

# Error-Injection-Based Failure Characterization of the IEEE 1394 Bus

D.J. Beauregard, Z. Kalbarczyk, R.K. Iyer  
Coordinated Science Laboratory  
Center for Reliable and High-Performance Computing  
University of Illinois at Urbana-Champaign  
{djbeau, kalbar, iyer}@crhc.uiuc.edu

S. Chau, L. Alkalai  
Jet Propulsion Laboratory  
National Aeronautics and Space Administration  
{savio.n.chau, Leon.Akalai}@jpl.nasa.gov

## Abstract

*This paper investigates the behavior of the IEEE 1394 bus in the presence of transient errors in the hardware layers of the protocol. Software-implemented error injection is used to introduce errors into the internals of the 1394 bus hardware chipset. Results from this study indicate that the IEEE 1394 bus protocol provides robust network communication in the presence of single-bit errors in the chipset.*

## 1 Introduction

The X2000 Future Deliveries Project at JPL-NASA aims to create a cost-effective yet reliable solution for spacecraft computer systems. The project focuses on using COTS components in an intelligent manner to support reliability and availability [3]. One of the tasks the project addresses is characterization of the fault tolerance, latency, bandwidth, and performance of IEEE 1394 bus architecture, an emerging standard in high-speed digital data transport for computers and for professional and consumer electronics products. The architecture has also been widely explored for use in highly dependable systems [2][7][8].

This paper presents results from the study exploring the behavior of the IEEE 1394 bus in the presence of transient errors in the hardware-implemented physical and link layers of the protocol stack. Software-implemented error injection is used to cost-effectively introduce errors into the internals of the 1394 bus hardware chipset, an approach that can produce similar fault coverage to hardware injection [4]. Most of the registers and message queues of the 1394 are memory mapped and hence can be accessed via software. Other recent studies of the 1394 bus include use of ion-induced injection [10] and emulation-based analysis [11].

Results from this study indicate that: (i) the IEEE 1394 bus protocol provides a robust network communication in the presence of single-bit errors in the chipset, (ii) no errors originating in a given node are observed to propagate and corrupt the operation of another node on the network, (iii) the vast majority of errors do not affect transmissions involving nodes other than the faulty one, (iv) the most serious recovery requires rebooting/resetting a single node, not the entire network, and (v) many of the errors can be detected by embedding data consistency checks (e.g., checksums on data and acknowledgment messages).

## 2 1394 Overview

The *IEEE 1394 Standard* is defined in terms of four layers: the physical layer (or PHY), the link layer, the

transaction layer, and the serial bus management layer. The PHY and parts of the link layer are implemented in hardware. Each has a set of software-accessible (i.e., memory mapped) registers. The transaction and serial bus management layers are defined completely at the software level. Nodes are physically connected point-to-point through cables in a tree structure, though any node can address any other node directly.

All communication on the 1394 bus is packet-based. Transactions can be classified as one of two types: (a) *asynchronous*, used when reliability is of primary importance, or (b) *isochronous*, used when bandwidth is of primary importance. An asynchronous transaction consists of one request packet and one response packet to contain data, and two one-byte acknowledgment packets to provide reliability. In contrast, an isochronous transaction consists of only one write request packet with no acknowledgment and no response. More details on the IEEE 1394 bus can be found in [1][5][6].

## 3 Target System Configuration

**Network.** The experimental testbed consists of PowerPC nodes connected via the IEEE 1394 bus. Each node on the network has 24 MB of RAM and no local permanent storage. The IEEE 1394 network is built using PCI interface cards based on the Texas Instruments 1394 chipset, with three available cable ports on each card. The system is also connected to a Solaris-based Ethernet network, which can be used to access remote permanent storage. The hardware used for this research was provided by JPL.

**Operating System.** Each node runs a version of the VxWorks real-time operating system. The address space is flat and globally accessible by all tasks.

**IEEE 1394 Driver.** The underlying link layer driver in the target system is the Mindready SedNet2 1394 Serial Bus Real-Time API. This is a C-based API supporting most of the link layer and PHY functionality. Two levels of API are provided. The high-level API supports the sending and receiving of most packet types and a limited set of control commands. Each received packet must be polled by a blocking function call. The low-level API provides interrupt callback functions for asynchronous events, in addition to support for acknowledgment packet reception and direct access to the PHY registers. All the applications in our experiments use only the low-level API. More information about the driver and API can be found in [9].

