Time Mach, both the socket library and the UX server access the network through the Real-Time Mach microkernel.

3.2 Solaris and Real-Time Mach Implementations of ORCHESTRA

This subsection describes the implementation of the ORCHESTRA fault injection tool on Solaris and Real-Time Mach for testing distributed applications and protocols that use sockets for communication. The tool provides a fault injection layer that sits between the socket interface and the system, allowing fault injection into packets that are sent and received on sockets. The fault injection engine used by the tool is the *PFICore*, which is provided as a library, called `libpficore.a`. The socket portion of the tool is provided as a library called `libpfisock.a`, and provides the same interface as the system socket layer, e.g. `socket()`, `bind()`, `connect()`, `send()`, `recv()`. For sending and receiving of messages, ORCHESTRA uses the system provided sockets. Because ORCHESTRA presents a socket interface of its own and sits on top of the system socket interface, there was no need for access to source code of the socket libraries. Furthermore, no kernel modification was necessary for the implementation of this tool.

Building the ORCHESTRA socket layer consisted of writing the routines which comprise the socket interface (e.g. `socket()`, `bind()`, `connect()`, `send()`, `recv()`), and providing support for polling lower layers with a thread. In addition, event queues were provided so that actions such as delaying messages can be performed.

An important feature of the ORCHESTRA socket tool is that the user is not required to modify any of their source code in order to use the tool. Since the tool is provided as two libraries (`libpficore.a` and `libpfisock.a`), all that is necessary is that the user re-link their application with the new libraries. In order for the user not to get the system socket routines instead of the ORCHESTRA socket routines, the names of the routines exported by the system must be modified. We provide a library modification tool for this purpose. On Real-Time Mach, it modifies the standard C library and the user level socket library to export routines named `pfisend()` and

`pfirecv()` instead of `send()` and `recv()`[7]. The ORCHESTRA socket library, `libpfisock.a`, exports `socket()`, `bind()`, `connect()`, `send()`, `recv()`, etc. For building an application with ORCHESTRA sockets, the user simply re-links with `libpfisock.a`, `libpficore.a`, and the modified system library (either `libc.a` or `libsockets.a`).

Solaris has several libraries that use the socket interface, and using the library modification tool to modify all libraries was difficult and cumbersome. For this reason, we do not require modification of the user's source code on Solaris, but do require that the user modify their object (.o) files before re-linking their application. `libpfisock.a` exports a modified socket interface, such as `pfisocket()`, `pfibind()`, etc. A tool is provided for modification of the user's object files (in fact is is the same tool as the library modification tool). The tool changes all instances of system socket calls in the object file to ORCHESTRA socket calls (i.e. `send()` is changed to `pfisend()`). After modifying their object files, the user simply re-links the application with `libpficore.a` and `libpfisock.a`. It should be noted that object file modification can be incorporated into the user's build procedure quite easily by defining a new default rule for compiling source files into object files in the `Makefile`.

3.3 Exploitation of Real-Time Mach Features

To deal with the timing perturbations of a fault injection mechanism on a target real-time protocol, one has to quantify the intrusiveness precisely and use certain operating system features to compensate for them. This is clearly a system-dependent solution which must rely on appropriate support from a real-time operating system. The Real-Time Mach microkernel supports an abstraction called *processor capacity reserve* [27] which allows application threads to specify their CPU requirements in terms of their timing requirements. If the request is admitted by the kernel, the task is guaranteed to receive the requested CPU allocation. The kernel allows multiple threads to be bound to the same reserve. However, it ensures that application threads bound to a reserve specification cannot disturb the timing of other time-critical applications. When an application exhausts its prescribed CPU allocation, the kernel suspends it or executes it at a lower priority. Since the kernel monitors the execution of threads to enforce reserves, system calls are available to provide feedback to an application about its CPU usage. Furthermore, reserve parameters can be dynamically adjusted subject to the admission policy of the kernel. Hence, timing behavior of application reserves can be monitored and changed dynamically.

In the implementation of ORCHESTRA on Real-Time Mach, the *processor capacity reserve* and *scheduler feedback* facilities in the operating system are utilized by the *PFICore* to quantify and to compensate for the intrusiveness of the fault injection experiments on the target real-time protocol. First, user level sockets have been used along with processor reserves to achieve predictable protocol processing. Since most of the intrusiveness of the proposed fault injection and probing technique will manifest itself as additional communication overhead, the *capacity reservation* facility in Real-Time Mach is an effective mechanism for compensating for this overhead by allocating extra CPU resources to schedule communication activities. In particular, protocol processing is made predictable by binding the protocol thread to a processor reserve so that sufficient time is set aside for communication. Second, the microkernel structure of Real-Time Mach allows the addition of the *PFICore* outside the microkernel either as a separate process or as part of a library that intercepts messages to/from the target protocol. In the case of the ORCHESTRA socket-based implementation, the *PFICore* is part of a library. The overhead associated with the fault injection

[7] Of course, all other socket calls such as `socket()`, `bind()`, `connect()`, etc. are also renamed. In addition, these libraries are not installed over the system libraries, but instead are installed wherever the ORCHESTRA libraries are installed.

layer can be measured more precisely since it is outside the microkernel. Furthermore, the capacity reservation facility in Real-Time Mach can be used to provide feedback on the computational resources consumed by the fault injection layer.

