Low Overhead Fault Tolerant Networking in Myrinet*

Vijay Lakamraju, Israel Koren and C.M. Krishna

Department of Electrical and Computer Engineering
University of Massachusetts, Amherst MA 01003
E-mail: {vlakamra,koren,krishna}@ecs.umass.edu

Abstract

Emerging networking technologies have complex network interfaces that have renewed concerns about network reliability. In this paper, we present an effective low-overhead fault tolerance technique to recover from network interface failures, more particularly network processor hangs. We demonstrate the technique in the context of Myrinet. Fault recovery is achieved by restoring the state of the network interface using a small backup copy containing just the right amount of information required for complete recovery. Our fault detection is based on a software watchdog that detects network processor hangs. Results on the Myrinet platform show that the complete fault recovery can be achieved in under 2sec while incurring a latency overhead of just 1.5μs during normal operation. The paper also shows how this fault recovery can be made completely transparent to the user.

1 Introduction

The complexity of network hardware has increased tremendously over the past couple of decades. This is evident from the amount of silicon used in the core of network interface hardware. A typical dual-speed Ethernet controller uses around 10K gates whereas a more complex high-speed network processor such as the Intel IXP1200 [8] uses over 5 million transistors. This trend is being accentuated by the demand for greater network performance, and so communication-related processing is increasingly being offloaded to the network interface. Nowadays, interfaces with a network processor and large local memory are not uncommon. For example, the Myrinet host interface card uses a custom 32-bit RISC processor core and onboard SRAM ranging from 512K to 8M bytes.

[1]This work has been supported in part by NSF under Grant CCR-0234363.

Unfortunately, such increased complexity renews concerns regarding reliability and availability of the system. Network interface hardware is prone to the same type of failures as the host hardware. Network interface failures can however be more detrimental to the reliability of a distributed system. As we will see in the next section, faults can cause the network processor to hang, causing the node to be cut-off from the rest of the system. Not only that, faults can also cause the host computer to crash/hang, and worse still, can sometimes even affect a remote network interface. So, detecting and recovering from such network interface failures as quickly as possible is crucial for a system requiring high availability and reliability.

In this paper, we present an efficient, low-overhead fault-tolerance technique for network interface failures. The focus will be on a specific type of failure, namely network processor hangs. The central philosophy behind our technique is to keep around enough network-related state information in the host so that the state of the network interface can be correctly re-established in the case of a failure. Clearly, the challenge in such a scheme is to provide for this "checkpointing" with as little performance degradation as possible. Having the host checkpoint the state of the network interface (in the classical sense) can substantially degrade the performance of the system. In our technique, the "checkpointing" is a continuous process in which the applications make a copy of the required state information before sending the information to the network interface and update it when the network notifies the application that the state information is no longer required. As the results will show, such a scheme greatly reduces the impact on the normal performance of the system. Our technique also incorporates a quick fault detection scheme based on software implemented watchdog timers. We believe that both the fault detection and recovery techniques are general enough to be applicable to many modern network technologies, primarily those that are microprocessor-based.

Before we detail our fault tolerance technique in Sections 3 and 4, we will briefly describe Myrinet, which is the platform for this case study and report on fault injection experiments that expose the vulnerability of such microprocessor-based network systems to faults. In Section 5, we discuss some implementation details and report on the performance and overhead of the fault tolerance scheme. We conclude the paper in Section 6.

2 Myrinet: An Example System

Myrinet [3] is a cost-effective, high-bandwidth (2 Gb/s), packet-communication and switching technology from Myricom Inc [11]. It employs wormhole switching, backpressure flow control and source routing to achieve low-latency ($\sim 8\mu$s) transfer of messages. A Myrinet network consists of point-to-point, full-duplex links that connect Myrinet switches to Myrinet host interfaces and other Myrinet switches.