**Application Software.** In order to fully test the operation of the IEEE 1394 bus, a workload is added to generate

sufficient activity on the network. Each node is fitted with an application that sends a large file to nodes on the network. The receiving node writes the received data to an output file on the host storage for later verification. Because the 1394 protocol restricts packet size to 2 kB (when using the maximum speed of 400 Mb/s), the large files are split into smaller packets and sent sequentially.

To provide for a flexible, modular design, and to adhere to the 1394 standard, the *Object-Oriented Transaction Layer* (OOTL), a custom transaction layer framework underlying the application, is implemented. The framework provides a flexible transaction layer interface that allows adding logging functionality for many events. For example, all response packets that are received are reported, along with the tag number of the request packet to which they correspond.

**NFTAPE/VxwFI Error Injection.** Error injection is provided through *NFTAPE*, a configurable software framework for injecting faults and compiling results in an automated fashion [12]. *NFTAPE* is a distributed environment that executes on a TCP/IP network. In our experiments, the Control Host (which facilitates specification and control of fault injection experiments) executes on a Solaris host system, while the Process Manager (a communication gateway between the control host and the target nodes) executes on each of the embedded VxWorks/PowerPC systems. Figure 1 gives the network setup with the *NFTAPE* framework in place.

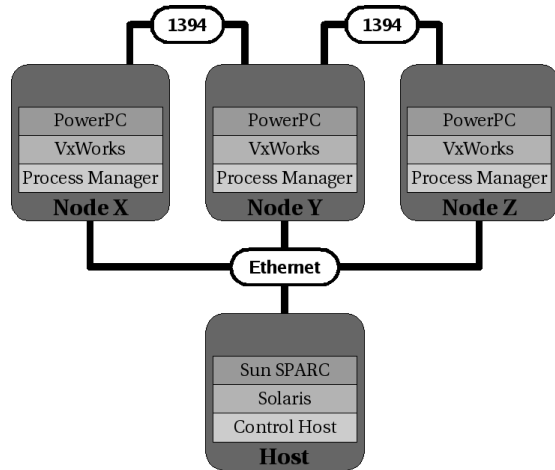


Figure 1: NFTAPE Network Setup

A lightweight fault injector, *VxwFI*, is developed for the target platform to facilitate single-bit and multiple-bit errors. In this study, *VxwFI* is used to target the chipset registers only. Recall that registers are memory mapped, hence the injection process does not cause any undesired side effects.

*NFTAPE*-driven error injection experiments are defined in terms of *campaigns*. Each campaign consists of a series of individual *runs*—repeated invocations of the application with identical inputs. Each run is injected with a different error.

There are about 4701 bits in the 1394 chipset that can be injected by software; read-only bits are protected by the hardware. Because of the time complexity involved (each

fault injection takes about three minutes, largely due to the high amount of logging information that must be sent to the Control Host), a subset of 332 of these bits are chosen for the fault injection campaign. This enables testing of more configurations than would otherwise be possible. The subset is generated from a careful analysis of the 1394 registers, grouping individual faults that would very likely have the same effect on the system and reducing each group to two or three sample faults. For instance, the *isoRecvIntEvent* register contains one bit for each of the 32 isochronous channels; therefore, it is possible to test only channels 0-3, assuming the other faults in the register would result in identical outcomes.

## 4 Experimental Procedure and Setup

All of the experiments follow the general procedure outlined in Table 1. *NFTAPE* also logs and monitors the received acknowledgment and response packets to facilitate more detailed analysis of errors that occur.

Table 2 provides details for four configurations (in terms of fault injection targets and communication patterns) used in the experiments. Figure 2 illustrates the communication and injection target for experiment 4; the other configurations follow similarly.

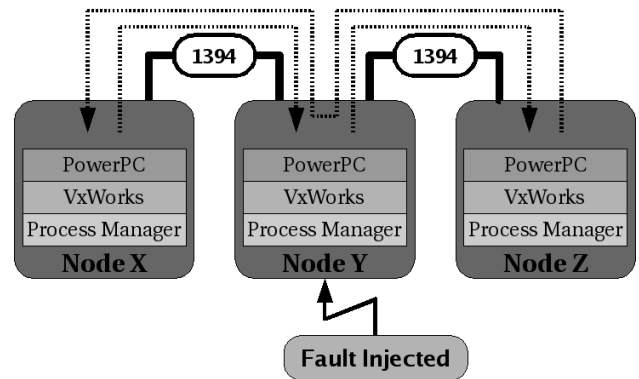


Figure 2: Setup for Experiment 4

For a given configuration, a fixed subset of nodes (termed the *active nodes*) are designated to transmit and receive data. Each active node sends a particular, unique file to another node. The file used in this study is a bitmap image (61-80 kB in size) in Portable Pixmap (PPM) format. PPM format is particularly well suited for visually displaying data corruption. Note that the IEEE 1394 bus does not discriminate against file types; corruption occurs on the level of raw data. The data flow between the active nodes is given in Table 2.