Finally, having the send/receive path linked as libraries inside the application (in the case of the user level socket library) means that protocol operations take place at the priority of the process at the user-level, alleviating problems with priority inversion due to FIFO scheduling within the UX server. Similarly, by running the *PFICore* as a user-level thread with the same priority as the target application, we ensure that the fault injection protocol processing routines do not cause priority inversion for the target application threads.

4 Experimental Results

Using the socket based ORCHESTRA tool described in previous sections, an experimental evaluation of several protocol implementations was performed. These experiments were conducted in order to demonstrate the range of capabilities of the ORCHESTRA fault injection both for asynchronous and synchronous protocols. This section reports on fault injection experiments on an asynchronous group membership service developed on the Solaris operating system, and on RT-Phone, a real-time audio-conferencing application on Real-Time Mach.

4.1 Fault Injection of an Asynchronous Group Membership Protocol

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* ([19, 22, 23, 33]) using the ORCHESTRA fault injection tool. 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, 28, 31]). A detailed exposition of this problem is beyond the scope of this presentation.

Informally, the strong group membership protocol, as described in ([22]), ensures that membership changes are seen in the same order by all non-faulty 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[8]. The leader sends a `MEMBERSHIP_CHANGE` message when a new group is being formed. A processor, upon receiving this message removes itself from its old group (if the message is from a valid leader). 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.

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

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 Unix and socket-level programming on TCP/IP. Furthermore, as part of the course project, they performed several extensive tests by instrumenting the 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) using the ORCHESTRA tool, we simply ran a library modification tool on the object files of the GMD and linked with the ORCHESTRA libraries. The library modification tool is used to change all the socket related calls in an object file to match the interface exported by the PFI libraries. The change is shown in Figure 5 and can be easily automated by having the default `make` rule run the library modification tool on object files. The experiments and results follow. Five Sun workstations running Solaris were used in these experiments. They will be referred to as Sun1-5 in the remainder of this subsection.

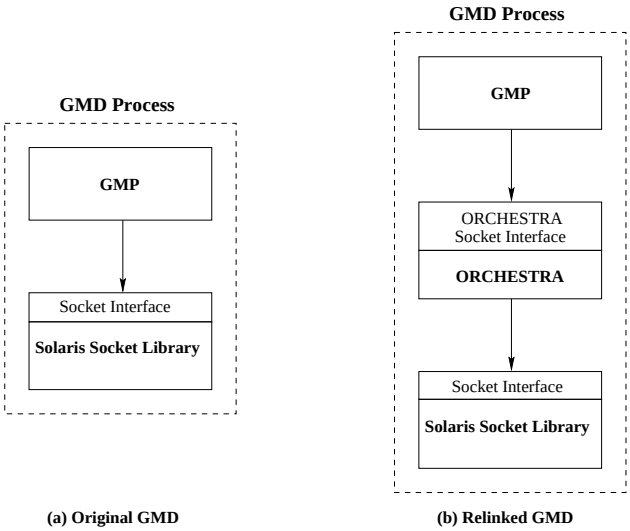

Figure 5: Protocol Layers in Group Membership Daemon: (a) GMD linked with the socket library in Solaris, (b) GMD linked with the ORCHESTRA libraries.

Experiment: Packet interruption

The first set of tests involved three machines and various types of packet interruption. The 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.

An error occurred 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 correctly 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 GMD did not find this error. The reason was that had never dropped heartbeats to the local machine. 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 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[9]. When it was un-suspended, it's timers had expired and the same bugs were observed.

Because 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 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. The machine would remain in the group until it began dropping heartbeats again, at which point the cycle would repeat, and the "faulty" machine would be kicked out of the group again.

A similar experiment was performed in which heartbeats were delayed by a few seconds. The result was exactly the same as in the dropped heartbeat experiment, 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 if the heartbeats were dropped.

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 (Sun1). Expected behavior was that Sun1 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 Sun1 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 Sun1 was dropped by the fault injector on the group leader, and the group leader did not send a `COMMIT` to Sun1. The two original machines formed a group with only themselves in it, and Sun1 stayed in a transitional state. Some time later, Sun1's `MEMBERSHIP_CHANGE` timer expired and it

[9]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.

sent out PROCLAIM messages to the others and the whole process repeated. Sun1 was never admitted to a group, which was the expected behavior.

In a variation on the previous test, the receive filter script of Sun1 was configured to drop incoming COMMIT packets. The expectation was that a group would be formed containing all machines, but Sun1 would not accept the view of the group (because it would not see the COMMIT). Since Sun1 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 Sun1 started running, it sent PROCLAIM messages and received a PROCLAIM from the group leader in response. Sun1 then sent a JOIN message to the leader. The leader sent out MEMBERSHIP_CHANGE messages to all machines and all responded with ACKs. The leader then sent COMMITs to everybody. Sun1 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 Sun1, the leader declared Sun1 dead and formed a new group which did not contain Sun1. Again, some time later, Sun1's MEMBERSHIP_CHANGE timer expired and it sent out PROCLAIM messages to the others and the process repeated.

Experiment: Network partitions

The group membership protocol is designed to handle network partitions. If a partition occurs, the result should be that separate but non-overlapping groups are formed. In order to test whether this was the case, 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 Sun{1-5}. In the second state, Sun{1-3} could only send messages to each other, and Sun{4,5} were similarly isolated.