The Myrinet host interface card is designed to provide a flexible and high performance interface between a generic bus, such as the SBus or PCI, and a Myrinet link. Figure 1 shows the organization and location of the Myrinet host interface card in a typical architecture. At the center of the microarchitecture is a chip, called the *LANai*, which contains a RISC processor, (Direct Memory Access) DMA logic (packet interface) to/from the network, External (or E) bus interface logic to/from the host, timers, and local configuration registers. The fast local synchronous memory (SRAM) is used to store the Myrinet Control Program (MCP) and for packet buffering. The MCP is the program that runs on the LANai RISC processor and provides the basic functionality for reliably transfering messages from the host to the Myrinet link.

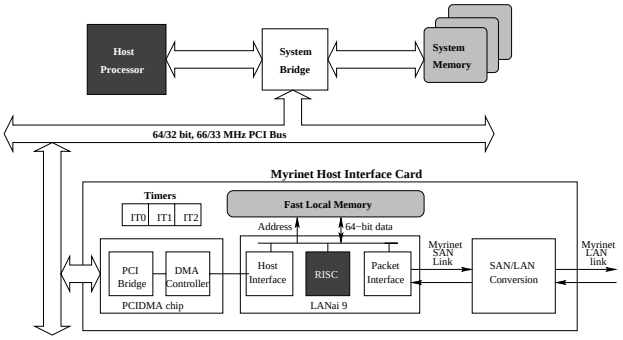

Figure 1. Myrinet host interface card

The Myrinet host software consists of a device driver that runs as part of the host operating system and a user library that provides a light-weight communication layer and API for the application software. The device driver provides important I/O device related interfaces to the user library, such as port opening and closing, memory mapping, interrupt handling and loading the MCP. The ability to change the network behavior through the MCP has made possible the implementation and testing of a number of communication protocols on Myrinet, including Active Messages [4], Fast Messages [12] and BIP [13]. Myricom's own software, called GM, borrows many features from these protocols and is currently the most widely used and preferred software for Myrinet. All these protocols are low-overhead protocols that avoid operating system intervention by providing a *zero-copy* mode of operation directly from the user space to the network (Figure 2). The scheme, however, requires that the user virtual memory on both the sending and the receiving side be pinned to an address in physical memory so that it will not be paged out during DMA carried out by the network interface. Thus, a user program employs system calls to allocate a number of unswappable pages of memory used for data-exchange, and thereafter avoids system calls.

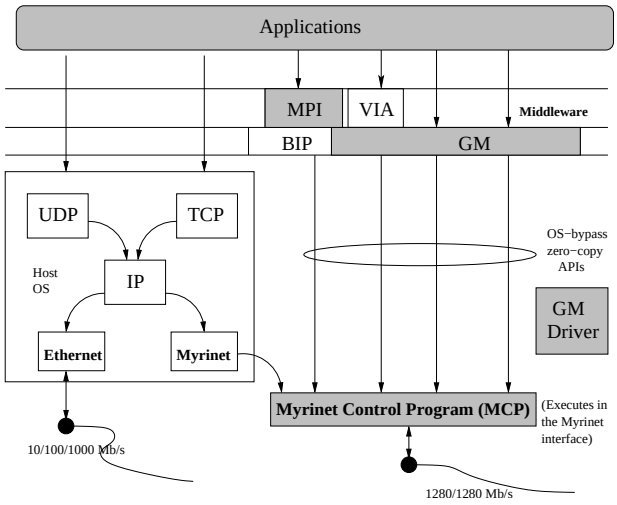

Figure 2. Myrinet software

Apart from the small latency, two other features make GM worth considering in distributed systems: reliable in-order delivery of messages and self-configuration. GM automatically handles transient network errors such as dropped, corrupted or misrouted packets. This handling is done transparent to the user and is mainly carried out in the MCP. For configuring the network, a program called the GM *mapper* is run on a node. The GM mapper initiates the mapping process and at the end of the mapping protocol, each interface

has a map of the network and routes to all other interfaces stored in its local memory. The GM mapper can also reconfigure the network if links or nodes appear or disappear. Inspite of these fault tolerance features, as we show next, Myrinet/GM may still not be favorable for systems requiring high availability for special applications, like the NASA REE supercomputer [6].

