Experimental Evaluation of Failure-Detection Schemes in Real-time Communication Networks *

Seungjae Han and Kang G. Shin
Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, MI 48109–2122.
email: {sjhan, kgshin}@eecs.umich.edu

Abstract

An effective failure-detection scheme is essential for reliable communication services. Most computer networks rely on behavior-based detection schemes: each node uses heartbeats to detect the failure of its neighbor nodes, and the transport protocol (like TCP) achieves reliable communication by acknowledgment/retransmission. In this paper, we experimentally evaluate the effectiveness of such behavior-based detection schemes in real-time communication. Specifically, we measure and analyze the coverage and latency of two failure-detection schemes — neighbor detection and end-to-end detection — through fault-injection experiments. The experimental results have shown that a significant portion of failures can be detected very quickly by the neighbor detection scheme, while the end-to-end detection scheme uncovers the remaining failures with larger detection latencies.

1 Introduction

Reliable communication is an essential service for many distributed applications, some of which require very fast recovery from failures, while others can tolerate slower failure recovery. For example, telecommunication services are provided with a very high availability goal (2 hours downtime in 40 years) and require fast failure recovery so that humans may hardly notice the service disruption caused by the failure. Effective failure detection with high coverage and low latency is a key in meeting such stringent reliability requirements. Telecommunication networks employ an expensive failure-detection technique using hardware duplication/comparison to detect switching-node failures [1]. However, such computer network applications as electronic mail, file transfer, or remote file services do not mandate fast failure recovery, but require reliable (correct) delivery of all messages even if it takes a long time. *Behavior-based* failure-detection schemes without hardware support may suffice for these applications. A network node (i.e., gateway) uses heartbeats to detect the failure of its neighbor nodes, and upon detection of a failure, the node updates its routing table to have the traffic detour around the faulty component. Each message is acknowledged hop-by-hop or at the end-to-end level, and unacknowledged messages are retransmitted.

Recently, computer networks are beginning to carry real-time messages for such applications as multimedia and distributed real-time control. Real-time communication requires different reliability goals as compared to non-real-time (or best-effort[1]) communication. Since the content of a real-time message is meaningful only when it is delivered in time, retransmitted messages due to their loss or corruption may be of little use. To mask the effects of failures, multiple copies of a message should be sent simultaneously via disjoint paths. This failure masking technique has an advantage that failures are handled without service disruption. However, it may be too expensive for applications with high volume traffic like video, and moreover, many real-time applications do not require such strict reliability as 'no message loss at all'. For example, loss of a couple of frames in video/voice data

*The work reported in this paper was supported in part by the Advanced Research Projects Agency, monitored by the US Airforce Rome Laboratory under Grant F30602-95-1-0044, the National Science Foundation under Grant MIP-9203895 and the Office of Naval Research under Grant N00014-94-1-0229. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

[1] The usual non-real-time datagram service is often called 'best-effort delivery', implying that the network attempts to deliver messages as quickly as possible by using the available resources.

streams is acceptable, and temporary message losses are also tolerable in many real-time control applications because of the 'system inertia' characterized by the control system deadline [2]. In such applications, it would be more attractive to detect and recover from persistent failures.

In this paper, we focus on the issue of failure detection, an important step of failure recovery. We experimentally evaluate the effectiveness of two behavior-based detection schemes: *neighbor detection* and *end-to-end detection*. Transient and intermittent faults are injected into the communication subsystem, using a software-implemented fault injector. In particular, we measure and analyze the coverage and latency of the detection schemes.

Before delving into the details of failure detection, we need to discuss some general characteristics of real-time communication. Real-time communication service is fundamentally different from best-effort service in that individual guarantees on such QoS (Quality of Service) as delay or throughput for each connection are the key requirement. Typically, real-time communication schemes in multi-hop packet-switched networks are built based on three principles: *QoS-contracted*, *connection-oriented*, and *reservation-based*. A contract between an application and the network should be established before actual data transfer. The application/client must first specify its input traffic behavior and required QoS. Then, the network service provider computes the resource needs (link and CPU bandwidths, buffer space) from this information, selects a path, and reserves necessary resources along the fixed path. If there are not enough resources to meet the QoS requirement, the client's request is rejected. This uni-directional virtual circuit is called a *real-time channel*. Messages of a real-time channel are delivered in the order they were generated, but the delivery of a message is not guaranteed when it is corrupted, delayed, or lost due to failures.

The rest of the paper is organized as follows. Section 2 discusses the failure-detection schemes under evaluation. Section 3 presents an overview of the fault injector used for experimental evaluation. Section 4 describes the experimental setup. Section 5 deals with the analysis of experimental results, and Section 6 concludes the paper.

2 Channel Failure Detection

Reliable transport protocols guarantee the eventual (loss-free) delivery of messages between two end-points. Therefore, the "grain" of failure detection is a message. Message loss is typically detected using the 'positive acknowledgment' method, in which the receiver informs the sender of the reception of each message (or a group of messages), so that the sender can detect delivery failures. 'Negative acknowledgment' is an alternative, in which the receiver detects message losses and requests the retransmission of missed messages to the sender or other servers.

