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

Seungjae Han and Kang G. Shin, *Fellow, IEEE*

Abstract

Effective detection of failures is essential for reliable communication services. Traditionally, non-real-time computer networks have relied on behavior-based techniques for detecting communication failures. That is, each node uses heartbeats to detect the failure of its neighbors, and the end-to-end transport protocol (e.g., TCP) achieves reliable communication by acknowledgment/retransmission. Recently, there is a growing demand for reliable 'real-time' communication, but little research has been done on the failure detection problem. In this paper, we present two behavior-based failure-detection schemes — neighbor detection and end-to-end detection — for reliable real-time communication services, and experimentally evaluate their effectiveness. Specifically, we measure and analyze the coverage and latency of these detection schemes through fault-injection experiments. The experimental results have shown that nearly all 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.

Index Terms — Real-time communication, network failures, failure detection, fault-injection experiments.

*A subset of the materials of this paper was presented at **FTCS-27**, June 1997, in Seattle, WA.

1 Introduction

Network reliability is usually measured by message-loss rate and failure-recovery delay. Conventional computer network applications such as electronic mail, file transfer, or network file systems do not mandate fast failure recovery, but require correct delivery of all messages even if it takes a long (possibly unbounded) time. The datagram service used for such applications is often called 'best-effort' service, implying that the network attempts to deliver messages as quickly as possible, without guaranteeing any delivery-delay bound. In datagram networks, the failure of a network component is usually detected by its neighbor components (i.e., routers or gateways) using heartbeats. Upon detection of a failure, each node updates its routing table so as to route the traffic around the faulty component. Reliable transport protocols for datagram communication (e.g., TCP) guarantee the eventual (thus loss-free) delivery of messages between two end-points. In these protocols, the "grain" of failure detection is an individual message (or packet), and message losses are usually detected at the end-to-end level. For example, with the 'positive acknowledgment' method, the receiver informs the sender of the reception of each message (or a group of messages), so that the sender may detect delivery failures. The 'negative acknowledgment' method is an alternative to this, in which the receiver detects message losses and requests the sender or other servers to retransmit the missing messages.

Recently, computer networks are beginning to carry real-time messages for applications such as multimedia and distributed real-time control. Real-time communication is fundamentally different from best-effort service in that individual guarantees on such Quality of Service (QoS) as delay or throughput for *each* connection are the key requirement. Telephone service which guarantees 64 Kbps for each connection is an example of real-time applications. To achieve QoS guarantees in packet-switched networks, the network uses a virtual circuit. The network selects a path, reserves necessary resources along the fixed path, and transfer messages over this fixed path. This uni-directional virtual circuit is often called a *real-time channel*.

Real-time communication requires a different reliability goal from conventional non-real-time communication (i.e., best-effort service). Unlike non-real-time messages, the content of a real-time message is meaningful or useful only if it is delivered in time. Retransmitting real-time messages in case of their loss or corruption may be of little use to the underlying application, since their deadline is usually too tight to allow the retransmission of missing messages. As a result, loss-free real-time communication is very difficult to achieve without using expensive forward-recovery techniques (e.g., multiple-copy transmission [1], error correction coding [2]). However, fortunately, many real-time applications do not require such strict reliability as 'no message loss at all.' For example, infrequent loss of two or three frames in video/voice data streams is acceptable in most multimedia applications. Temporary message losses for a short period of time are also tolerable in real-time control applications because of the 'system inertia' characterized by the control system deadline [3]. Therefore, the grain of failure detection in real-time communication service is *channel failures* instead of individual message losses. A real-time channel is said to have 'failed,' if the rate of correct[1] message delivery within a certain time interval is below a certain threshold specified by the application.

Since a long service disruption will cause application/mission failures, effective failure detection with high coverage and low latency is essential to reliable real-time communication.

[1] both logically and temporally

For example, telephone service requires fast failure recovery so that humans may hardly notice the service disruption caused by network component failures. In telephone networks, an expensive failure-detection technique using hardware duplication/comparison is used to quickly detect switching-node failures [4], and spatial redundancy is applied to the physical layer to mask link failures. However, (hardware) duplication is not a viable solution for large-scale computer networks such as Internet, considering the complexity of designing routers and the innate heterogeneity of sub-networks. A key challenge is, therefore, how to achieve effective failure detection without relying on duplication, particularly for real-time communication.

To achieve this goal, we use "behavior-based" failure-detection schemes, which are similar to the existing techniques used for datagram networks. In our schemes, channel failures are detected at two different levels; at the link level and at the connection level. With the link-level detection scheme (called 'neighbor detection'), the health of a network component (i.e., node or link) is monitored by periodic heartbeat exchanges. With the connection-level detection scheme (called 'end-to-end detection'), the status of a real-time channel is monitored by its receiver node. The former is similar to the reachability test between gateways in the Internet Protocol (IP) [5], while the latter is, in principle, similar to the negative acknowledgment method. The deployment of our schemes can benefit from these similarities by easing the integration with the existing network protocols. We will not address the integration issue of our schemes with existing network protocols. The focus of this paper is on evaluating the effectiveness of our detection schemes. In particular, we measure and analyze their coverage and latencies in detecting channel failures, through fault-injection experiments on a laboratory testbed.

The rest of the paper is organized as follows. Section 2 presents the basic idea of our failure-detection schemes. Section 3 gives an overview of the fault injector used for experimental evaluation. Section 4 describes the experimental setup used. Section 5 analyzes the experimental data collected, and Section 6 concludes the paper.

2 Channel Failure Detection

We first present two behavior-based failure-detection schemes which do not require any special hardware support, so they can be used in *any* network.

2.1 Neighbor Detection

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 of time, it considers the silent neighbor failed and stops sending heartbeats to that node. This scheme is specified by two parameters: heartbeat interval t_h and a tolerable number, m, of heartbeat misses. When no heartbeat has been received from a node for $(m+1)t_h$ time units, the node is declared to have failed. Heartbeats do not carry any useful information, and regular messages can be used as heartbeats to reduce the additional traffic resulting from heartbeats. Explicit heartbeats are sent only if there are no regular messages for one heartbeat interval.

In networks where two nodes are connected by dual simplex links, a node cannot tell the

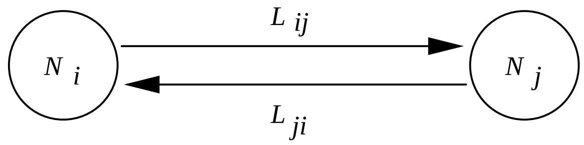


Figure 1: Two nodes connected by dual simplex links

failure of its neighbor nodes from that of the corresponding links by exchanging node heartbeats only. Instead of relying on sophisticated diagnoses, we treat all channels running through the "suspected" link as faulty. That is, when the link L_{ji} from node N_j to node N_i failed, the channels on the link L_{ij} from N_i to N_j will be considered failed, even if they are healthy, because N_i can no longer detect the crash of N_j or L_{ij} without healthy L_{ji} (Figure 1). Such channels should be torn down even if they themselves are healthy. This is reasonable because a channel cannot maintain dependable service if no one else can monitor its health.