When the machines began dropping messages, two active but disjoint groups were formed. One consisted of Sun{1-3}, and the other contained Sun{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 short time later, the machines entered the second state again and the process repeated.

In another test, the group 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 group leader in case of leader failure. There were two courses of action, but the result was the same for both. In the end, the crown prince had formed a singleton group by itself, and everyone else formed 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 except 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 except 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.

Experiment: 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 Sun1 was configured to drop PROCLAIMs 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 (Sun1).

The GMDs on the two machines were started and allowed to form a group. Sun1 was then started, and sent PROCLAIMs 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 (Sun1) 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.

Experiment: 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 Sun1 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 messages.

To begin the test, Sun1 and the group leader were started and formed an initial group. When a third machine was started later, Sun1 received a second MEMBERSHIP_CHANGE and went into a state in which incoming heartbeat and COMMIT messages were dropped. Soon after, Sun1 was still in a transitional state, when no timers (except for the MEMBERSHIP_CHANGE timer) were supposed to be set. However, Sun1 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.

4.2 Fault Injection Experiments on RT-Phone

The objective of the experiments in this subsection was to demonstrate the capabilities of the ORCHESTRA fault injection environment by testing the functional and timing characteristics of RT-Phone, a real-time teleconferencing application on the Real-Time Mach operating system. A closely related objective was to illustrate that by measuring the intrusiveness of the fault injection layer, one can compensate for timing perturbation with significant accuracy.

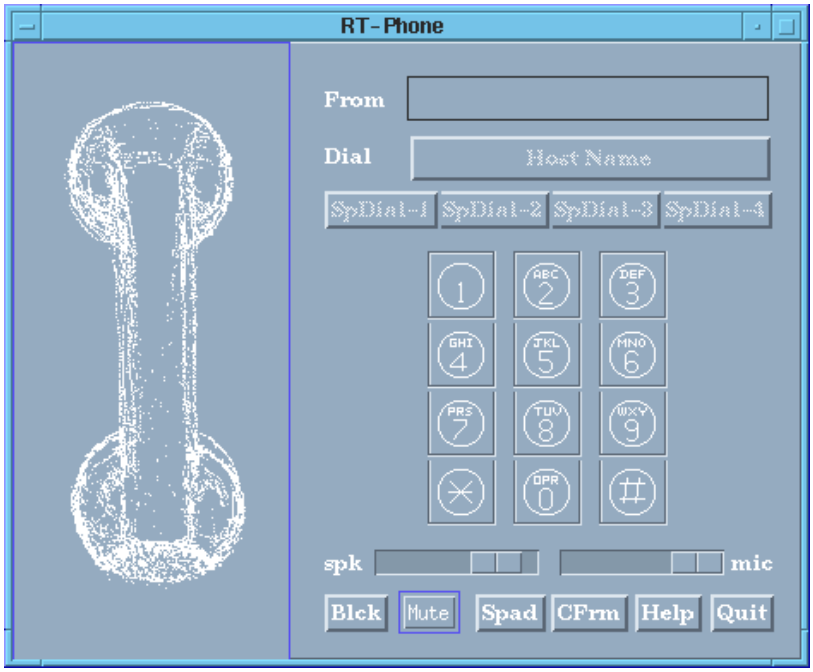

Figure 6: RT-Phone Application Window

RT-Phone is a distributed teleconferencing application with a telephone-pad-like Motif-based graphical interface [24]. A caller and a callee can establish a 2-way audio connection across the network using the graphical user interface shown in Figure 6. Various control buttons are also available to set the volume level of the speaker and the microphone, to block incoming calls selectively, and to modify the quality of the audio streams that are transmitted across the network. After a connection is established, the sound card samples the microphone input and fills the DMA buffer; an audio server thread periodically copies the sampled audio data into a buffer shared by the communication thread which in turn transmits the packet to the other end of the connection. The receiver goes through a similar set of steps (in the reverse order) to play back the audio on the speakers. In order for the phone to be full duplex (i.e. both parties can talk and listen at the same time), there are two audio servers and two sound cards on each machine. There are also two communication threads in the RT-Phone application. The basic architecture of each audio endpoint is shown in Figure 7. The RT-Phone servers use scheduling support on Real-Time Mach to ensure an end-to-end delay guarantee of less than 100 ms under various loads. The end-to-end delay in this application is a function of the reservation period for the RT-Phone threads, i.e., the frequency of the invocation of the audio server and communication threads. When the reservation period decreases (more frequent sampling and transmission of audio data), the end-to-end delay decreases proportionally and the CPU load increases non-linearly.