Here we are interested in detecting failures of a grain different from the above-mentioned protocols, i.e., *channel failures*. A real-time channel is said to have 'failed', if the rate of correct[2] message delivery within a certain time interval is below a threshold specified by the application. In this section, we present two behavior-based failure-detection schemes to uncover channel failures. These schemes do not require any special hardware support so that they can be used in *any* network.

2.1 Neighbor Detection Method

To detect node crash/hang failures or permanent link failures, adjacent nodes periodically exchange node heartbeats ("I am alive"). If a node does not receive heartbeats from one of its neighbors for a certain period, it considers the silent neighbor failed and stops sending heartbeats to that node. Heartbeats do not carry any useful information, and regular messages can be used as heartbeats. Explicit heartbeats are sent only if there are no regular messages for a pre-specified period.

The heartbeat scheme is specified by three parameters: heartbeat-generation interval t_g, heartbeat-check interval t_c, and a tolerable number, m, of heartbeat misses. Assume that the same value is used for both t_c and t_g. Then, unless the clock and/or scheduling skews $\ll t_c$, the heartbeat checker should wait one more t_c after the number of heartbeat misses reached m, because of the variance in the actual heartbeat-generation/check timing between two adjacent nodes. Thus, it takes $(m+2)t_c$ for a node to detect the failure of one of its neighbors.

When two nodes are joined by dual simplex links, a node cannot tell the difference between the failure of its neighbor nodes and that of the corresponding links by exchanging local heartbeats only. Instead of relying on sophisticated diagnosis, we treat all channels running through the suspected link as faulty. So, when the incoming link from a node fails, the channels on the outgoing link to the node will be considered failed, even if they were healthy. This is reasonable because a channel cannot maintain dependable service if the health of the channel cannot be monitored.

[2] In terms of both content and timing.

2.2 End-to-End Detection Method

The end-to-end detection method involves both end nodes of a real-time channel. It works as follows. The source node, whenever necessary, injects a "channel heartbeat" into the channel message stream. A channel heartbeat is a sort of real-time message, and the intermediate nodes on a channel do not discriminate channel heartbeats from data messages. Each channel heartbeat contains the sequence number of the latest data message. In this way, the destination node can monitor the number of data messages lost. If the message-loss rate of the channel exceeds the threshold γ specified by the application, the destination node declares that the channel has failed.

For each channel, the source node manages a heartbeat-generation timer which is incremented by clock interrupts. The heartbeat-generation timer is reset every time a message (data or heartbeat) is transmitted over the channel. Only when the value of the heartbeat-generation timer reaches the maximum heartbeat interval h_{max}, an explicit channel heartbeat is generated. Therefore, when h_{max} is set to a sufficiently large value relative to the data message interval, explicit heartbeats will seldom be generated due to the (near) periodic nature of real-time messages, thus making the overhead of channel heartbeats small. The h_{max} of a real-time channel should be chosen to fit the channel's traffic characteristics. It should be larger than the minimum message interval of the channel which is specified in the channel's QoS contract; otherwise, the resources reserved for the channel will not be sufficient to carry both data and heartbeat messages of the channel. The smallest possible failure-detection latency of a channel is therefore determined by the channel's traffic characteristics.

2.3 Experiment Goals

The end-to-end detection scheme will uncover *all* channel failures, so the main concern is its detection latency. Under this scheme a large minimum message interval of a channel will result in a long detection latency. Even when the message interval is small, if the channel's γ value is large, a long detection latency will result. The neighbor detection scheme has a much weaker dependency on the underlying traffic than the end-to-end scheme, because it monitors the behavior of a neighbor node, rather than that of a real-time channel. Therefore, one can achieve smaller detection latencies by the neighbor scheme than by the end-to-end scheme. However, in the neighbor scheme, there is a possibility that a node is not operating correctly in terms of message processing, but still generates node heartbeats or propagates part of regular messages. In such a case, the neighbor scheme will result in less than perfect detection coverage. Even though the faulty node becomes silent eventually, the detection latency may be larger than that of the end-to-end scheme. In case of multiple channels going through a node, another problem arises: have all channels on the node or part of them failed, when the node stops generating heartbeats? In this paper, we would like to answer the above questions by experimentally evaluating the coverage and latency of the neighbor detection scheme.

3 DOCTOR: Integrated Fault Injection Tool Set

Fault injection has long been viewed as a useful means of testing/evaluating fault-tolerant systems. Numerous hardware-implemented fault injectors (HFIs) [3–5] have been developed and used for various experiments. However, as the complexity of contemporary computer increases as a result of using highly-integrated VLSI chips, it is becoming more difficult, or nearly impossible, to evaluate dependability with HFIs alone. On the other hand, software-implemented fault injectors (SFIs) [6–10] have been proposed as less expensive and more controllable alternatives. Although SFI techniques such as overwriting memory or register contents are becoming popular, they still face many difficulties. For example, the intrusion into normal execution by fault injection should be minimized and isolated to obtain accurate measurements, especially in real-time systems. In order to remedy some of these difficulties, we have developed an integrated fault-injection environment called DOCTOR[11], particularly for distributed real-time systems.