2.2 End-to-End Detection

This scheme involves both the end nodes of a real-time channel. 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 of a channel do not discriminate channel heartbeats from regular data messages. Each channel heartbeat contains the sequence number of the latest data message. In this way, the destination node can count the number of data messages lost by monitoring the reception of data messages and the sequence numbers carried by channel heartbeats.[2] If the message-loss rate of the channel exceeds a threshold 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 periodically incremented. 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. Thus, when h_{max} is set to a value sufficiently large 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} value of a real-time channel should be chosen to fit the channel's traffic characteristics. In real-time communication, a contract on traffic characteristics and QoS levels is established between an application/client and the network before data messages are actually transferred. The network computes and reserves resources for a real-time channel from this information. The h_{max} value of a real-time channel should therefore be larger than the channel's minimum message interval specified in the 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 heartbeat interval of a channel is thus determined by the channel's traffic characteristics.

[2]This is easy to implement because real-time channel service guarantees in-order message delivery.

3 Fault-Injection Tool Set

The use of our failure-detection schemes is not restricted to a particular system/application. However, their effectiveness is intrinsically related to many system-dependent issues, such as operating system support, communication protocol implementation, and hardware capability. To fully explore such issues, we have implemented and evaluated the failure-detection schemes on a laboratory testbed, which consists of three nodes connected by point-to-point links.

This section describes the fault-injection tool set that we used for experimentally evaluating the failure-detection schemes. The basic design principles are explained first, and the implementation details are then described.

3.1 DOCTOR: An integrateD sOftware-implemented fault injeCTiOn enviRonment

Fault injection has long been viewed as a useful means of testing/evaluating fault-tolerant systems [6]. Numerous hardware-implemented fault injectors (HFIs) [7, 8, 9] have been developed and used for various experiments. However, as the complexity of contemporary computer systems 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) [10, 11, 12, 13, 14] 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, such as limited accessibility to hardware, perturbation to workloads, and poor time-resolution [6]. List below are the desirable features of a fault-injection tool for the evaluation of our failure-detection schemes.

1. The fault model should include faults on communication links or communication adaptor circuitry as well as faults inside a processing node such as memory, CPU, or bus.

2. The fault injector should be able to coerce the whole target system (i.e., consisting of multiple processing nodes) to follow a certain intended experiment scenario, which requires it to orchestrate all participants' operations.

3. Fault injection and data collection must require as few modifications to the target system as possible.

4. The intrusion into normal execution by fault injection and data collection should be minimized and isolated to obtain accurate measurements.

5. If clocks are not synchronized or clock skews among different processing nodes are unacceptably large, a special time-stamping technique without using the system clock should be employed for the accurate measurement of detection latencies.

To provide the above-mentioned features, we have developed an integrated fault-injection environment called DOCTOR, which provides a complete set of tools for automated fault-injection experiments: fault-selection tool, fault injector, synthetic workload generator, data monitor, post data-analysis tool. The fault-selection tool, EGM (Experiment Generation Module) is responsible for generating a set of fault instances to be injected. ECM (Experiment Control Module)

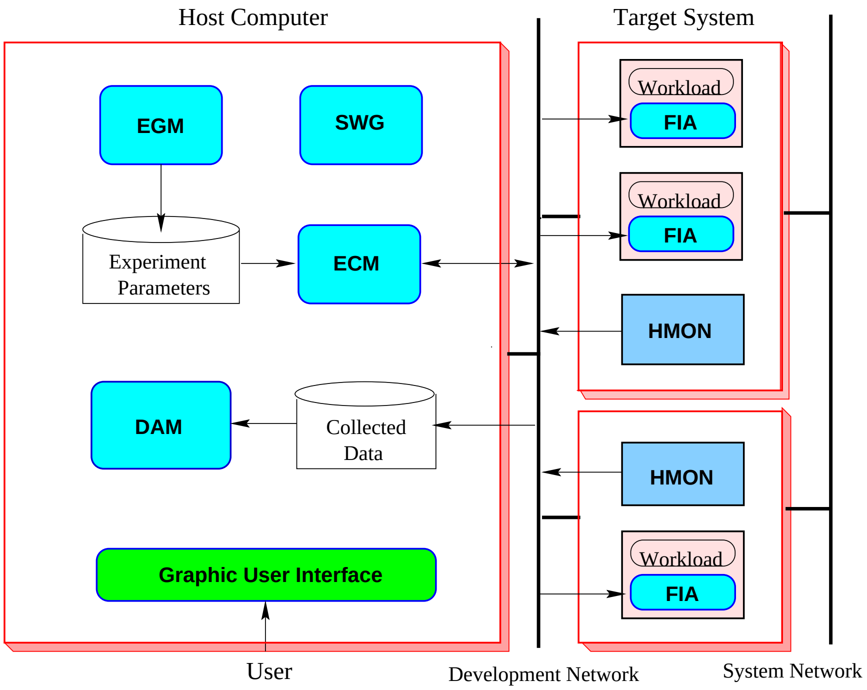


Figure 2: The organization of DOCTOR

and FIA (Fault Injection Agent) compose the fault injector. ECM is the run-time experiment manager and FIA performs actual fault injections under ECM's control. SWG (Synthetic Workload Generator) is to generate various artificial workloads. The data monitor, HMON (Hybrid MONitor), collects the experimental data at run time. DAM (Data Analysis Module) analyzes the collected data off-line after completing the experiment.

One distinct structural property of DOCTOR is the separation of software components of the host computer from those of the target system, as depicted in Figure 2. ECM runs 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. Another advantage is high portability since the highly system-dependent part is isolated from the rest. It also eases the synchronization of multiple target nodes, i.e., orchestrating the execution of the distributed system under test. The target system may consist of several processing nodes, each of which can be a single- or multiple- processor system. In Figure 2, several processing nodes are connected via a 'system network' and are linked to the host computer by a separate 'development network.' Although the isolation of the experiment control traffic from the application traffic is ideal, the same physical network may be used for both the system and development networks, since the traffic over the development network is usually concentrated on before starting and after finishing each experiment, so that it will not disturb the application traffic over the system network at run time.

In the remainder of this section, we discuss the implementation of the fault injector and the data monitor, two most important modules in DOCTOR.

3.2 Software-implemented Fault Injection

Faults affect various aspects of the system state or operational behavior, such as memory or register contents, program control flow, clock value, the condition of communication links, etc. Modifying memory contents has been a basic technique of software-implemented fault injectors. Faults are likely to (eventually) contaminate certain parts of memory, so memory faults can represent not only RAM errors but also emulate faults occurring in the other parts of the system. The memory fault model is quite powerful, but some faults may affect system memory contents in a very subtle and nondeterministic way, and hence, it is very difficult to emulate such a faulty behavior with memory fault injection alone. For instance, it is difficult to emulate erroneous communication-related functions by just changing memory contents. A more sophisticated fault model is therefore required.