Figure 8 shows the various stages in the processing of audio data as it is sent from one host to

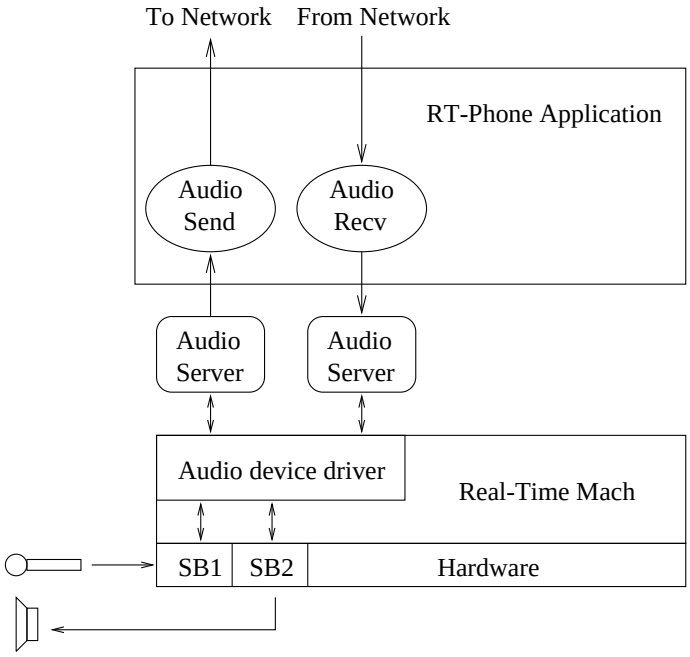

Figure 7: Architecture of RT-Phone Endpoint

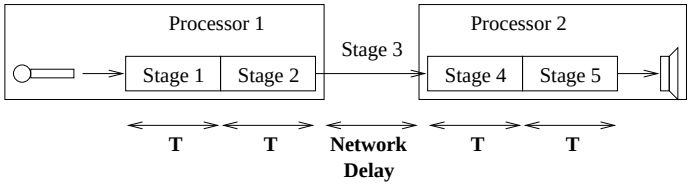

Figure 8: Audio Stream Pipeline stages

another. First, in Stage 1, the audio input from the microphone is sampled by the audio driver in the kernel. In Stage 2, the audio data is processed by the audio server, sent to the phone application, and is sent to the network. The data is transmitted from one host to another in Stage 3. In Stage 4, the data is received and sent through the audio server to the kernel sound driver. Stage 5 is the audio output stage, in which the data is played out through the speaker. The end-to-end delay of the audio stream is $3T + Network\_delay$. (Note that the delay is not $4T + Network\_delay$.[10]) A typical value for T is 16 ms for a connection with 16KHz sampling and 1 byte/sample [24].

The list of experiments conducted on RT-Phone follows:

Performance measurement of network delay component of end-to-end delay: In this experiment, ORCHESTRA was used to measure the network delay component of the end-to-end delay shown in Figure 8. The quality of the audio stream depends on an end-to-end delay constraint of 100 ms being met. Therefore, it is important that the fault injection layer does not impose a delay that will cause audio stream degradation. As mentioned above, a typical delay for each pipeline stage is 16 ms. This means that the network delay must not exceed 52 ms (of course, it should be much, much lower). We found that by using the processor capacity reserve facility, it was possible to obtain network latencies (including fault injection layer processing) under 10 ms. The maximum delays depend on the reserve period of the fault injection thread, but can generally be held below 25-30 ms, with most transmissions taking 1-2 ms.

Protocol functional testing: This set of experiments tested the protocols for connection establishment and modification of the audio stream quality. The messages that are sent from one host to another for connection establishment and modification of audio stream quality are sent via UDP. No attempt was made to reliably transmit these messages; if any messages are dropped by ORCHESTRA, problems occur. However, the problem can usually be remedied by simply clicking on the correct interface button, causing a retransmission of the message. The reason that these protocols were not made reliable is that the RT-Phone project was targeted to demonstrate end-to-end guarantees on audio quality, not reliable connection setup/teardown or stream parameter modification.

Measuring intrusiveness: In this set of experiments, several components of the overhead and cost associated with the socket based ORCHESTRA tool were measured. Measurements were taken in three areas: a) overhead of the thread management in the ORCHESTRA layer, b) cost of each Tcl script invocation, and c) overhead of different Tcl fault injection scripts. Details of these measurements are described below.

We have performed several experiments to calculate the overhead associated with the socket ORCHESTRA tool. In the first experiment, 10000 messages were sent from one machine to another. The thread associated with the ORCHESTRA tool on the client was bound to a *processor capacity reserve* so that the time accumulated against the reserve could be measured. The accumulated time was used to obtain a per message overhead. This was done first with fault injection disabled, and then with several different scripts in place when fault injection was enabled. The overhead per message with fault injection disabled averaged 390 μs, with a standard deviation of 30 μs. This overhead is primarily due to context switching between the fault injection layer thread and Mach when messages are sent, and is also due to work that the fault injection layer must perform to keep track of all of the user's sockets. With fault injection enabled, the overhead for a NULL script was 508 μs, with a standard deviation of 28 μs. This means that the overhead for calling the Tcl interpreter but not doing anything is approximately 118 μs. With two longer scripts, the time per

[10]The reason is that the data begins playing on the speaker in Stage 5, so Stage 5 is not part of the end-to-end delay.

	Overhead measured using RT-Mach capacity reserve	Overhead measured via code instrumentation
Thread overhead	390 μs	N/A
Tcl interpreter w/ NULL Script	508 (118*) μs	115 μs
Tcl interpreter w/ Script 1	616 (226) μs	225 μs
Tcl interpreter w/ Script 2	711 (321) μs	325 μs

*The higher number shows the total overhead of thread and Tcl interpreter. The numbers in parentheses show the overhead of the Tcl interpreter alone.