DOCTOR provides a complete set of tools for automated fault-injection experiments. One of the core parts of DOCTOR is the fault injector which consists of three modules: experiment generation module (EGM), experiment control module (ECM), and fault injection agent (FIA). EGM is responsible for preparing experiments, such as generating a set of fault instances to be injected, while ECM is the run-time experiment manager. FIA performs actual fault injections under ECM's control. Another core part is the data monitor (HMON) which collects the experimental data at run time. Data analysis module (DAM) analyzes the collected data off-line after completing the experiment.

One distinct structural feature of DOCTOR is the separation of software components of the host computer from those of the target system. Thus, EGM and ECM run on the host computer while FIA runs on the target system. It has the advantage of reducing the run-time interference with the target system caused by fault injection, because only essential components are executed on the target system. It also increases the portability of DOCTOR, since the highly system-dependent part is isolated from the rest.

Evaluation of a fault-tolerance mechanism needs a systematic fault-injection plan, and thus, the capability of injecting a proper fault instance into a proper location at a proper time is essential. DOCTOR supports the injection of a variety of faults and errors, ranging from low-level faults such as memory or processor faults to high-level errors such as communication errors. Three temporal properties — transient, intermittent, and permanent — are supported for the following fault types.

Memory fault: Contents of the cache or main memory are corrupted. The fault injection target can be either explicitly specified by the user, or chosen randomly from the address space using the symbol table and object file information. For better controllability, DOCTOR allows faults to be injected only into a certain memory section of a particular target task (or executable object image), such as text area, global variable area, or stack/heap area.

Processor fault: CPU faults are emulated by corrupting the contents of CPU registers. The timing of injecting register errors can be randomly selected or can be controlled to be the time when a specific task or instruction is executed. Bus faults can be emulated as well, by corrupting the content of an instruction just before its execution and restoring it after the instruction cycle. Similarly, various types of faults in the processing unit can be emulated, such as ALU errors and instruction fetching unit errors.

Communication error: Errors in communication links can be emulated using a special fault-injection layer inside the protocol stack. The user can define the intended fault behavior, while some pre-defined fault types are supported including message loss, corruption, delay, and duplication. Fault injection timing/duration can be specified by either time or message history (e.g., dropping several consecutive messages of a certain type).

In addition to the capability of injecting various faults, effective data collection is essential in fault-injection experiments. Typically, software monitors suffer from poor timing-resolution and cause significant performance intrusion. On the other hand, signal-level hardware monitors lack the ability of capturing a wide range of software-level events, such as OS events or application-specific events. To overcome the shortcomings of these conventional approaches to data collection, we have developed a hybrid-style data monitor, called HMON, which mainly observes software events.

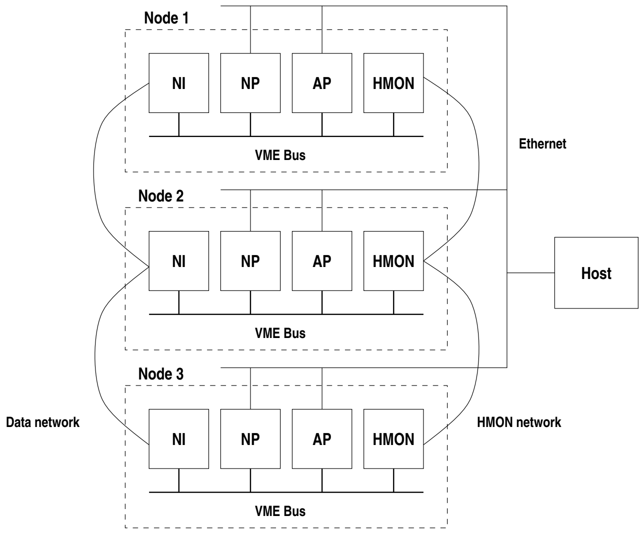

Figure 1: Configuration of the experimental platform

The data monitoring process consists of three steps: *event probing*, *time-stamping*, and *data logging*. In our approach, event probing is done by software to preserve flexibility, the advantage of software monitors — special program codes are inserted into the objects to be monitored. Probing targets may include system call invocations, context switches, interrupts, fault injections, fault detections, etc. In contrast, time-stamping and data logging are done by custom hardware to maximize timing accuracy and to minimize intrusion. For data collection in distributed systems, multiple HMONs can be used. Each network node will be equipped with its own HMON, and these HMONs are connected via a dedicated HMON network for generating synchronized time-stamps.

4 Experimental Setup

In this section, we describe the hardware/software configuration of the experimental platform, the real-time communication subsystem, and the failure detection mechanisms implemented on the platform. We also outline our fault-injection experiments.

4.1 Experimental Platform

As shown in Figure 1, the experimental platform consists of three nodes, Nodes 1–3. Each node is a VME bus-based multiprocessor system with Motorola 68040 microprocessors. In each node, a CPU board (labeled as NP) is dedicated to communication processing, while a separate CPU board (labeled as AP) is used for application processing. As a communication fabric between nodes, a network interface board (NI) featuring the 'ANSI Fiber Channel 3.0 standard' is equipped with each node. In addition, a HMON board is added to each node for data collection. Node 1 and Node 2 are connected by two simplex network links (i.e., optical fibers), one for each direction. The same type of connection exists between Node 2 and Node 3. A SUN workstation serves as the host machine, and is connected to nodes through an Ethernet. Nodes are not equipped with disks, and application/system software is downloaded from the host machine.