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 or system-behavior-level errors. In essence, low-level faults are injected into addressable storage space, such as caches, main memory, and CPU registers, while higher-level errors are injected by manipulating system software services such as message transmission or system-call services. The fault-injection techniques used by DOCTOR are briefly described below. (Details on the fault-injector implementation can be found in [15].)

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. For better controllability, DOCTOR allows faults to be injected only into a certain memory section of a particular task, such as text area, global variable area, or stack/heap area. Timers are usually used to trigger the injection of memory faults.

Processor fault: Faults can occur in CPU components such as data registers, address registers, the data fetching unit, control registers, the op-code decoding unit, ALU, and so on. However, accessibility to hardware components is usually limited. To overcome this difficulty, we inject erroneous effects rather than faults themselves, by emulating the consequences of CPU faults at an architecture-independent level; hence, the contents of CPU registers or instruction codes are the targets of fault injection. The timing of fault injection can be randomly selected as in memory-fault injections, or can be controlled to be the time when a specific instruction of a specific task is executed. We use the 'trap' insertion technique to interrupt the execution of the target task at the particular instruction. The insertion timing of trap instructions is usually controlled by using timers. Transient bus faults and permanent faults can be emulated as well, but these fault types are not used in the experiments reported in this paper.

Communication error: Errors in communication links can be emulated using a special fault-injection layer which is embedded into the protocol stack of the underlying OS at the compile time. The user can specify the intended fault behavior by supplying plug-in functions which will be called by the fault-injection layer, 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).

3.3 Non-intrusive Data Monitoring

The data monitoring instrument to trace the run-time behavior of the system under test and to collect various dependability parameters is an important component in fault-injection experiments. Conventional hardware monitors probe a fixed number of physical signals to obtain hardware-level data. Though such hardware monitors eliminate interference in the monitored system, they lack the ability of capturing a wide range of software-level events. Meanwhile, software-implemented monitors can cause unpredictable interferences. Such software monitors usually rely on the operating system service to snapshot the timing of a certain event occurrence. Time-stamping by software involves two potential problems, the poor resolution of time-stamps and the overhead of time-stamping. The problem of time-stamping is complicated even further in distributed systems where events can occur asynchronously in different processing nodes. Comparison of time-stamps is meaningful only if the clock services are synchronized with sufficient accuracy. However, it is often an impractical assumption since the typical clock skew bound is in the order of millisecond, while the required time-stamp resolution may easily reach the microsecond order.

To overcome the limitation of pure hardware monitors and software monitors, we have developed a hybrid data monitor. We envision the data collection process consisting of three steps: event probing, time-stamping, and event logging. In DOCTOR, event probing is done by the special code inserted into the monitored objects, such as system software, application program, fault injector, or fault-tolerance mechanisms. Events may include system call invocations, context switches, interrupts, fault injections, fault detections, and so on. Then, the probed events are time-stamped and logged by the special hardware (i.e., HMON).

HMON is implemented as a separate board to reduce the hardware dependency on the processor board and to allow a HMON to be shared by several processor boards on the same backplane bus (i.e., VME bus). The logical architecture of HMON is shown in Figure 3. HMON is a passive device[3] and communicates with other processor boards via a memory-map interface. Thus, when a certain address in the HMON memory-map is accessed, it performs a pre-specified action. For example, when a memory write operation is requested at specific addresses, the data content is combined with a time-stamp to make a logging entry, and the new entry is stored into the on-board memory. HMON is not equipped with disks, so the data collected in the on-board memory is dumped to the host machine through the development network after experiments or during experiments.

For distributed data collection, a HMON is assigned to each processing node, and a dedicated simple network, called the 'HMON network,' is used to connect HMON boards. Each HMON has one output port and 7 input ports to communicate with other HMONs, and two HMONs are linked using a twisted-pair cable. Rather than directly synchronizing the internal timer of each HMON, we use an indirect way. When a signal comes through an input port from another HMON, the identifier of the input port is stored with current time-stamp. The HMON which generates the signal also logs the event with its time-stamp, so that in the post analysis all the events can be totally ordered by time-stamps. It is a cheaper and more accurate way to achieve the same effect as if all time-stamps are generated by a centralized clock.

Young also used a hardware/software hybrid monitor [16], where a commercial logic analyzer

[3] In terms of bus arbitration.

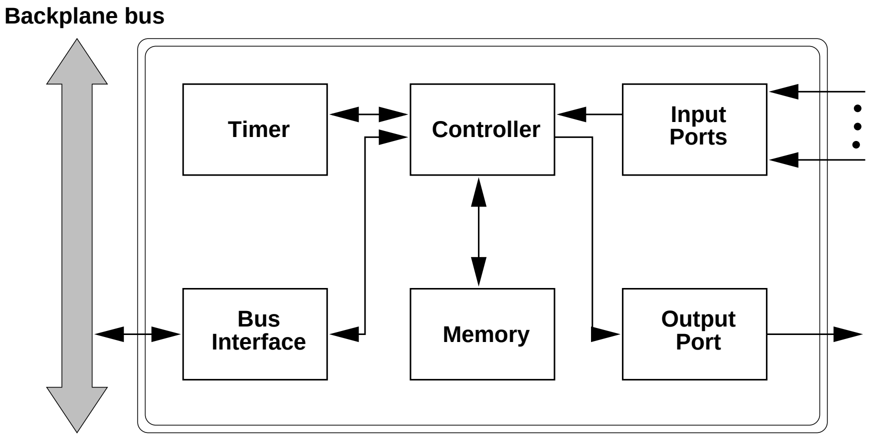


Figure 3: Architecture of HMON

was used to probe hardware signals and a software monitor checked the system status to assist the control of the hardware monitor. However, his major concern lied in the measurement of low-level system activities, and did not consider the distributed environment. By contrast, our approach is more oriented toward software-implemented monitors, and since the control logic of HMON is implemented with a FPGA chip, the functionality of HMON can be easily re-programmed.[4]

4 Fault-Injection Experiment Setup

This section describes the hardware/software configuration of the experimental platform and the specification of fault-injection experiments conducted on the platform.

4.1 Testbed

As shown in Figure 4, the experimental platform consists of three nodes, Nodes 1–3. Each node is a VME bus-based multiprocessor system with two CPU boards equipped 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 [18] is equipped with each node. Nodes 1 and 2 are connected by two simplex links (i.e., optical fibers), one in each direction. The same type of connection exists between Node 2 and Node 3. In addition, a HMON board is added to each node for data collection, and all HMON boards are connected by the HMON network. A SUN workstation serves as the host machine, and is connected to nodes through an Ethernet (i.e., development network). Nodes are not equipped with disks, and application/system software is downloaded from the host machine.

[4] For instance, we used HMON for the performance measurement of real-time channel protocol which is used in our experiment [17].