Table 1: Intrusiveness Results

message rose to 616 and 711 μs respectively, meaning that the script overheads were 226 and 321 μs. Standard deviations on these measurements were 35 and 36 μs respectively. The fact that different scripts have differences in execution times on the order of hundreds of microseconds is due mainly to the fact that Tcl is interpreted. Work presented in [32] addresses this by providing a version of Tcl that accepts compiled scripts as input. Speedups of 8-12 times over the interpreted case were presented in this paper, which would bring our measurement differences down onto the order of tens of microseconds.

The overhead of fault injection scripts measured using processor capacity reserve was validated by performing another experiment. In this experiment, the Mach real-time clock was sampled both before and after calling the Tcl interpreter on different send filter scripts. The real-time clock has a resolution of 1 μs. The two times were then subtracted to obtain a per script invocation overhead. For the NULL script and the other two scripts, the time per script invocation matched the average script overhead per message that was obtained by averaging over 10000 messages. That is to say, the NULL script time was 115 μs, and the other scripts were 225 and 325 μs respectively. Again, the standard deviation was about 35 μs. These performance results are summarized in Table 1.

5 Graphical User Interface for Script Generation

As mentioned in section 2.3, scripts are executed by the *PFI* layer to orchestrate the system into particular states and to inject various types of faults into the system. In order to make script specification as quick and easy as possible, we developed a graphical script editor to generate test scripts. A fault injection script can be viewed as a state machine which operates on a message. When a new message enters the fault injection layer, a handle to the current message is set up in the interpreter, and the state machine (Tcl script) is run through the interpreter. The message starts in an initial state and moves through the state machine state by state. In each state some type of action is performed. The next state the message moves to is based on information contained in the state machine, such as message type, number of messages seen in a sequence, or recent message history. The message exits the state machine when the script executes a `return` instruction. At this point, the fault injection layer checks to see if the handle to the current message still exists. If it does exist, the message is passed to the next layer. If the handle no longer exists, the message was dropped or delayed and the fault injection layer simply moves on. The user generates test scripts by "drawing" the state machine representing the test in the graphical script editor. The editor then automatically generates the Tcl code for the test script. For the majority of tests, it is simpler to generate the fault injection scripts using the editor than it is writing the scripts directly in Tcl. In addition, the user does not need a complete knowledge of Tcl to create fault injection

scripts using the editor.

In the next sections we will walk through the generation of a test script, describe the features of the editor, and then briefly discuss its design.

5.1 An Example

In order to show the advantages and usage of the graphical script editor, we will discuss the test that was used to check the response of the group membership daemon (GMD) to dropped heartbeat messages, as described in the first experiment of Section 4.1. Recall that Group membership daemons periodically exchange heartbeats to keep track of who is up. In this test we want a node to oscillate between two states. In one state we would like the node to behave correctly. In the other state, however, we would like all outgoing heartbeat messages to be dropped.

We start by examining the actual Tcl code generated by the graphical script editor. (The line numbers were added for the purpose of this discussion.)

```
1  # Created by the Orchestra GUI on Thu Oct  3 14:50:13 EDT 1996
2
3  # handle initializations
4  if {[catch {set state}] == 1} {
5      global state initial_state
6      set state "Initial"
7      set initial_state "Initial"
8
9      global counter
10     set counter 0
11 }
12
13
14 # loop through the states until an exit is reached
15 while {1} {
16     if {$state=="Initial"} {
17         # code segment for Blank action Initial
18
19
20         # user script
21
22
23         # transitions to next state
24         if {[msg_type cur_msg] != "heartbeat"} {
25             set state "exit"
26         } elseif {[msg_type cur_msg] == "heartbeat"} {
27             set state "counter"
28         }
29     }
30     if {$state=="counter"} {
31         # code segment for Counter action counter
32        incr counter 1
33
```

```
34
35          # user script
36
37
38
39          # transitions to next state
40          if {![expr int($counter/10)%2]} {
41              set state "exit"
42          } elseif {[expr int($counter/10)%2]} {
43              set state "drop"
44          }
45      }
46      if {$state=="drop"} {
47          # code segment for Drop action drop
48          msg_drop cur_msg
49
50
51          # user script
52
53
54          # transitions to next state
55          if {1} {
56              set state "exit"
57          }
58      }
59      if {$state=="exit"} {
60          # code segment for Exit action exit
61
62          # transitions to next state
63          set state $initial_state
64          return
65      }
66  }
```

The script is run for each message that enters the fault injection layer. If the message is not a heartbeat, the message is simply passed to the next layer. If the message is a heartbeat, a counter is incremented. Depending on the value of the counter, the message is either dropped or passed to the next layer.

Lines 4-11 initialize the state machine. The condition for the *if* statement is only true the first time the script is run. This code sets the initial state and initializes the counter to 0.