An extended version of the pSOS^{+m} real-time OS [12] is used for AP's system software. The AP-side software is not important in our experiment, since APs run very simple applications which request message delivery to the associated NP, and retrieve messages received by the NP. NP employs a derivative of *x*-kernel 3.1 [13] as a system executive and a substrate for building the protocol stack. Since NPs do not run user tasks, we disabled the virtual address management of *x*-kernel. Thus, all tasks in NP are executed within a single (kernel) address space. Memory protection of *x*-kernel was also disabled to minimize the overhead.

4.2 Communication Subsystem

Each NP features the real-time communication scheme described in [14]. The protocol stack includes protocols for application program interface (API), network management (NM), remote procedure call (RPC), transport-level fragmentation (FRAG), unreliable datagram service (HNET), and the device driver for network interface boards (DD). The API protocol exports routines that applications can use to set up/tear down real-time channels and perform data transfer on the channels. The RPC protocol is used by the NM protocol for transmitting channel establishment/teardown messages. The HNET protocol covers the function of the network layer and part of the datalink layer. The run-time message scheduler is implemented in it.

The NP system software, *x*-kernel, uses a non-preemptive scheduling policy with 32 priority levels for task scheduling, and its protocol processing is based on the *process-per-message* model. Whenever a message arrives at a network device or needs to be transmitted into the network, a process (or thread) is created to shepherd the message through the protocol stack; this eliminates extraneous context switches encountered in the usual process-per-protocol model. Once a protocol thread is scheduled, it runs without preemption until completion of protocol processing. While the process-per-message model suffices for best-effort messages, it introduces complexity for maintaining QoS guarantees and performing traffic policing. For this reason, we implemented the run-time message scheduler as a special thread that is created at system startup and runs at the highest-priority level. Implementation details can be found in [15].

4.3 Failure-Detection Mechanisms

For the neighbor detection scheme, the node heartbeat generator/checker is implemented as a separate task. It is periodically executed and checks special flags, each of which is associated with a link and is set whenever a message is transmitted over the link. If the flag is not set when the heartbeat generator is invoked, a new heartbeat is generated and sent as a best-effort message. The heartbeat checking is also done in a similar way. There is an alternative implementation for heartbeat generation. Instead of running as a separate task, the heartbeat generator can run on top of the clock interrupt thread, and heartbeats are directly fed to the device driver without going through the message scheduler in the HNET protocol. However, we have not used this implementation option because of the following two drawbacks. First, execution time of the clock interrupt handler is extended, during which other interrupts are disabled. Second, even when OS task scheduler or the message scheduler hang, heartbeats will be sent out, which will lower the detection coverage.

For the end-to-end detection scheme, NM spawns a special thread for each channel. This thread is invoked using watchdog timers (or alarm functions). Thus, every time a message of a real-time channel is generated by the application, the thread associated with the channel is scheduled to be invoked after a delay of h_{max}. When a new data message is generated by the application before the timer is expired, the timer is reset to h_{max} again; hence, the scheduled thread is canceled and a new thread is scheduled to be invoked after a delay of h_{max}. When activated, the thread generates channel heartbeats for the corresponding real-time channel.

4.4 Experiment Description

The experiment controller, ECM, on the host machine communicates with FIAs in the target nodes through an Ethernet. To minimize the interference caused by fault injection, the function of the FIA in NP is minimized; the communication with ECM is done by a FIA proxy in AP, and the FIA in NP

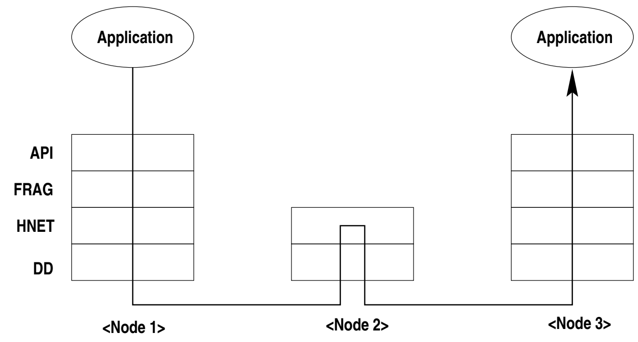

Figure 2: Real-time message passing

communicates with the FIA proxy through the VME bus. The FIA proxy is also responsible for controlling HMON and uploading the collected data by HMON.

One or two real-time channels were established from Node 1 to Node 3 through Node 2. The end-to-end delay requirement was 30 msec and the application program generated real-time messages regularly once every 50 msec without any burst. A channel failure is said to occur if no message is delivered for more than 250 msec. Since messages were generated periodically, no 'channel heartbeat' was added in between two consecutive messages of the real-time channel. The interval of 'node heartbeats' was set to 30 msec and the tolerable number of (node) heartbeat misses was set to 1. Thus, the node heartbeat generator and receiver are invoked once every 30 msec to generate or check heartbeats, and if heartbeats are not received for three consecutive checking intervals, a failure is declared (detected).