All of Myrinet's high availability features assume that the LANai processor executes error-free. This can be a very costly assumption to make in some applications, more particularly space applications. For example, a cosmic ray could cause an instruction in the MCP to flip a bit which could make it an invalid instruction. When the LANai executes the invalid instruction, it could simply crash. In some cases, the host application accessing the Myrinet card also hangs and hence it may be required to restart the application/machine.

Such were the type of effects seen in fault injection experiments performed by our research group as well as other research groups [15]. Transient faults in the network processor were simulated by flipping bits randomly in the code segment of the MCP. Rather than injecting faults into the entire LANai SRAM, one section of the MCP code, namely *send_chunk*, was selected and for each experiment, a fault was injected at a random bit location in this section while it was handling some network communication. Since *send_chunk* corresponds to a serial piece of code that is executed by the LANai each time a message is sent out, we are assured that all the faults are activated. Table 1 shows a summary of the results from these experiments and those reported in [15]. The network hardware (LANai 9) and software (GM 5.1) used in our experiments is the latest (as of this writing) and hence validates a more recent Myrinet technology.

Table 1. Results of fault injection on a Myrinet system (1000 runs)

Failure Category	% of Injections	
	Our work	Iyer *et al.*[15]
Local Interface Hung	28.6	23.4
Messages Corrupted	18.3	12.7
Remote Interface Hung	0.0	1.2
MCP Restart	0.0	3.1
Host Computer Crash	0.6	0.4
Other Errors	1.2	1.1
No Impact	51.3	58.1

It is clear from the table that interface hangs and dropped/corrupted messages account for more than 90% of the failures that affect the network interface in some undesirable way. Surely, these results could be different if fault injection is carried out on some other section of the code, but the results give a flavor of the different types of failures one can expect. A interface hang could mean that the LANai simply stopped executing instructions or that it has entered into an infinite loop , causing it to stop responding to user requests. While dropped/corrupted messages are already well handled by the GM software, there is no easy way to correctly recover from interface hangs. The driver could be reloaded and the application restarted from a safe checkpoint (if there is one). But, as we shall see in the next section, this does not always ensure correct recovery. Middleware, such as MPI, built on top of GM, consider GM send errors to be fatal and exit when they encounter such errors. This can cause a distributed application using MPI to come to a grinding halt if proper fault tolerance is not implemented.

The table also shows other types of failures, such as host computer crashes, which are caused by faults that propagate from the network interface to the host system. While we will not concern ourselves with such types of failures in this paper, the point of all this discussion is to show that faults affecting the network interface can have a substantial effect on the reliability of the system. For the rest of the paper, we will concentrate on interface hangs.

3 Recovery Strategy

Recovery from a host interface failure primarily involves restoring the state of the interface to what it was before the failure. Simply resetting the interface and reloading and restarting the MCP would not be sufficient as it can cause messages to be lost or duplicate messages to be received. In order to elucidate this aspect, let us take a closer look at the programming model and design of GM.

3.1 GM Programming Model

Communication between user processes in separate nodes using GM takes place through endpoints called "ports" with two non-preemptive priority levels for messages. The programming model is "connectionless" in that there is no need for the client software to establish a connection with a remote port in order to communicate with it. The sender simply allocates DMAable memory, initializes the memory segment and informs the LANai that the message needs to be sent out. The receiver, on the other hand, allocates DMAable memory and notifies the MCP that it is ready for receiving. The MCP is responsible for the rest of the communication process, i.e., DMAing the contents of the message from/to host memory, building the packet according

to the Myrinet specifications, setting up connections, sending and receiving the packet and ensuring reliable in-order delivery of messages. A "connection" corresponds to a logical link to a remote node which the MCP uses to multiplex all the traffic to that node. The MCP uses a version of the Go-Back-N protocol to handle transient network errors such as dropped, corrupted, or misrouted packets and ensure in-order delivery of packets over each connection. Finally, the MCP informs the user process of a message arrival or a successful send by posting an event in its event queue.