The rest of the code is a while loop. Inside the while loop different actions are taken depending on the current state. Lines 16-29 is the code for the initial state which does not perform any actions. The transition to the next state is based on the type of the current message. If the message is a heartbeat the next state is the counter state else the next state is the exit state. Lines 30-45 is the counter state. In this state the counter is incremented and the message will transition to either the exit state or the drop state depending on the value of the counter. Lines 46-58 is the drop state

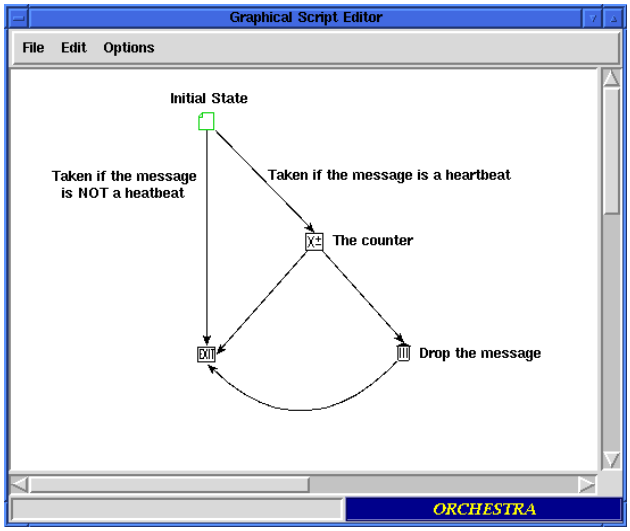

Figure 9: Example Script in Graphical Script Editor

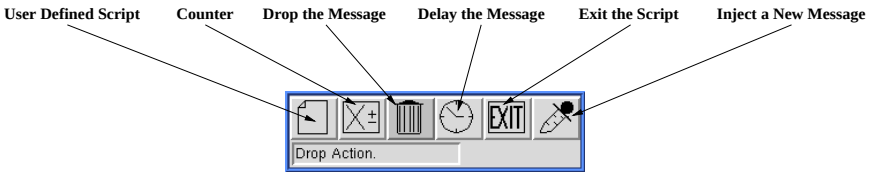

Figure 10: The predefined actions

where the current message is dropped. Lines 59-65 is the exit state. The exit state sets the state back to the initial state and exits the while loop.

Although the Tcl code for this test is relatively straightforward, it is tedious to write by hand and is especially unwieldy for the user who is not familiar with Tcl. On the other hand, the graphical representation of the script is relatively simple to specify. Furthermore, only a passing knowledge of Tcl is required to use the graphical editor. Anyone who has ever used an interactive drawing or paint program such as Xfig will be instantly familiar with the use of the graphical editor. The graphical representation of the script is shown in Figure 9.

5.2 Features of the Graphical ScriptEditor

The editor generates Tcl scripts from the graphical representation of the test's state machine that can be directly used by the *PFI* layer. The state machines are specified using the familiar point and click interface found in most interactive drawing programs. Users select tools from a floating toolbar and draw objects on a canvas by clicking and dragging the mouse. State machines are made up of two types of objects, *actions* and *transitions*. Actions are the "states" of the state machine and are connected together by transitions. Properties of an object, such as the condition for a transition, are set through dialog boxes invoked by clicking on the object with the edit tool. The editor knows about connections between actions and transitions. When objects or groups of objects are moved, these connections are maintained.

Figure 10 shows the set of predefined actions available in the current implementation of the editor. These include actions to count, delay, drop, and inject messages. In addition to the default

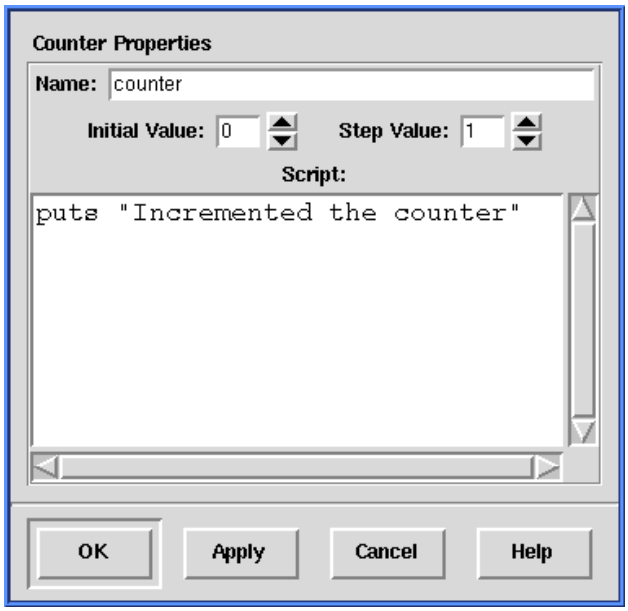

Figure 11: Counter Dialog

activities performed by each type of action, users can add their own Tcl commands by embedding a Tcl script in the dialog box for the action. Figure 11 shows a typical dialog box for the counter action.

5.3 Implementation of the Graphical Script Editor

The graphical script editor was developed using *[incr Tcl]* [26] which is an object oriented version of Tcl. Because *[incr Tcl]* is based on Tk, it was relatively easy to create the application compared to using Motif. It is also much more portable, allowing the editor to run on platforms including UNIX, Windows 95/NT, and Macintosh. This application lends itself well to object oriented programming techniques. For example, it is trivial to generate the Tcl code from the graphical representation. Each type of action contains a method for generating its corresponding Tcl code. A test script is generated by invoking this method for each action in a state machine graph.

An important benefit of the object oriented design is that it is extensible. New types of actions can easily be added by providing implementations for a few methods. Everything else can be inherited from the `Action` class. This allows designers to add their own special actions. The designers of the target layer being tested can provide the test personnel with a basic set of actions that can manipulate the distributed computation. The testers can then develop state machines from these actions and have the editor generate the scripts to test the system.