Faults were injected into the NP of the intermediate node, Node 2. As illustrated in Figure 2, only the bottom two protocols, HNET and DD, are executed for run-time message passing at Node 2.[3] Since we are interested in detecting failures of real-time channels after their establishment, we restrict the fault-injection target to these two protocols and other OS modules, particularly the task manager (TM) and the clock service (CS).

Three classes of faults were injected: memory faults, CPU register faults, and communication faults. Memory faults were injected into the text area of the target modules at randomly-selected times. The effects of memory faults in the data area can be covered largely by CPU register faults, since memory variables are typically loaded into registers. To emulate CPU faults, the values of data/address registers were corrupted. Time-driven triggering was not very effective in injecting CPU register faults in our platform. It is because message threads are created and destroyed very quickly and thus, the CPU is idle for a large portion of time, unlike usual fault-injection experiments in which application programs run continuously. To increase the fault-activation rate, a different method was used to trigger a fault-injection. When an instruction in the target address is executed, a fault is injected into the register used by the instruction. In addition, we inject faults into PC and CCR to study the effects of faults in control registers. To maximize the chance of fault activation, CCR faults are injected when conditional branch instructions are executed. We also emulate the faults in (physical) communication links. The fault-injection layer is inserted between DD and HNET. Since the results of such communication errors as message drop or message data corruption are straightforward, we inject corruption errors into the message header part.

[3] NM/RPC protocols are used for channel establishment and API/FRAG is executed only at end-nodes.

Each experiment was fully automated, so that multi-run experiments were done without human intervention. To synchronize the start and the end of each run, dummy FIAs were executed at NPs of Node 1 and 3. Each experimental run consists of 6 sequential steps:

Step 1: EGM generates a fault-injection script for the run. (Scripts for multiple runs can be generated at once.)

Step 2: ECM downloads system software (including communication subsystem) and fault-injector software to NPs and APs, and remotely boots the target system.

Step 3: When the connection between ECM and FIA proxy is ready, ECM sends the current fault-injection script to FIA proxy.

Step 4: FIA waits until applications establish real-time channels.

Step 5: After the message transmission is started, FIA injects a fault (or multiple faults). During the run, HMON collects time-stamped data such as message generation, message relay, message reception, fault injection, failure detection, and heartbeat generation/reception.

Step 6: When the pre-specified experiment duration is reached, the collected data is uploaded to ECM, and FIA proxies reset all nodes for the next run.

After each experiment, we calculated (i) the channel-failure rate, and (ii) failure-detection coverage/latency of the neighbor detection scheme. First,

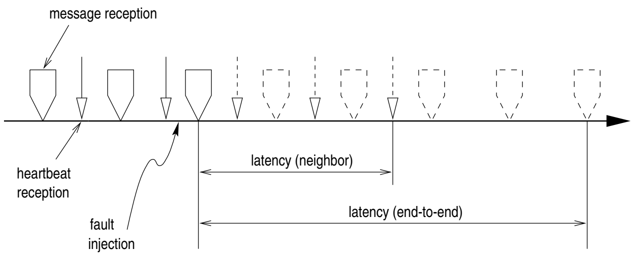

Figure 3: Failure-detection latency

each fault injection will result in one of 3 cases: *no error*, *tolerable error*, *channel failure*. The channel-failure rate is then computed as the ratio of the third case to all cases. The failure-detection coverage is the percentage of detections among the runs in which channel failures had occurred. We measured only the failure-detection coverage of the neighbor scheme, because the end-to-end scheme has always perfect coverage. When calculating the coverage of the neighbor scheme, we excluded the case when the neighbor scheme detects a failure which had already been detected by the end-to-end scheme. Finally, the failure-detection latency is computed as the duration between the time of the last message delivered correctly and the time of failure detection. Figure 3 illustrates the failure-detection latency. In our current experimental setup, the detection latency of the end-to-end scheme is always 250 msec, and the minimum possible detection latency of the neighbor scheme is 90 msec.[4] The detection latency of the neighbor scheme depends on (i) how long it takes until the fault affects the real-time message, and (ii) how long it takes until heartbeats are affected by the fault.

5 Experimental Results

In this section, we present the experimental results and analyze their implications. The data was collected from more than 10,000 experimental runs, each of which took about 65 seconds; 35 seconds for experiment setup + 30 seconds for executing the experiment.

5.1 Injection of Transient Faults

In this experiment, transient single-bit toggle faults were injected, with one real-time channel set up. According to the common practice in software-implemented fault injectors, we use the term 'transient' to mean the opposite to 'permanent'. For example, register faults are transient because the corrupted register contents can be overwritten by the subsequent instructions. A message fault corrupts the header of only one message, so it is also transient. The faults injected into memory are also transient, but, since the program text area is corrupted, they will have properties similar to permanent memory faults. The experimental results are summarized in Tables 1 and 2.

[4] One has to be careful in interpreting the detection latencies of these two detection schemes, because destination nodes detect failures in one scheme, while intermediate nodes do in the other scheme.