Flow control in GM is managed through a token system, similar to that used in credit-based flow control [10]. Both sends and receives are regulated by implicit tokens, which represent space allocated to the user process in various internal GM queues. A send token consists of information about the location, size and priority of the send buffer and the intended destination for the message. A receive token contains information about the receive buffer such as its size and the priority of the message that it can accept. A process starts out with a fixed number of send and receive tokens and relinquishes a send token each time it calls GM API's *gm_send()* function and a receive token with a call to *gm_provide_receive_buffer()*. A send token is implicitly passed back to the process when its callback function is called and a receive token is passed back when a message is received from the receive queue using the *gm_receive()* call. Apart from the notification of received messages and completion of sends, the receive queue is also used for other sundry purposes such as alarms. There are other GM internal events which a process is not expected to handle and can simply pass them to *gm_unknown()* which handles them in a default manner. This programming style allows maximum overlap between computation and asynchronous communication. Figure 3 shows the schematic of a typical control flow in a GM application.

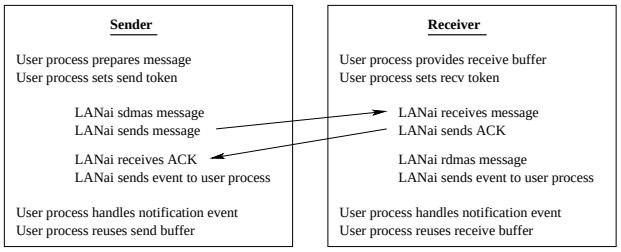

Figure 3. A typical control flow

3.1.1 Duplicate Messages

Reliable transmission in GM is achieved through the use of sequence numbers and these sequence numbers are maintained solely by the MCP and are therefore transparent to the user. If the MCP is simply reloaded and restarted on failure, the state of the connections and the sequence numbers are lost. This loss of information does not allow the messages to be retransmitted reliably. Consider the example shown in Figure 4. A sending node crashes when an ACK is in transit. After recovering from the failure, since all state information is lost, the sender may try to resend the message with a invalid sequence number. The receiver would reply by sending a NACK with the expected sequence number. At this point, if the sender resends the messages with this sequence number, the receiver would incorrectly accept a duplicate message.

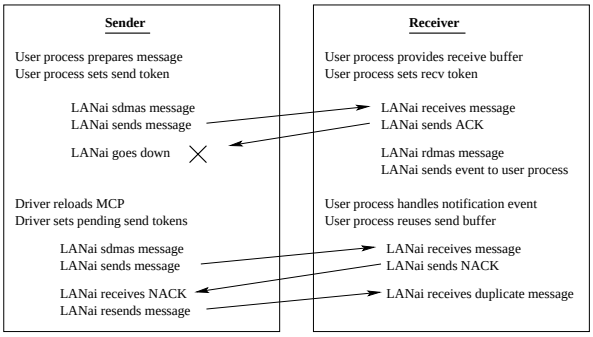

Figure 4. The case of duplicate messages

This problem arises due to the lack of redundant state information. If information of all streams of sequence numbers was stored in some stable storage, then the MCP could use this redundant information during recovery to send out messages with the correct sequence number and avoid the problem of duplicate messages. The key, however, is to manage redundancy so that the performance of the network is not impacted greatly.

3.1.2 Lost Messages

The GM programming model is "connectionless" in that the sender does not explicitly set up a connection with the receiver. So, if the faulty node is a receiver, then there is not much state information that needs to be restored. The receiver in GM sends out an ACK as soon as it receives a valid message. This can lead to faulty behavior as shown in Figure 5. Consider the case when the LANai crashes after the send of the ACK is complete but before the entire message has

been DMAed to the host memory. This can happen if the host DMA interface is not free and so, the DMA is delayed. The receiver will never receive that message again because as far as the sender is concerned, it received the ACK for the message and notified the application that the send was successful. The sender would not resend the message and so, as far as the receiver is concerned, the message is lost forever.