6 Related Work

Past research closely related to this work can be classified into three areas: (a) packet monitoring and filter-based approaches; (b) fault injection techniques; and (c) formal verification methods.

(a) 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 [29], a more recent work on BSD Packet Filter (BPF) which uses a register-based filter evaluator [25], and the Mach Packet Filter (MPF) [37] 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 by Comer and Lin [6] to study five TCP implementations. In addition to repeating TCP experiments similar to those reported in [6], our approach allows manipulation of messages as well as simulation of more complex failure models which are not possible with techniques that are based primarily on monitoring and gathering trace data. A detailed comparison on fault injection of the TCP protocol appears in [10].

(b) Fault injection techniques:

Various techniques based on fault-injection have been proposed to test fault-tolerance capabilities of systems. A recent survey paper [5] presents several fault injection studies, and also discusses several different tools that facilitate the application of fault injection in various environments. Hardware fault-injection (e.g. [2, 35]) and simulation approaches for injecting hardware failures (e.g. [8, 16]) have received much attention in the past. Recent efforts have focused on software fault-injection by inserting faults into system memory to emulate errors [4, 34]. Fault-injection and testing dependability of distributed systems has received more attention in recent years by several research projects [3, 13, 14, 18]. 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 [1] calculates fault coverages of a communication network server by injecting physical faults, and it tests certain properties of an atomic multicast protocol [36] in the presence of faults. Other work can be characterized as deterministic approaches to test generation [3, 13]. In [13], the evaluation of design fault coverage is based on a stochastic model. The authors of [3] propose a framework to conduct test sequences for identifying design and implementation faults in complex fault-tolerant protocols. The work reported in [18] 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.

Rather than estimating fault coverages for evaluating dependability of distributed systems, the work presented in this paper focuses on techniques for identifying violations of protocol specifications and for detecting design or implementation errors. This approach 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. Finally, one significant distinguishing characteristic of our approach is the focus on real-time as well as fault tolerance characteristics of distributed systems.

The *Delayline* [21] tool and the EFA fault injection environment reported in [14] are closest to the approach described in this paper. The *Delayline* tool 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. However, it is not intended for manipulating packets and injecting new messages. The EFA fault injector proposes inserting a fault injection layer below the fault tolerant-target protocol layer. Their work differs from ours on several key points. The first is that their fault injection layer is driven by a program which is compiled into the fault injection layer, while ours is driven by scripts which are interpreted at run time, allowing faster turn around time for new tests. The second is that their fault injection layer is fixed at the data link layer. Although our implementation of ORCHESTRA described in this paper is fixed at the socket layer, the protocol independent nature of the *PFICore* allows the *PFI* layer to be moved around the protocol stack fairly easily. In a previous paper [10] we describe an *x*-kernel layer implementation which can be placed anywhere in an *x*-kernel protocol stack.

We should note that the work on the EFA project has recently concentrated on automatic generation of fault cases to be injected. An attributed Petri net model is used to derive the fault cases by a reachability analysis. This is a very promising research avenue which has been reported in [15]. This approach differs from the work reported in this paper in that fault injection scripts in ORCHESTRA are hand-crafted by the user and this process is facilitated by a state-transition-based graphical user interface. Last, the implementation of ORCHESTRA on Real-Time Mach takes advantage of some Real-Time Mach features to attempt to compensate for intrusiveness of the fault injection layer for real-time applications. The Petri net model used in [15] does allow for specification of real-time protocols. However, to our knowledge, the EFA work does not attempt to quantify or compensate for intrusiveness of the fault injector.

(c) Formal verification methods:

While a detailed discussion of the past work on formal verification methods is beyond the scope of this document, their relationship to script-driven probing and fault injection requires further elaboration. Formal verification methods are used to prove that a system specification satisfies a given property or to demonstrate that a system specification is consistent with an implementation or a refinement. The work described in this paper attempts to check the correctness of certain execution paths for a given implementation (or an executable specification). The goal is not to demonstrate correctness for every computation of a distributed protocol, instead to focus selectively on specific system computations. Hence, this work is much closer to recent research efforts on execution simulation of real-time systems, given a formal system specification. Our primary focus, however, is on studying the actual implementations of distributed real-time protocols and on injecting faults into an execution trace from an implementation, not from a simulation. Another distinguishing characteristic of this work is the emphasis on testing the *fault-tolerance* capabilities of distributed real-time systems. In summary, we believe that the proposed technique for probing and fault injection of real-time protocols is complementary to the ongoing research on formal methods.

7 Concluding Remarks

This paper presented the software architecture of ORCHESTRA, a fault injection environment for testing distributed dependable systems. The paper focused on architectural features supporting portability and minimizing intrusiveness on target protocols, with explicit support for testing real-time protocols and easy specification of fault injection scripts. The paper also described in detail the implementation of the ORCHESTRA architecture on two different platforms: the Solaris and Real-Time Mach operating systems. The ORCHESTRA tool (and an earlier prototype implementa-

tion) have been used to conduct extensive experiments on several commercial and research systems including six implementations of the TCP communication protocol [10], a primary-backup replication protocol [11], a distributed group membership service, and a real-time audio-conferencing application, as reported in this paper. These experiments revealed major design/implementation errors as well as subtle violations of the protocol specifications. We were surprised by the power of a few relatively simple primitives offered to a protocol developer who uses the proposed approach. We were also surprised by the discovery of very intricate timing bugs which seem almost impossible to detect without a fine deterministic control on the relative ordering and timing of messages delivery in a distributed computation.