Channel failure (cf); tolerable error (ok); no error (no)

		HNET	DD	TM	CS	Total
Mem	runs	308	306	309	338	1261
	cf	54.2%	51.6%	49.8%	50.0%	**51.4%**
	ok	0.3%	0.3%	0.3%	0%	0.2%
	no	45.5%	48.0%	49.8%	50.0%	48.4%
Ax, Dx	runs	307	312	296	303	1218
	cf	59.6%	42.3%	56.8%	38.6%	**49.3%**
	ok	2.9%	1.3%	1.4%	1.0%	1.6%
	no	37.5%	56.4%	41.9%	60.4%	49.1%
CCR	runs	376	376	364	378	1494
	cf	40.4%	60.6%	66.8%	33.3%	**50.1%**
	ok	10.1%	11.2%	5.2%	0.3%	6.7%
	no	49.5%	28.2%	28.0%	66.4%	43.2%
PC	runs	270	287	272	278	1107
	cf	97.8%	99.3%	98.5%	97.1%	**98.2%**
	ok	1.5%	0.3%	0%	0%	0.5%
	no	0.7%	0.3%	1.5%	2.9%	1.4%
Msg	runs	302				
	cf	**43.7%**				
	ok	0.7%				
	no	55.6%				

Table 1: Fault manifestations: transient fault injection

The first observation from these experimental results is the relatively low coverage. For example, $(72.6 + 0.1)/(100 - 18.2) = 88.8$ % of failures were reported in [16] to have been detected by the system hardware detection mechanism, which was also based on a Motorola 680x0 CPU. This is much higher than the coverage observed in our experiment. This discrepancy can be explained based on the following four reasons. First, the two used different fault-injection methods. In [16], a hardware-implemented (pin-level) fault injector was used, so the fault injected at a CPU pin for two memory cycles can be manifested as several errors at the level which a software-implemented fault injector deals with. Moreover, the faults forced into control signal pins will make more pronounced impact on the target system than the data-level errors. Second, we excluded the late detections by the neighbor scheme from the coverage calculation. As shown in Table 3, the portion of late detections within the set of undetected failures ranges from 0% to 100%, depending on fault types and fault-injection targets. Third, the underlying workload (i.e., system software and ap-

Coverage (c); latency mean (l_m), latency var (l_v) in msec

		HNET	DD	TM	CS	Total
Mem	c	49.1%	35.4%	39.0%	28.4%	**38.0%**
	l_m	138.7	148.0	131.3	125.2	136.4
	l_v	24.0^2	25.5^2	21.1^2	14.3^2	23.4^2
Ax, Dx	c	50.8%	38.6%	50.0%	11.1%	**40.2%**
	l_m	144.2	138.4	138.4	135.7	140.5
	l_v	21.7^2	23.2^2	27.2^2	20.7^2	24.1^2
CCR	c	31.6%	29.4%	69.1%	0%	**37.9%**
	l_m	132.0	142.9	138.4	NA	138.4
	l_v	21.5^2	27.9^2	21.5^2	NA	23.5^2
PC	c	86.7%	87.7%	82.5%	80.0%	**84.3%**
	l_m	140.5	134.3	132.4	137.7	136.2
	l_v	20.9^2	20.7^2	19.4^2	25.5^2	21.9^2
Msg	c	**39.4%**				
	l_m	150.1				
	l_v	20.5^2				

Table 2: Detection coverage and latency: transient fault injection

	HNET	DD	TM	CS	Total
Mem	15.3%	13.7%	8.5%	2.5%	9.5%
Ax,Dx	16.7%	8.6%	26.2%	4.8%	13.6%
CCR	7.7%	9.9%	33.3 %	0%	10.5%
PC	97.1%	100%	100%	92.6%	97.1%
Msg	3.7%				

Table 3: The percentage of late failure detection among undetected failures

plication program) was different. Computationally-intensive workloads (e.g., sorting, searching, matrix multiplication, etc.) were executed in the experiments of [16]. The dependency of fault-tolerance measures on workload has been reported by several researchers, e.g., [17,18]. Fourth, our target system, like most other real-time systems, is not equipped with memory-protection capability. The experimental result in [5] indicates that memory protection can enhance detection coverage up to 15%.

One can make several other observations. The composition of fault manifestations varies greatly, depending on fault types and injection targets.[5] For example, fault injection into PC has resulted in very high channel-failure rate and detection coverage. Though some of the failures caused by PC faults have gone undetected by the neighbor scheme, most of them were eventually detected (see Table 3). In contrast, the faults injected into CCR have resulted in much lower detection coverage than those injected into PC. It is because CCR errors cause incorrect control flow, which is difficult to detect without special hardware support (e.g., a watchdog processor), while many PC errors can be detected by CPU-intrinsic fault-detection mechanisms like bus error, unaligned memory access, etc. The faults injected in the clock service routines were not detected well by the neighbor scheme. Particularly, address/data register faults and CCR faults into the clock service resulted in very low coverage, implying the need for incorporating some additional anomaly-detection features (e.g., software assertions) into the clock service routines.

Interestingly, while many of the tolerable errors were due to 'message deadline violations' and some of them were due to 'message losses', very few 'message data corruption' errors were found. This is because the message data part is saved into the memory by the device driver when the message arrives and is not copied or modified by other protocols in x-kernel. We also measured the probability of *false alarm*; no channel failure had actually occurred even if the neighbor scheme signals a failure detection. This happens when both heartbeats and real-time messages are not transmitted for longer than 90 msec, then the real-time message delivery service returns to normal in 250 msec. According to our experimental results, false alarms occur very rarely and the false-alarm probability is statistically negligible.