Since each node sends a unique file, we can positively identify the source of any data seen in the results of the experiment. Any mix-up involving the destinations for individual transmissions will be observable. The file sizes are large enough to keep the images somewhat recognizable, even when distorted, but small enough to keep the transmission time of each run reasonable.

Each file is broken up into multiple smaller-sized packets by the sender in order to fit within the 2kB packet limit. The

IEEE 1394 standard does not prescribe any special relationship of one packet to another, so we create a custom protocol specifying how a file is sent over the network in packets: Each file transmission begins with a *start packet* and ends with a *finish packet*. Both of these are 1394 request packet types. Each request packet (start, finish, or regular data) must be followed by one acknowledgment and one response packet from the receiver before the next request packet is sent.

Almost every action on the bus has a timeout associated with it to allow transactions to complete without stalling the campaign. Each acknowledgment packet must be received within 20 seconds, while a response packet must be received within 60 seconds. If a timeout expires, an attempt is made to clean up the system state, and the fault injection campaign continues.

**Table 1: General Procedure for Experiment**

Actions for an Individual Node
1. The bus is reset with the middle node as the root.
2. The nodes are informed of the physical IDs of the other nodes according to the mapping protocol provided in OOTL.
3. Each transmitting node sends data to a prescribed receiver.
4. The injector flips the bit in the target register of the target node.
5. Each transmitting node sends its data once again, as in Step 3.
6. The injector writes the original register value back into the target register of the target node.

**Table 2: Configuration for Each Experiment**

Experiment	Targeted Node	Data Flow
1	X	X to Y, Y to X
2	X	X to Z, Z to X
3	Y	X to Z, Z to X
4	Y	X to Y, Y to Z, Z to X

## 5 Fault Injection Results

Out of the 332 faults injected in the *experiment 1* scenario, only 16 errors surfaced. In *experiment 2 and 4* scenarios in which an active node was injected exhibited only 17 errors, including all the errors exhibited in the two-node scenario. The remaining scenario, experiment 3, exhibited only 2 errors. The injected faults that produced errors, as well as the errors themselves, are described in Table 3.

Each column in the *outcomes* section of Table 3 indicates the abnormal behavior that a fault causes. An **X** indicates that the abnormal behavior occurred in the presence of the corresponding fault. The outcome categories are as follows:

- A:** Missing output on fault node.
- B:** Missing output on nonfaulty node receiving data from faulty node.

- C:** Faulty node crashes and reboots.
- D:** Faulty node does not process some types of incoming packets correctly.
- E:** Nonfaulty node does not receive all packets from faulty node.
- F:** System cannot quickly recover from fault. (See Section 5.2)

An **X** in one of the last four columns indicates that the unusual behavior corresponding to the fault was present in the experiment for that column. Otherwise, the system behaved normally for that experiment in the presence of the corresponding fault.

### 5.1 Immediate Effects

Examining the immediate errors that occurred due to faults is useful in determining error detectability. From an end user’s perspective, errors are categorized as follows:

1. A node writes corrupt output.
2. A node crashes (causing an automatic reboot) and halts the experiment.
3. A node produces no output file.

Error categories 1 and 2 appeared in only a very few cases. A *corrupt output* (category 1) occurred only once: in the presence of the byte-swap bit fault. This is considered a fail-silence violation because the node is unable to detect the fault without looking at the output. The resulting corrupt PPM file is unreadable because of the erroneous header. Figure 3 demonstrates what happens when most of the data is corrupted in a Portable Greymap (PGM) file in a way that is analogous to the way the PPM was corrupted.<sup>1</sup> Note that this is all due to a single-bit fault in the register.

A *crashed node* error (category 2) occurred in four situations. All of these faults occurred in the IntEvent register, which caused some form of a false isochronous interrupt event. A crash can be easily detected by employing a software heartbeat or a timeout.

It is important to note that even if the node crashes, the 1394 chipset hardware will still function, since it does not rely on the CPU to do many of its autonomous tasks. This was explicitly tested for the faults in category 2. The node will continue to acknowledge packets addressed to itself and repeat incoming packets to all ports. This important feature of the IEEE 1394 bus normally allows all nonfailing nodes to communicate with each other, even if a node in the middle of the network has failed. Even when power is taken from the failed node, the 1394 chipset can usually take enough power from the cable to keep the rest of the network intact.