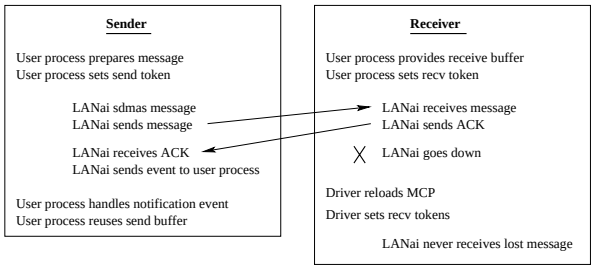

Figure 5. The case of lost messages

This problem arises because of the lack of a proper commit point for a send-receive transaction. The receiver should send out an ACK only when the message has been copied to its final destination.

4 Putting it Together

The above discussion indicates that reloading the MCP alone does not guarantee correct recovery. What is required is to restore the state of the network interface to a point that guarantees the correct handling of future messages as well as messages in flight at the time of failure. A crude way to achieve this is by periodically "checkpointing" both the application and the network interface state and retracting back to the last checkpoint in the case of a network failure. Such a scheme however involves a great deal of overhead and in many ways can work against the very basis of using a high-speed network. Clearly, storing the entire state of the network interface is an overkill. What is important is to keep a copy of just the right amount of state information required for complete recovery. For example, the case of duplicate messages can be dealt with by simply keeping a copy of the sequence number of the message that was last acknowledged. The challenge, however, is in recognizing what is required and what can be left out. The design of our fault tolerance scheme was driven by two key objectives: (i) introducing minimal copy overhead and (ii) maintaining the same application programmers interface (API) so that minimal or no changes to application source code would be required.

4.1 Setting the stage

Since we are considering only network interface failures, a safe place for storing the required network interface state is the host's memory. Apart from sequence numbers, it is also important to keep a copy of the send and receive tokens. As we had discusses earlier, a process implicitly relinquishes a send token (and passes it to the LANai) when a call to a GM *send* is made and gets it back when the send is successfully complete. A send token consists of information about the location, size and priority of the send buffer and the intended destination for the message. It is important to keep an updated copy of all the send tokens that are in possession of the LANai so that this information can be used during fault recovery to resend the messages that have yet to be acknowledged. Similar is the case with the receive tokens. Keeping a copy of the forfeited receive tokens allows us to notify the LANai of all the pinned-down DMA regions that have not yet been filled by the LANai.

In our implementation, extra space is allocated by the user process to maintain a copy of the send token queue and the receive token queue. When a call to any of the *gm_send()* functions is made, a copy of the send token is added to the queue. Since the size of a token is small, the overhead from a simple memory-copy is quite insignificant. The process also stores a copy of the receive token when it provides receive buffers. Apart from this information, the host also needs to have a copy of the sequence numbers used for each connection. This is easily achieved by having the user process generate the sequence number and pass it through the send token to the LANai. The MCP now simply uses these sequence numbers rather than generating its own. If messages are to be assigned sequence numbers strictly on a per-connection basis to maintain the original GM protocol, all the processes on a node sending messages to the same remote node need to be synchronized so that a continuous stream of sequence numbers for the connection is obtained. Such a synchronization can however introduce unnecessary overhead. A simple solution to this is to generate independent streams of sequence numbers for each remote node on a per-port basis. This generation can be done entirely within a single process but requires that the receiver now acknowledge on a per-port basis rather than simply on a per-connection basis. Thus, the receiver now has to keep an ACK number for every (connection,port) pair. The extra memory requirement is however not large since GM allows only 8 ports per node. This is the main deviation from the original GM structure, as depicted in Figure 6.