Now, let us consider the detection latency. In the current experimental setup, a real-time message is transmitted once every 50 msec, and a heartbeat is transmitted 30 msec after each real-time message was transmitted. If faults prevent the transmission of both real-time messages and heartbeats at the same time, the detection latency of the neighbor scheme will be either 90 msec or 120 msec. If a fault is activated within 30 msec after a real-time message was sent (thus, between a real-time message and a heartbeat), since all messages after that point will be dropped, the failure-detection latency will be the same as three consecutive heartbeat intervals, 90 msec. On the other hand, if a fault is activated between a heartbeat and a real-time message, the heartbeat will be sent and only the real-time message will not be sent, extending the detection latency by 30 msec. Thus, the average detection latency is theoretically $30/50 \times 90 + 20/50 \times 120 = 102$ msec, provided the fault-activation time is evenly distributed.

However, in reality, the nature of faults is not that simple; the measured detection latency was about 140 msec on average. There are two reasons for this. First,

[5] The channel-failure rate can be slightly increased, if the experiment duration is extended, because there could be the cases of very long fault-activation latency.

		HNET	DD	TM	CS	Total
Mem	runs	251	249	251	250	1001
	cf	85.3%	90.0%	72.9%	73.2%	**80.3%**
	ok	0.8%	0%	0%	1.2%	0.5%
	no	13.9%	10.0%	27.1%	25.6%	19.2%
Ax, Dx	runs	252	251	250	246	999
	cf	59.1%	42.6%	61.6%	37.4%	**50.3%**
	ok	4.4%	1.6%	1.2%	1.2%	2.1%
	no	36.5%	55.8%	37.2%	61.4%	47.6%
CCR	runs	247	235	245	254	981
	cf	51.8%	65.5%	68.6%	39.4%	**56.1%**
	ok	9.7%	8.1%	5.3%	0.8%	5.9%
	no	38.5%	26.4%	26.1%	59.8%	38.0%
PC	runs	224	240	227	228	919
	cf	97.8%	99.2%	96.0%	99.1%	**98.0%**
	ok	1.8%	0.8%	0.9%	0%	0.9%
	no	0.4%	0%	3.1%	0.9%	1.1%
Msg	runs	252				
	cf	**73.4%**				
	ok	0%				
	no	26.6%				

Table 4: Fault manifestations: intermittent fault injection

		HNET	DD	TM	CS	Total
Mem	c	71.0%	66.5%	76.5%	57.4%	**67.9%**
	l_m	143.3	140.8	135.1	136.3	139.2
	l_v	26.3^2	22.4^2	23.1^2	23.2^2	24.0^2
Ax, Dx	c	57.0%	40.2%	65.6%	12.0%	**47.8%**
	l_m	145.5	143.4	141.1	126.8	142.4
	l_v	23.4^2	27.4^2	25.6^2	14.9^2	24.9^2
CCR	c	56.2%	33.1%	72.0%	2.0%	**44.7%**
	l_m	142.1	148.9	142.9	164.5	144.2
	l_v	26.4^2	31.0^2	28.9^2	14.8^2	28.6^2
PC	c	86.8%	88.7%	83.9%	78.8%	**84.6%**
	l_m	143.1	134.0	135.2	137.4	137.4
	l_v	23.3^2	20.9^2	22.9^2	21.8^2	22.4^2
Msg	c	**68.1%**				
	l_m	149.1				
	l_v	19.8^2				

Table 5: Detection coverage and latency: intermittent fault injection

faults may delay or drop messages for some duration before a complete channel failure. Late real-time messages are the same as message losses from the application's point of view, while they are considered as implicit heartbeats from the heartbeat checker's perspective. Thus, delayed messages may extend the detection latency. Second, fault-propagation delay can be another reason for a long detection latency. Suppose a fault is injected between a real-time message and a heartbeat. The heartbeat may not be affected by the fault because of the fault-propagation delay, but the next real-time message can be affected by the fault. Then, the detection latency will become 120 msec, instead of 90 msec. Moreover, the fault-propagation delay can be different for real-time messages and heartbeats. In an extreme case, faults affect real-time messages quickly and affect heartbeats slowly, which will result in a very long detection latency.

5.2 Injection of Intermittent Faults

To examine the performance of the neighbor detection scheme for harsher (than transient) faults, we experimented the case of intermittent faults. Up to five single-bit toggle faults were injected, where each fault was independently selected. The experimental results are summarized in Tables 4 and 5. As expected, higher channel failure rate and detection coverage have resulted as compared to the case of transient fault injections. In particular, the detection coverage for memory faults or message faults increased by a larger margin than address/data register faults and CCR faults which seem to affect the program execution more delicately.

5.3 Effects of Faults on Multiple Channels

The same experiment as in Section 5.1 was conducted with two real-time channels set up, in order to study the effects of faults on multiple real-time channels. We measured the coverage and latency separately for each channel. Though the results were not presented due to space limitation, they were close to those shown in Tables 1 and 2.