Our insight into the applicability of this approach and its limitations has grown with each experiment. An objective of our future work is to conduct further experiments on implementations of commercial fault-tolerant and real-time protocols using the ORCHESTRA environment. Other challenges include automatic generation of ORCHESTRA scripts from a high-level specification of the protocol, and development of a methodology and a formal framework for orchestrating a distributed computation. Furthermore, until now, this work has focused mainly on data collection in order to check for correct protocol behavior. Another goal of future work in this area is to further develop data collection and post analysis facilities.

Acknowledgment

We are very grateful for the insightful comments of the reviewers of the earlier conference version of this paper. In addition, we owe many thanks to Ragunathan Rajkumar and Chen Lee of Carnegie Mellon University for all of their help in getting a working version of the RT-Phone application, and for their assistance on the Real-Time Mach platform. Finally, we thank the original implementors of the group membership service for providing us with a stable version of the protocol to test. We should also thank them for leaving a few bugs in the implementation for us to find.

References

[1] J. Arlat, M. Aguera, Y. Crouzet, J.-C. Fabre, E. Martins, and D. Powell. Experimental evaluation of the fault tolerance of an atomic multicast system. *IEEE Trans. Reliability*, 39(4):455–467, October 1990.

[2] 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.

[3] D. Avresky, J. Arlat, J. Laprie, and Y. 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. 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.

[5] J. A. Clark and D. K. Pradhan. Fault injection: A method for validating computer-system dependability. *IEEE Computer*, pages 47–56, June 1995.

[6] D. E. Comer and J. C. Lin. Probing TCP implementations. In *Proc. Summer USENIX Conference*, June 1994.

[7] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–188, 1991.

[8] E. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behaviour. In *Proc. International Symposium on Fault-Tolerant Computing*, pages 236–243. IEEE, 1990.

[9] S. Dawson and F. Jahanian. Deterministic Fault Injection of Distributed Systems. In K. Birman, F. Mattern, and A. Shiper, editors, *Lecture Notes in Computer Science # 938: Theory and Practice in Distributed Systems*, pages 178–196. Springer-Verlag, September 1994.

[10] S. Dawson and F. Jahanian. Probing and Fault Injection of Protocol Implementations. *Proc. Int. Conf. on Distributed Computer Systems*, pages 351–359, May 1995.

[11] S. Dawson, F. Jahanian, and T. Mitton. A Software Fault-Injection Tool on Real-Time Mach. In *IEEE Real-Time Systems Symposium*, Pisa, Italy, December 1995.

[12] S. Dawson, F. Jahanian, and T. Mitton. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *International Symposium on Fault-Tolerant Computing*, pages 404–414, Sendai, Japan, June 1996.

[13] 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.

[14] K. Echtle and M. 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.

[15] K. Echtle and M. Leu. Test of fault tolerant distributed systems by fault injection. *IEEE Fault Tolerant Parallel and Distributed Systems*, June 1995.

[16] K. Goswami and R. Iyer. Simulation of software behaviour under hardware faults. In *Proc. International Symposium on Fault-Tolerant Computing*, pages 218–227. IEEE, 1993.

[17] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*. Addison Wesley, 1993. Second Edition.

[18] S. Han, H. A. Rosenberg, and K. G. Shin. DOCTOR: An integrateD SOftware fault injeCTOn environment. Technical report, University of Michigan, 1993.

[19] M. A. Hiltunen and R. D. Schlichting. Understanding membership. Technical Report TR 95-07, University of Arizona, Tuscon, Arizona, June 1995.

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

[21] D. B. Ingham and G. D. Parrington. Delayline: A Wide-Area Network Emulation Tool. *Computing Systems*, 7(3):313–332, Summer 1994.

[22] F. Jahanian, R. Rajkumar, and S. 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.

[23] H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[24] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor reservation and dynamic qos in real-time mach. In *Proceedings of Multimedia Japan 96*, April 1996.

[25] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX Conference*, pages 259–269, January 1993.

[26] M. J. McLennan. The new [incr Tcl]: Objects, mega-widgets, namespaces and more. In *USENIX Third Annual Tcl/Tk Workshop*, pages 151–159, Toronto, Canada, July 1995.

[27] C. W. Mercer, J. Zelenka, and R. Rajkumar. On Predictable Operating System Protocol Processing. Technical Report CMU-CS-94-165, Carnegie Mellon University, May 1994.

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

[29] 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.

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

[31] 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.

[32] A. Sah and J. Blow. A compiler for the Tcl language. In *Proceedings of the Tcl'93 Workshop*, Berkeley, California, June 1993.

[33] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proc. Int'l Symp. on Fault-Tolerant Computing*, pages 534–543, June 1993.

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

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

[36] P. Verissimo, L. Rodrigues, and M. Batista. Amp: A highly parallel atomic multicast protocol. In *Proc. of ACM SIGCOMM*, pages 83–93, Austin, TX, September 1990.

[37] M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Efficient packet demultiplexing for multiple endpoints and large messages. In *Winter USENIX Conference*, January 1994. Second Edition.