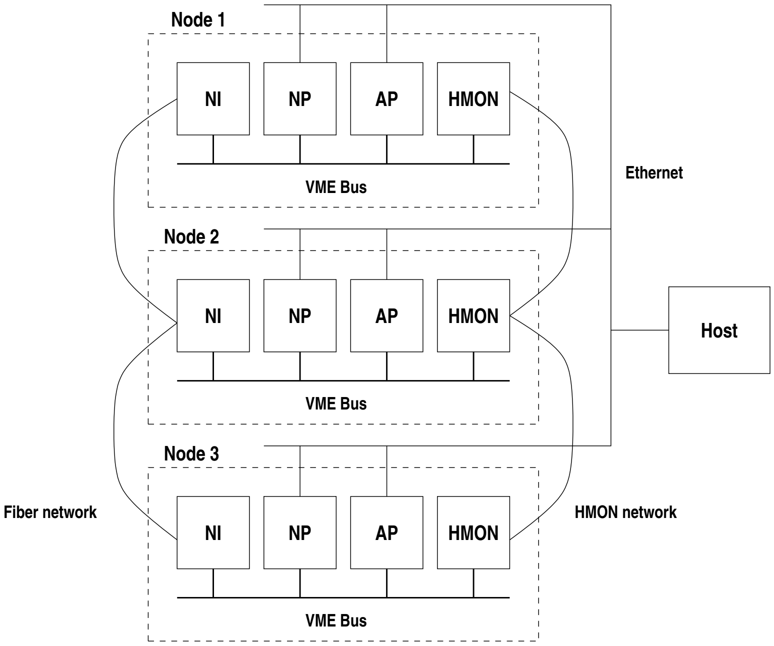


Figure 4: Configuration of the experimental platform

An extended version of the pSOS^{+m} real-time OS [19] 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 the messages received by the NP. NP employs a derivative of x-kernel 3.1 [20] 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. (Note that these restrictions are typical of a real-time OS.)

Each NP executes the real-time channel protocol described in [21]. Figure 5 gives an overview of the NP protocol stack. The protocol stack includes protocols for application program interface (API), network management (NM), remote procedure call (RPC),[5] transport-level fragmentation (FRAG),[6] network/data-link layer (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 for channel establishment/teardown messages. The HNET protocol covers the function of the network layer and part of the data-link layer. The run-time message scheduler is embedded in it. Figure 6 illustrates the run-time protocol activity for real-time message passing when a real-time channel is established from from Node 1 to Node 3 through Node 2. NM and RPC protocols are not shown in Figure 6, because they are used only for channel establishment. Also, notice that API and FRAG protocols are executed only at end-nodes.

The NP system software, x-kernel, uses a non-preemptive scheduling policy with 32 prior-

[5] A modified version of x-kernel's CHAN protocol.
[6] A modified version of x-kernel's BLAST protocol

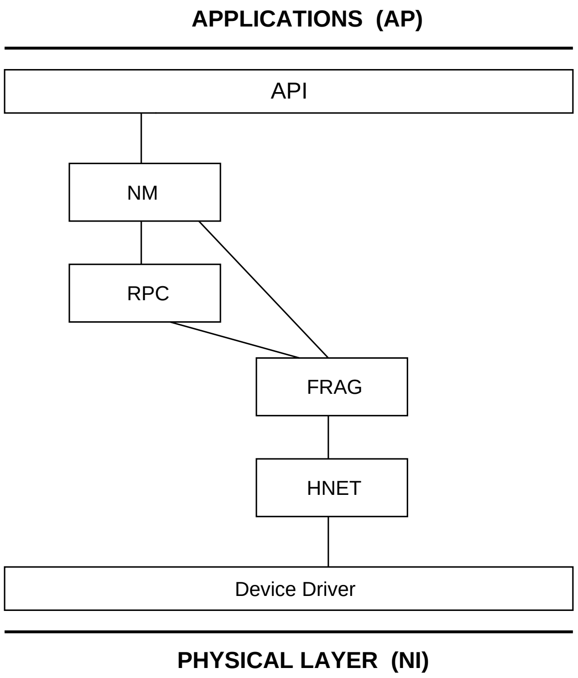


Figure 5: The protocol stack in NP

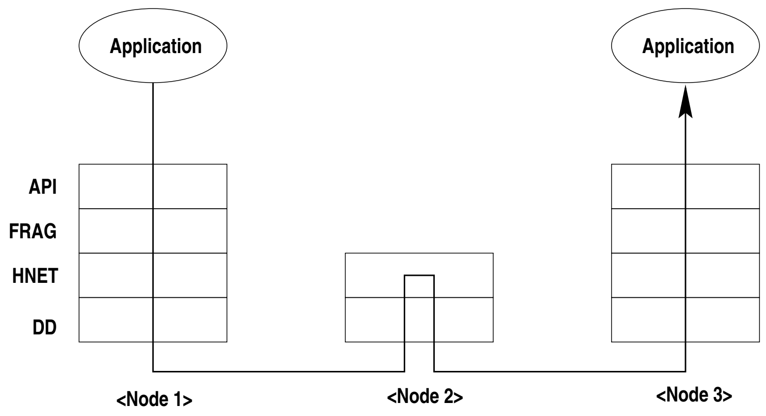


Figure 6: Real-time message passing at run-time

ity 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 [22].

At the first stage of an experiment, EGM analyzes the executable images of the fault-injection targets, and extracts their symbol/object-code information, and generates experiment parameters such as fault injection scripts (i.e., fault types and injection timings), the start/quit time of each experiment, and so on. Then, based on these parameters, ECM sends commands to FIAs about the fault-injection plan and the synchronization of each run. FIA executes the commands from ECM and reports its activities to HMON. Faults were injected into NP and a FIA resides in each NP. FIA runs as a separate process or runs on the interrupt thread depending on the type of fault to be injected. The function of the FIA on NP other than fault injection is minimized; the communication with ECM is indirectly done by a FIA proxy, which is running on AP, through the Ethernet, and the FIA communicates with the FIA proxy by the direct memory access through the VME bus. In this way, we avoid the need of installing the Ethernet protocol in NP, which may interfere the workload execution in NP (i.e., real-time communication over the optical fiber network). The FIA proxy is also responsible for uploading the data collected by HMON to the host machine.

The experiment was fully automated, so that multi-run experiments were done without human intervention. In a fault-injection experiment, one of the factors that determine the quality of analysis results will be the number of runs.[7] Therefore, it is very useful to automate multi-run experiments. The key problem in experiment automation is the synchronization and re-initialization of several processes involved. The level of re-initialization required depends on the status of the target system after completing each run. In some cases, it may suffice to restart only application workloads, but here we had to reset the whole system because fault injection can corrupt the OS area. Each run consists of 5 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 software (including OS and fault-injector) 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 through FIA proxy.

Step 4: FIA waits until applications establish real-time channels. After the message transmission is started, FIA starts fault injection and HMON collects data.

Step 5: 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.

[7] Each fault-injection experiment with specific workloads is called a *run*.