A surprising result is that, among more than 2,000 experiments, we did not observe any case in which one channel fails but the other channel does not. This is attributed to the sharing of program codes in processing real-time messages belonging to different channels. Recall that in x-kernel, whenever necessary, a shepherd thread is spawned to process a new message, and all shepherd threads execute the same (protocol-processing) program code.[6] If the execution of a thread is faulty because of faults in the local data of the thread, only the message associated with the thread will be affected, not all messages of a channel. In contrast, if the source of the incorrect execution is in a globally-shared component like program code or system software, all messages of all channels will be affected. If a faulty thread affects the execution of other threads or the channel-specific data (i.e., a

[6]The property of program code sharing in message processing is not limited to our platform and is common in most conventional protocol implementations such as in BSD Unix.

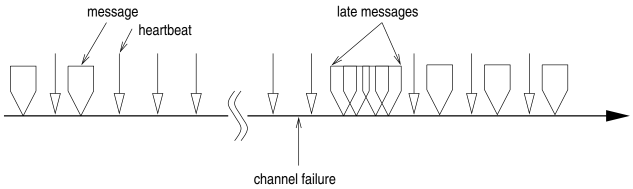

Figure 4: A typical example of undetected channel failure

link-deadline) is corrupted, only a subset of channels fail. However, our experimental results have shown this possibility to be negligible.

6 Conclusion

In this paper, we investigated the effectiveness of two failure-detection schemes — neighbor and end-to-end detection — in real-time communication networks. The neighbor scheme has been used widely in many non-real-time computer networks to monitor the health of network nodes. Our experimental results on a platform, which was designed without any particular consideration for fault-tolerance, have indicated that the coverage of the neighbor scheme is not very high and a long latency occasionally results, but in general, it detects a significant portion of failures very quickly. One main reason for this low coverage and occasional long latency is that faults do actually cause real-time messages to be delayed. A typical failure symptom which was not detected by the neighbor scheme is illustrated in Figure 4. The end-to-end scheme can detect the failures which were missed by the neighbor scheme. Despite its perfect coverage, this scheme has such weaknesses as high overhead, long latency, and the inability to locate failures. Thus, improving the coverage and latency of the neighbor scheme is a crucial step for fast failure recovery.

References

[1] W. N. Toy, "Fault-tolerant design of AT&T telephone switching system processors," in *Reliable Computer Systems: Design and Evaluation*, pp. 533–574. Digital Press, 1992.

[2] K. G. Shin and H. Kim, "Derivation and application of hard deadlines for real-time control systems," *IEEE Trans. on System, Man, and Cybernetics*, vol. 22, no. 6, pp. 1403–1413, November 1992.

[3] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems.," in *Proc. IEEE FTCS*, pp. 348–355, 1989.

[4] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proc. IEEE FTCS*, pp. 340–347, 1989.

[5] H. Madeira and J. Silva, "Experimental evaluation of the fail-silent behavior in computers without error masking," in *Proc. IEEE FTCS*, pp. 350–359, 1994.

[6] Z. Segall et al., "FIAT – fault injection based automated testing environment," in *Proc. IEEE FTCS*, pp. 102–107, 1988.

[7] R. Chillarege and N. S. Bowen, "Understanding large system failures — a fault injection experiment," in *Proc. IEEE FTCS*, pp. 356–363, June 1989.

[8] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A tool for the validation of system dependability properties," in *Proc. IEEE FTCS*, pp. 336–344. IEEE, 1992.

[9] K. Echtle and M. Leu, "The EFA fault injector for fault-tolerant distributed system testing," in *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 28–35. IEEE, 1992.

[10] W. Kao, R. Iyer, and D. Tang, "FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults," *IEEE Trans. Software Engineering*, vol. 19, no. 11, pp. 1105–1118, November 1993.

[11] S. Han, K. G. Shin, and H. Rosenberg, "DOCTOR: An integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems," in *Proc. IEEE IPDS*, pp. 204–213, 1995.

[12] L. M. Thompson, "Using pSOS$^+$ for embedded real-time computing," in *Proc. COMPCON*, pp. 282–288, 1990.

[13] L. L. Peterson, N. C. Hutchinson, S. W. O'Malley, and H. C. Rao, "The *x*-Kernel: A platform for accessing internet resources," *IEEE Computer*, vol. 23, no. 5, pp. 23–33, May 1990.

[14] D. D. Kandlur, K. G. Shin, and D. Ferrari, "Real-time communication in multi-hop networks," *IEEE Trans. Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044–1056, October 1994.

[15] A. Mehra, A. Indiresan, and K. G. Shin, "Resource management for real-time communication: Making theory meet practice," in *Proc. IEEE RTAS*, pp. 130–138, 1996.

[16] M. Rela, H. Madeira, and J. Silva, "Experimental evaluation of the fail-silent behavior in programs with consistency checks," in *Proc. IEEE FTCS*, pp. 394–403, 1996.

[17] S. Butner and R. Iyer, "A statistical study of reliability and system load at SLAC," in *Proc. IEEE FTCS*, pp. 207–209, 1980.

[18] R. Chillarege and R. Iyer, "Measurement-based analysis of error latency," *IEEE Trans. Computers*, vol. 36, no. 5, pp. 529–537, May 1987.