The *missing output* error (category 3) was by far the most common type, occurring in 12 fault injections. One possible way of detecting this condition would be to have a third party checking the output. It is also possible for the transaction layer on one of the nonfaulty nodes involved in transmission to detect some of these errors. If the nonfaulty node receives bad acknowledgments or does not receive

<sup>1</sup> The PGM format is a grayscale version of the PPM format. It is used here because the discoloration produced for the PPM format used in the experiments cannot be properly seen in black-and-white.

response packets, it could suspect an error. Otherwise, the transmission will look perfectly acceptable from the perspective of the nonfaulty nodes.

Five of the errors in Table 3 are not detectable by a nonfaulty node in its transaction layer because the faulty node either does not send its data or does not process its incoming data correctly. These include the *run* bit in the AR Response Context Control, the *ARRS* bit (also pertaining to asynchronous responses), the *noByteSwapData* bit, the *writeReg* bit (discussed in Section 5.2) and the *phyReqResourceX* bit.

## 5.2 Severity Analysis

The clearest categorization of faults can be obtained by assessing the severity of the faults. Table 3 indicates whether the system could quickly recover from a fault. A quick recovery is defined as removal of the fault (either because the fault is transient or the system corrects it),

followed by a 1394 bus reset, followed by a repeat transmission. For seven of the errors, the system could have quickly recovered. This type of fault is termed *less severe*. In the other cases, the recovery algorithm requires a laborious reinitialization of some kind. This type of fault is termed *more severe*.

The *less severe* faults were those that simply toggled a simple switch or set some particular threshold. An example of this was the *asyReqResourceAll* bit in the Asynchronous Request Filter register. When this bit is flipped from 1 to 0 in the fault injection, all requests outside of config ROM space are suddenly blocked. This fault causes the faulty node to send no acknowledgment and no response packets, preventing another node from sending its data. When the bit is flipped back to 1, transmission can proceed again as normal. This does not necessarily imply that recovery is complete; the requestor will still have to retransmit any data that was ignored by the faulty node.

**Table 3: Faults That Caused Observable Errors**

Targeted Register/Bits	Outcomes						Affected Exps.			
	A	B	C	D	E	F	1	2	3	4
AR Request Context Control – run bit	X			X	X	X	X	X		X
AR Response Context Control – run bit		X		X	X	X	X	X		X
Asynchronous Request Filter – asyReqResourceAll bit	X			X	X		X	X		X
Bus Options – max_rec field	X			X	X		X	X		X
HC Control – noByteSwapData bit	(1)	(1)					X	X		X
HC Control – softReset bit	X			X	X	X	X	X		X
IntEvent – ARRQ bit	X			X	X	X	X	X		X
IntEvent – ARRS bit		X		X	X	X	X	X		X
IntEvent – busReset, selfIDComplete bits	X			X	X		X	X		X
IntMask – masterIntEnable bit	X			X	X	X	X	X		X
IsoXmitIntEvent – isoXmit[0-3] bits	X	X	X	X	X	X	X	X	X	X
PHY Control – writeReg bit		X		X	X			X		
Physical Request Filter – phyReqResourceX bit	X			X		X	X	X		X
Port Status Page 0 (physical layer) – port disabled bit		X			(2)	X		X	X	

Notes: (1) The output on each node was corrupted. (2) The nonfaulty nodes cannot communicate with each other.



**Figure 3: The Original Image (top) and the Effect of a Single-bit Fault in One of the Link Layer Registers (bottom).**

The *more severe* faults were those that affected interrupts or made some drastic change to the system state. One example of this is the *softReset* bit in the HCControl register. Setting this single bit resets most of the registers to power-on values and flushes the packet FIFO's. The biggest problem with this error is that the driver (which is not "reset") is inconsistent with the state of the chipset.

One of the less severe faults proved to be an interesting special case. The target register was the PHY control register, which allows access to the physical layer registers via the link layer. The fault flipped the write register bit, which triggered a write to the gap count register in the physical layer, changing it from 63 to 0. In normal operation, the gap count must match on all nodes, and all nodes must agree on any change. The gap count plays a critical role in determining the idle time that each node must wait between packets before attempting to arbitrate for the bus. A node with a mismatched gap count can have an unfair advantage or disadvantage during bus arbitration. This fault can easily be removed with two 1394 bus resets. After two consecutive resets, the physical layer will reset these registers to 63 automatically.