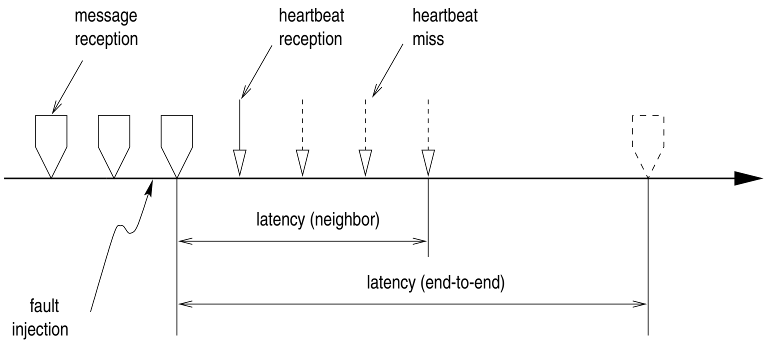


Figure 7: Failure-detection latency

4.2 Experiment Goal

A node-heartbeat generator runs as a separate process in each NP. It is periodically executed and checks the 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 over the corresponding link. On the receiving side, the heartbeat checker regularly checks the flags which are set when a message is received over the associated link. The heartbeat checker resets the flags after checking. If the heartbeat checker finds a flag not set for a period longer than the specified threshold, it declares a failure detection. Channel heartbeat generation and checking are done similarly.

The end-to-end detection scheme will uncover *all* channel failures, so the main concern is its detection latency. Under this scheme the receiver node (i.e., the destination node) has to wait for the loss of a specified number of messages before declaring a failure occurrence. Thus, the failure-detection latency is tightly coupled with the channel failure model which is specific to applications. The neighbor detection scheme has a much weaker dependency on the channel failure model than the end-to-end scheme, because it monitors the behavior of a neighbor node, rather than that of a real-time channel. Therefore, if a neighbor node (or the incoming link from it) fails and stops sending heartbeats, the neighbor scheme can quickly declare channel failures before the end-to-end scheme detects them. Figure 7 illustrates the failure-detection latency of the two detection schemes. In our experiment, we define the failure-detection latency as the duration between the time of the last data message delivered correctly and the time of failure detection.

Though the neighbor scheme may have smaller failure detection latencies than the end-to-end scheme, it may result in less than perfect detection coverage. For example, if a node is not operating correctly in terms of message processing, but still generates some node heartbeats or propagates part of regular messages, the neighbor scheme will not detect channel failures. Even though the faulty node or link become silent eventually, the detection latency of the neighbor scheme may be larger than that of the end-to-end scheme. Therefore, enhancing the coverage of the neighbor scheme without extending detection latencies is an important issue. For instance, instead of running as a separate process, the heartbeat generator can run on top of the clock interrupt thread. 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

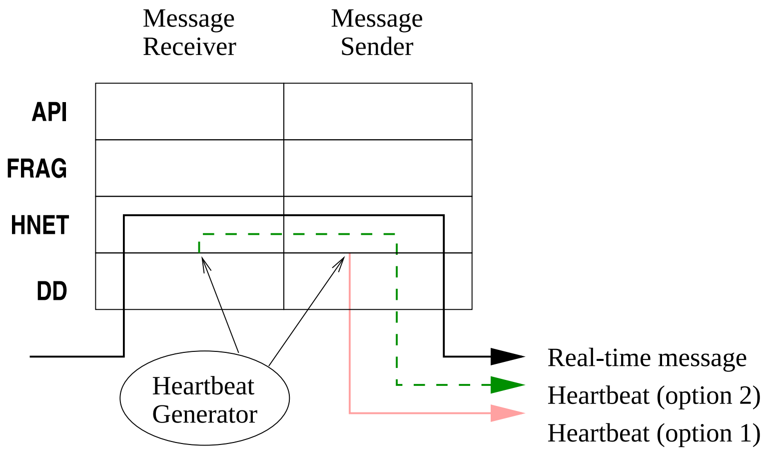


Figure 8: Two implementation options for transmitting node heartbeats

other interrupts are disabled. Second, even when the OS task scheduler or the message scheduler hangs, heartbeats will be sent out, which will lower detection coverage.

Not only the mechanism of generating heartbeats but also the transmission path of heartbeats will influence the efficiency of the neighbor detection scheme. In the experiment, we tested the following two heartbeat-transmission mechanisms:

(i) Sending heartbeats as best-effort messages (option 1),

(ii) Sending heartbeats as real-time messages (option 2).

Figure 8 depicts the transmission path of heartbeats in the two implementations. The rationale behind the option 2 is that, the closer the transmission path of heartbeats is to that of regular real-time messages, the higher the detection coverage will be. The obvious shortcoming of the option 2 is the execution overhead of sending heartbeats as real-time messages, e.g., queuing by the real-time message scheduler.

The overall experimental goal is to measure the efficiency of the neighbor scheme (i.e., its detection coverage and latency) and to examine if and how much sending node-heartbeats as real-time messages can enhance the detection coverage of the neighbor scheme.

4.3 Experiment Specification

We describe the details of the experiment plan using the $\boldsymbol{FARM}$ specification model [7], which formulate a fault-injection experiment with four attributes: a set of faults $\boldsymbol{F}$, a set of activations $\boldsymbol{A}$, a set of readouts $\boldsymbol{R}$, and a set of derived measures $\boldsymbol{M}$.

As the $\boldsymbol{A}$ attribute, which specifies the workloads used to exercise the target system, real-time channels were established from Node 1 to Node 3 through Node 2 on the testbed. The end-to-end delay requirement of the real-time channels was 30 msec and the application program generated real-time messages regularly once every 50 msec without any burst. Since messages were generated periodically, no 'channel heartbeat' needs to be injected into the message stream of real-time channels. A channel failure is said to occur if no message is delivered for more than 400 msec (late messages are treated the same as lost messages). The interval of 'node heartbeats' was set to 60 msec and the number of tolerable heartbeat misses was set to 2. Hence, if heartbeats are not received for three consecutive heartbeat intervals (i.e., 180 msec), a failure is declared by the neighbor scheme. For both node and channel heartbeats, the heartbeat generators and checkers were periodically invoked once every 30 msec.

As the ***F*** attribute, three types of faults were injected: memory faults, CPU register faults, and communication faults. For all fault types, transient single-bit toggle faults were used. According to the common practice in software-implemented fault injection, we use the term 'transient' to mean the opposite to 'permanent.' For example, register faults are transient in a sense that the corrupted register contents can be overwritten by the subsequent instructions. A communication fault corrupts only one message, so it is also transient. The faults injected into memory are also transient, but, when the program text area is corrupted, they will have properties similar to permanent memory faults. Because we are interested in the detection of failures which occur at components on real-time channel paths, faults were injected into only the NP of Node 2, the intermediate node, and were injected during the transmission of data messages after real-time channels are established. Therefore, only the HNET and DD protocols were selected as fault-injection targets among the NP protocols. From the operating system modules, the task manager (TM) and the clock service (CS) were chosen as fault-injection targets.

Memory faults were injected into the random locations of the text area of the target programs 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.[8] 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 experiment setup. 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 trap-driven method was used to trigger a fault-injection. Thus, at a randomly selected time, FIA substitutes a trap instruction for the instruction in the target address. When the instruction in the target address is being executed, the current task is interrupted and a fault is injected into a register used by the target instruction. In addition, we injected faults into CCR (condition code register) to study the effects of faults in control registers. To maximize the chance of fault activation, CCR faults were injected only when conditional branch instructions were executed. We also emulated the faults in communication links. Since the impacts of such errors as message drop or message data corruption are straightforward, we injected corruption errors into the message header part. To this end, the fault-injection layer was inserted between DD and HNET.

As the ***R*** attribute, HMON collects time-stamped data of such events as message generation, message relay, message reception, fault injection, failure detection, and heartbeat generation/reception. Note that we need distributed monitoring, since these events are generated by different processing nodes.

After each experiment, we measured (i) the channel-failure rate, and (ii) failure-detection

[8] Some memory faults in the data area have the possibility of being overwritten, which can affect their activation or propagation.

coverage/latency of the two detection schemes. These correspond to the M attribute. First, the result of each fault injection is classified into 3 categories: *no error*, *tolerated 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. 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. The failure-detection latency is the elapsed time since the last data message delivered correctly, when a failure is detection.

5 Analysis of Experimental Results

The experimental data were collected from more than 15,000 runs, each of which took about 70 seconds: 40 seconds for experiment setup plus 30 seconds for executing the experiment. We analyze the collected data from three perspectives: fault manifestation, failure-detection coverage, and failure-detection latency. The same experiments were performed with two different implementations of the neighbor scheme. In the first set of experiments (Experiment 1), node heartbeats were transmitted as best-effort messages, i.e., option 1. In the second set of experiments (Experiment 2), node heartbeats were transmitted as real-time messages over a special-purpose real-time channel, i.e., option 2.

5.1 Fault Manifestations

Table 1 shows the composition of fault manifestations. Before analyzing the experimental data, we would like to address the repeatability of the experiments. In Table 1, the fault manifestations observed in Experiment 1 do not exactly match those of Experiment 2, although the overall trend of fault manifestations is similar. It is mainly because fault instances were randomly generated for each run, so that the fault instances injected in Experiment 1 might be different from Experiment 2. Even if we injected the same fault instances, we might not get the same results. In general, the repeatability of fault-injection experiments is limited, because the granularity of controlling the injection timing is usually too coarse to inject faults at the exactly same moment repeatedly. The target system may follow slightly different execution paths in different experiments, particularly in the distributed system environment as ours.[9] Second, the number of runs in each case is not the same. The primary reason for this is that fault injection was not triggered in some runs, and we discarded such runs. In case of memory fault injection, such a case never occurred, but in case of register fault injection, it did occur when the target instructions were not executed within the run duration. Another reason for discarded runs is the failure in initializing the real-time channels or synchronizing fault injector modules.

Although a large number of runs were executed, the obtained composition of fault manifestations may not be representative, because our experiments are designed to increase the fault activation rate. Since the fault manifestations will be greatly affected by the types, targets, and injection timings of faults, a representative subset of the real fault population should be injected in order to obtain the representative values. However, the derivation of a representative fault

[9] [23] reports the occurrence of multiple symptoms with the injection of same faults.

		Experiment 1				Experiment 2			
		HNET	DD	TM	CS	HNET	DD	TM	CS
Memory	runs	392	387	393	387	396	398	398	399
	cf	28.1%	9.3%	17.8%	15.8%	31.3%	10.3%	16.8%	18.5%
	ok	0.8%	0.3%	0.0%	0.3%	2.3%	0.0%	0.0%	0.8%
	no	71.2%	90.4%	82.2%	84.0%	66.4%	89.7%	83.2%	80.7%
Dx	runs	387	373	343	373	398	385	327	382
	cf	17.1%	13.1%	11.4%	2.4%	13.3%	11.7%	13.1%	2.6%
	ok	3.1%	8.8%	0.0%	0.0%	3.5%	3.4%	0.6%	0.0%
	no	79.8%	78.0%	88.6%	97.6%	83.2%	84.9%	86.2%	97.4%
Ax	runs	389	359	379	304	396	388	391	318
	cf	68.1%	59.9%	61.5%	45.5%	74.2%	62.9%	70.6%	41.5%
	ok	6.2%	6.4%	0.0%	17.1%	2.3%	4.9%	2.3%	15.7%
	no	25.7%	33.7%	38.5%	37.5%	23.5%	32.2%	70.6%	42.8%
CCR	runs	307	289	340	345	340	318	334	357
	cf	7.2%	13.8%	44.1%	0.0%	12.6%	20.4%	52.7%	0.6%
	ok	18.6%	17.3%	9.4%	0.9%	10.9%	6.0%	11.7%	0.8%
	no	74.3%	68.9%	46.5%	99.1%	76.5%	73.6%	35.6%	98.6%
Msg	runs	386				397			
	cf	16.8%				17.4%			
	ok	0.0%				0.0%			
	no	83.2%				82.6%			

Table 1: Fault manifestations, where cf = channel failure; ok = tolerated error; no = no error

set is an open problem except in very small-scale systems. A more meaningful implication of our experimental result is the general tendency and the pattern of system behaviors caused by faults.

In general, the fault manifestations differed by fault types. For example, the faults injected into address registers resulted in much higher channel-failure rates than those into data registers. The targets of fault injection also affect the fault manifestations. For example, faults injected into CS had a relatively low activation rate, as compared to TM.

Several interesting behaviors were observed on the errors that were tolerated by the system. Most of the tolerated errors were due to 'message deadline violations' and some of them were due to 'message losses.' However, very few 'message data corruption' were found. This is because the message data part is stored by the device driver upon arrival of the message, and is not copied or modified by other protocols in x-kernel, so that the chance of message corruption is low. In some cases of message deadline violations, the delayed messages were eventually delivered, and the system behavior returned to normal. In some other cases, the abnormal behavior keeps on occurring and disappearing without causing a channel failure or heartbeat interruption. When such an abnormal behavior continues for longer than three consecutive heartbeat intervals but less than 400 msec, the neighbor scheme signals a failure detection, even if no channel failure

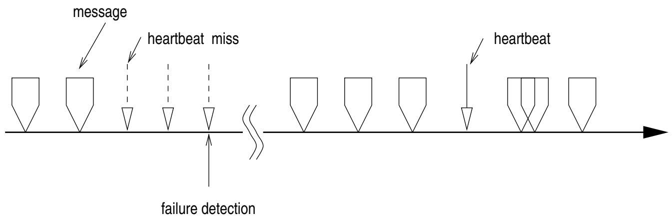


Figure 9: An example of false alarm

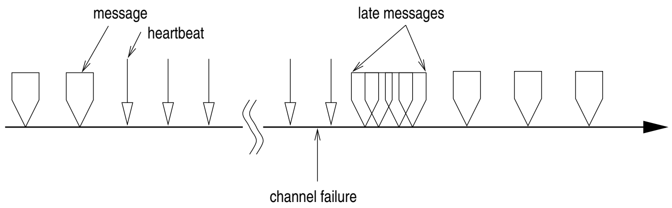


Figure 10: An example of channel failure undetected by the neighbor scheme

had actually occurred. We call this a *false alarm*. An example false alarm is depicted in Figure 10. According to our experimental results, false alarms are rare but do occur. If we reduce the number of missing node-heartbeats before declaring a failure in the neighbor scheme, the likelihood of false alarm will increase. The possibility of false alarm keeps the neighbor scheme from quickly declaring a failure upon missing a single heartbeat, especially when the heartbeat interval is small.