## 6 Evaluation

A number of observations can be made based on these results:

1. *Most faults in an inactive node have no effect on the system.* This is demonstrated by experiment 3. We also demonstrated that a node on which the operating system has crashed can still forward transmissions intended for other nodes to the rest of the network. This means that recovery on the faulty node, if detected, can take place in parallel with other network activity.
2. *Only one error affected transmissions that did not involve the faulty node.* This particular fault disables a port on middle node of the system. Disabling this port caused the bus to be partitioned.
3. *Some of the errors that occurred persisted even after the fault was removed.* This is due to the fact that some bits irreversibly affect the system state. These situations were resolved simply by rebooting the faulty node. Other less conservative approaches could have worked, but the node initialization sequence is relatively complex.

In consideration of these observations, implementing a recovery system capable of rebooting the faulty node is advised. Our focus on small, embedded platforms allows reboot recovery as a viable option. The crucial feature of this mechanism would be that it is functionally independent of the health of the faulty node's operating environment. One example of this type of mechanism is an external watchdog.

The results demonstrate that nodes in an IEEE 1394 system should do the necessary checks for acknowledgment packets and response packets. These checks are normally seen by the transaction layer. Most of the faults observed in our experiments can be detected by observing the flow of acknowledgments and responses. Detection is only possible

if the faulty node is the source or destination of some transmission. In order to provide even better detection, one should also place checking mechanisms in the data itself to verify its integrity. One option is to include periodic checksums or tags in the output that can be used by software to verify the data. The checksum mechanism implemented in the 1394 protocol is not sufficient for this type of verification because the physical layer calculates it at the time of transmission, when the data may have already been corrupted.

## 7 Conclusions

This work gives ample evidence that the IEEE 1394 protocol provides a robust network of nodes in the presence of single-bit hardware errors. No faults are shown to propagate on the bus, and a large majority of faults do not produce any errors. Only one fault had any effect on transmissions involving nodes other than the faulty node. We also clearly demonstrated that any recovery includes, at most, a reboot of the faulty node. Detection of many of these errors is relatively straightforward, requiring standard packet consistency checks and error detection incorporated into the data (e.g., a checksum).

## 8 References

- [1] D. Anderson. *FireWire System Architecture*. Addison-Wesley, second edition, 1999.
- [2] G. Baltazar and G.P. Chapelle. *Firewire in Modern Integrated Military Avionics*. IEEE Aerospace and Electronics Systems Magazine, vol. 16, no. 11, Nov 2001, pp. 12-16.
- [3] S.N. Chau, L. Alkalai, A.T. Tai and J.B. Burt. *Design of a Fault-Tolerant COTS-Based Bus Architecture*. IEEE Transactions on Reliability, vol. 48, no. 4, Dec 1999, pp. 351-359.
- [4] E. Fuchs. *Validating the Fail-Silence Assumption of the MARS Architecture*. DCCA-6, Mar 1997.
- [5] IEEE. *IEEE Standard for a High Performance Serial Bus*, std 1394-1995 edition, 1995.
- [6] IEEE. *IEEE Standard for a High Performance Serial Bus - Amendment 1*, std 1394a-2000 edition, 2000.
- [7] J.R. Marshall. *Building Standards Based COTS Multiprocessor Computer Systems for Space Around a High Speed Serial Bus Network*. 17th AIAA/IEEE/SAE Digital Avionics Systems Conference, vol. 1, Nov 1998.
- [8] L. Ruiz, and P. Dallemagne. *An Application Layer for Using Firewire in Industrial Applications*. WFCS 2000, Sep 2000.
- [9] Sederta, Product Division of Mindready Solutions, Inc. *SedNet Version 2.x 1394 Serial Bus Real-Time API Reference Manual*, third edition, 2000.
- [10] C. Seidllick, S. Buchner, H.S. Kim, P.W. Marshall and K.A. LaBel. *Test Methodology for Characterizing the SEE Response of a Commercial IEEE 1394 Serial Bus (FireWire)*. IEEE Transactions of Nuclear Science, vol. 49, no. 6, Dec 2002, pp. 3129-3134.
- [11] D. Steinberg and Y. Birk. *An Empirical Analysis of the IEEE-1394 Serial Bus Protocol*. IEEE Micro, vol. 20, no. 1, Jan/Feb 2000, pp. 58-65.
- [12] D. Stott, B. Floering, D. Burke, Z. Kalbarczyk and R.K. Iyer. *NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors*. IPDS 2000, Mar 2000.