5.2 Failure-Detection Coverage

The end-to-end detection scheme had always perfect coverage in detecting channel failures, as expected. Meanwhile, some channel failures went undetected by the neighbor scheme. A typical failure symptom which was not detected by the neighbor scheme is illustrated in Figure 10, where faults cause only data messages to be delayed or dropped without affecting the transmission of node heartbeats. The measured detection coverage of the neighbor scheme (both Experiment 1 and 2) are summarized in Table 2.

	Experiment 1				Experiment 2			
	HNET	DD	TM	CS	HNET	DD	TM	CS
Memory	87.3%	94.4%	95.7%	85.5%	96.8%	100%	100%	98.6%
Dx	72.7%	100%	97.4%	44.4%	79.2%	97.8%	100%	70.0%
Ax	98.9%	100%	91.8%	92.0%	100%	100%	100%	96.2%
CCR	68.2%	92.5%	71.3%	N/A	90.7%	100%	100%	100%
Msg	100%				100%			

Table 2: Detection coverage of the neighbor scheme

The first observation from Table 2 is that the coverage of the neighbor scheme was relatively insensitive to the fault types or locations with some exceptions, in both Experiment 1 and 2. The faults in the memory (i.e., the program text area) or data/address registers were detected very well, whereas CCR errors resulted in lower coverage. This is because CCR errors cause incorrect control flows which are difficult to detect without special hardware support (e.g., a watchdog processor), while errors in the memory or data/address registers can be detected by CPU-intrinsic fault-detection mechanisms for bus errors, unaligned memory accesses, etc. The faults injected into the CS module, particularly data register faults, also resulted in lower coverage than those into other modules. (N/A indicates that no channel failure occurred despite fault injections.)

The second observation is that the heartbeat-transmission mechanism significantly affects the performance of the neighbor detection scheme. Comparison between Experiment 1 and 2 indicates that the coverage of option 2 (i.e., Experiment 2) is higher than that of option 1, and near-perfect coverage was observed for most cases in Experiment 2. The reason for the high coverage in Experiment 2 is the sharing of program codes in processing real-time messages belonging to different real-time channels. That is, node heartbeats went through the same execution path as real-time data messages in Experiment 2 (see Figure 8). If a fault affects the real-time channel which carries data messages, it is very likely to affect the real-time channel for node heartbeats as well. Recall that in x-kernel, whenever necessary, a shepherd thread is spawned to process a new message, and all shepherd threads execute the same program code. Under this protocol processing model, 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, which will end up with a transient error. 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, which will not hurt the detection coverage of the neighbor scheme. Only when the channel-specific data (i.e., a link-deadline) is corrupted, the corresponding channels will fail while other channels won't. However, according to our experimental results, the chance of such occurrences is very low. The property of program code sharing in message processing is not limited to x-kernel and is common in most conventional protocol implementations such as in BSD Unix.

5.3 Failure-Detection Latency

If the channel heartbeat checker continuously monitors the reception of messages, the end-to-end scheme will always result in a constant detection latency, which is 400 msec in the current experimental setup. However, since the channel heartbeat checker was periodically invoked once every 30 msec, a failure may not be declared immediately after the passage of 400 msec without any traffic (i.e., channel heartbeat or data message). Thus, in the current setup, the worst-case detection latency is 430 msec. The expected detection latency of the neighbor scheme is 180 msec, i.e., duration of three consecutive heartbeat losses. However, the actual detection latency measured was about 190–200 msec on average (Table 3). More than 200 msec or less than 180 msec latencies were occasionally observed.

There are three reasons for the large latencies. First, the heartbeat generator and checker were periodically invoked once every 30 msec. As a result, heartbeats might not be generated

	Experiment 1				Experiment 2			
	HNET	DD	TM	CS	HNET	DD	TM	CS
Memory	198.3	192.8	201.7	193.0	197.1	198.0	196.5	197.5
Dx	192.5	200.4	194.5	191.5	201.5	204.1	201.3	203.3
Ax	193.7	193.2	193.4	194.6	204.1	204.6	203.6	203.6
CCR	196.3	196.8	194.3	N/A	206.9	209.7	203.4	205.5
Msg	193.5				196.8			

Table 3: Detection latency of the neighbor scheme (mean values in msec)

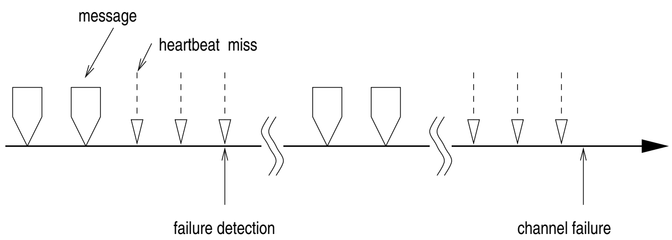
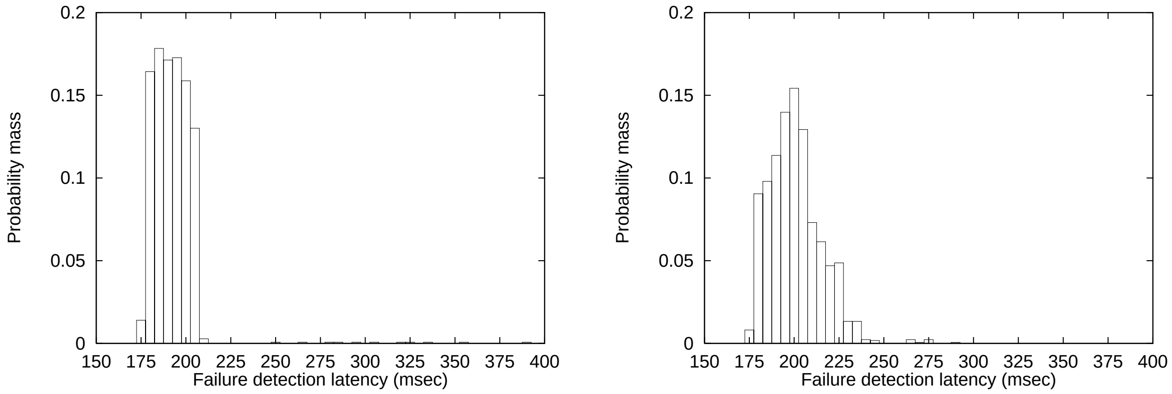


Figure 11: An example of early failure detection

exactly 60 msec after the last message transmission, and a failure may not be declared immediately after the passage of 180 msec without any traffic. Second, faults might delay or drop messages for some time 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. Third, different error-propagation delays for real-time messages and node heartbeats can result in a long detection latency; faults may affect real-time messages quickly and affect heartbeats slowly.

Small detection latencies can also be explained by the error-propagation delay. In extreme cases, even negative detection latencies were observed, albeit rarely. The negative detection latency means that a failure is detected before it actually occurs. This happens when message transmission is stopped for longer than 180 msec (i.e., failure detection by the neighbor scheme) and then message transmission is resumed for a while but the channel eventually fails. In such a case, the detection precedes the channel-failure occurrence. Figure 11 illustrates this case. Notice that the situation is the same as false alarm except the eventual failure.

Comparison of fault detection latencies between Experiment 1 and 2 reveals that Experiment 2 resulted in a slightly larger average than Experiment 1. For better understanding, the overall distributions of detection latencies are compared in Figure 12. We can see that the population of long latencies was increased in Experiment 2 as compared to that of Experiment 1. It may be because the error-propagation process is different under the two implementations.



(a) Experiment 1 (b) Experiment 2

Figure 12: Comparison of detection latency distributions

5.4 Workload Dependency

We also conducted fault-injection experiments by setting up multiple real-time channels. The existence of peer channels did not have a significant impact on the performance of the failure-detection schemes. When a channel suffered an abnormal behavior, other channels experienced similar symptoms in most cases.

6 Conclusions

In this paper, we introduced a new failure model for real-time communication services and presented two behavior-based failure-detection methods — neighbor and end-to-end detection — for dependable real-time communication. We then investigated the effectiveness of these detection methods through fault-injection experiments.

Though the basic idea of our detection schemes is not totally new, our work is unique in that we applied the detection schemes to real-time communication and evaluated their efficiency. In most previous research on dependable real-time communication, even the definition of failures is not clearly given and failures are assumed to be detected immediately upon their occurrence. In [24, 25, 26], the "I am alive" heartbeat method is implicitly assumed to detect all failures, but this assumption is valid only if the network components have the *failure-silent* characteristic. In general, the failure-silent semantic is difficult to achieve without relying on redundancy.

Our experimental results on a laboratory testbed, which was designed without any particular consideration for fault-tolerance, indicates that the "I am alive" heartbeat method (i.e., the neighbor scheme) occasionally have long latencies and imperfect coverage, if it is not carefully designed. By contrast, the end-to-end scheme can detect the failures which had gone undetected by the neighbor scheme. However, despite its perfect coverage, the end-to-end scheme has such weaknesses as high overhead, long latency, and the inability to locate failures. Thus, improving the detection coverage of the neighbor scheme is crucial for effective failure detection. To this end, we employed a new heartbeat-transmission technique in which heartbeats are sent along a dedicated real-time channel instead of being sent as best-effort messages. This technique greatly improved the detection coverage of the neighbor scheme up to almost 100 % with slightly-increased detection latencies. The most important contribution of this paper is that we showed that the 'near-failure-silence' behavior of network nodes can be achieved without applying expensive redundancy techniques in real-time communication networks. This result contrasts with the previous experimental results on the failure-silence behavior in various application areas [9, 27, 28, 23, 29].

Acknowledgments

We would like to thank Harold Rosenberg and Jaehyun Park for their contribution in the implementation of DOCTOR. Atri Indiresan provided the skeleton of the real-time channel protocol used in our experiment. The work reported in this paper was supported in part by the National Science Foundation under Grant MIP-9203895, the Office of Naval Research under Grant N00014-94-1-0229, and Mitsubishi Electric Research Laboratory, Cambridge, MA.

References

[1] P. Ramanathan and K. G. Shin, "Delivery of time-critical messages using a multiple copy approach," *ACM Trans. Computer Systems*, vol. 10, no. 2, pp. 144–166, May 1992.

[2] A. Banerjea, "Simulation study of the capacity effects of dispersity routing for fault tolerant realtime channels," in *Proc. ACM SIGCOMM*, pp. 194–205, 1996.

[3] 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.

[4] 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.

[5] D. Comer, *Internetworking with TCP/IP*, Prentice-Hall, 1995.

[6] M. Hsueh, T. Tsai, and R. Iyer, "Fault injection techniques and tools," *IEEE Computer*, pp. 75–82, April 1997.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] L. Young, C. Alonso, R. Iyer, and K. Goswami, "A hybrid monitor assisted fault injection environment," *Dependable Computing for Critical Applications 3*, vol. 3, pp. 281–302, 1993.

[17] A. Indiresan, *Exploring Quality-of-Service Issues in Network Interface Design*, PhD thesis, University of Michigan, 1997.

[18] *Fibre Channel Physical and Signalling Interface (FC-PH)*, American National Standards Institute, rev. 3.0 edition, June 1992. Working draft.

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

[20] 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.

[21] 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.

[22] 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.

[23] D. Stott, G. Ries, M. Hsueh, and R. Iyer, "Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection," *IEEE Trans. Computers*, vol. 47, no. 1, pp. 108–119, 1998.

[24] H. Kopetz and G. Grunsteidl, "TTP - a time-triggered protocol for fault-tolerant real-time systems," in *Proc. IEEE FTCS*, pp. 524–533, 1993.

[25] A. Banerjea, C. Parris, and D. Ferrari, "Recovering guaranteed performance service connections from single and multiple faults," Technical Report TR-93-066, UC Berkeley, 1993.

[26] R. Kawamura, K. Sato, and I. Tokizawa, "Self-healing ATM networks based on virtual path concept," *IEEE Journal on Selected Areas in Communications*, vol. 12, no. 1, pp. 120–127, January 1994.

[27] 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.

[28] J. Silva, J. Carreira, H. Madeira, D. Costa, and F. Moreira, "Experimental assessment of parallel systems," in *Proc. IEEE FTCS*, pp. 415–424, 1996.

[29] S. Chandra and P. Chen, "How fail-stop are faulty programs?," in *Proc. IEEE FTCS*, pp. 240–249, 1998.

Biography

Seungjae Han is a post-doctoral research fellow of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, the University of Michigan, Ann Arbor, Michigan. He received both the B.S. and M.S. degrees in Computer Engineering from Seoul National University, Seoul, Korea, in 1989 and 1991, respectively, and the Ph.D degree in Computer Science and Engineering from the University of Michigan in 1998. His research interests include computer networks, fault-tolerant systems, and real-time systems.

Kang G. Shin is Professor and Director of the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, Michigan.

He has authored/coauthored about 600 technical papers and numerous book chapters in the areas of distributed real-time computing and control, computer networking, fault-tolerant computing, and intelligent manufacturing. He has co-authored (jointly with C. M. Krishna) a textbook "Real-Time Systems," McGraw Hill, 1997. In 1987, he received the Outstanding IEEE Transactions on Automatic Control Paper Award, and in 1989, Research Excellence Award from The University of Michigan. In 1985, he founded the Real-Time Computing Laboratory, where he and his colleagues are investigating various issues related to real-time and fault-tolerant computing.

His current research focuses on Quality of Service (QoS) sensitive computing and networking with emphases on timeliness and dependability. He has also been applying the basic research results to telecommunication and multimedia systems, intelligent transportation systems, embedded systems, and manufacturing applications.

He received the B.S. degree in Electronics Engineering from Seoul National University, Seoul, Korea in 1970, and both the M.S. and Ph.D degrees in Electrical Engineering from Cornell University, Ithaca, New York in 1976 and 1978, respectively.