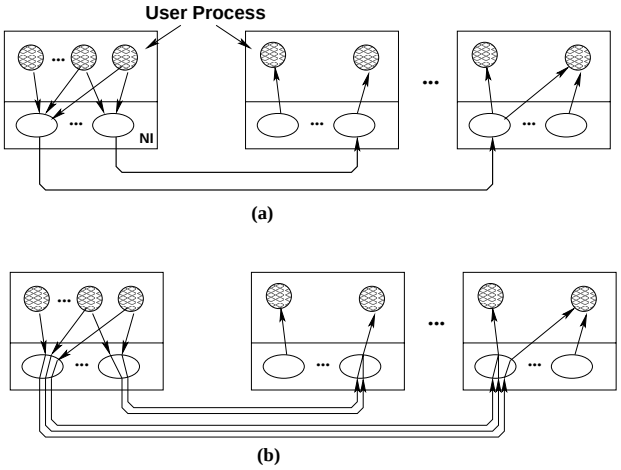

Figure 6. (a) All streams are multiplexed into a single connection (b) Independent streams per connection

Another difference with the original GM is with regard to the commit point on the receiver side. In our implementation, we delay the sending of an ACK to after the DMA of the message into the user's receiver buffer is complete. This increases the network occupancy of the message but, as the results in Section 5 show, the impact on performance is not at all significant. Since the receiver must also keep a copy of the ACK number for every stream, the LANai needs to notify the host of the sequence number of the message that has just been ACKed. This it does, by including the sequence number as part of the event posted by the LANai into the user process's receive queue. The receiver, at this time, also deletes the corresponding copy of the receive token. Similarly, on the sender side, the copy of the send token is removed just before the callback function for that send token is invoked. Note that all these changes can be implemented within the GM library functions, thus making them transparent to the user. There are other pieces of the state information that need to be stored by the host, such as the priorities and sizes of messages acceptable by the receiver, but we will not focus on these, as they are not critical to the understanding of our scheme. Thus, in a nutshell, the user keeps a copy of the required LANai state that is not implicitly stored in the host memory. All this sets the stage for the actual recovery part.

4.2 Fault Detection

Traditionally, fault detection has been achieved through heartbeats or "I am alive" messages. The Myrinet switch uses such heartbeat signals to determine if a host interface on the other side of the link is powered down. Having the LANai DMA such a message or interrupt the host processor periodically to notify it of its health, can be very expensive. Another scheme would be to have the host processor poll the network interface card periodically, but this would require context switches and, depending on the polling frequency, can again impose a significant performance overhead. One way to relieve the host processor of such a chore, would be to associate a monitor (by way of a watchdog processor, for example) with the LANai processor. This however requires extra hardware which may not be present in network interfaces. Our fault detection scheme is based on a fairly simple watchdog, but one implemented in software and using the low-granularity interval timers present in most interfaces. It takes advantage of the structure of the MCP.

The MCP is basically an event-driven program. It executes a fixed (set of) action(s) when a set of events occur and some conditions are satisfied. For example, a *Send DMA* routine is started when the following conditions are true: a proper send has been posted, the host DMA interface is free and the LANai has sufficient free space for a DMA operation into its local SRAM. Similarly, a timer routine is called when an interval timer expires. The LANai has three interval timers. These are 32-bit counters that are decremented every $1/2\mu$s. In GM's MCP, only one of the three interval timers (IT0) is used. When this timer expires, the corresponding bit in the interface status register (ISR) is set and in the next dispatch cycle, a timer routine, called *L_timer()* is called. The host uses this routine to notify the LANai of various user actions, such as opening and closing a port, request for pausing the LANai as well as setting alarms. At the end of the *L_timer()* routine, ITO is reset, causing it to expire again after a predetermined interval. Numerous experiments with our Myrinet network revealed that the maximum time between these timer routine invocations during normal operation is around 800μs. Note that this value is not equal to the predetermined interval as the MCP event handling is serialized.

We use one of the remaining interval timers for our fault detection purposes. One of the spare interval timers (say IT1) is first initialized to a value just slightly greater than 800μs. The *L_timer()* routine is modified to reset IT1 whenever it is called. The interrupt mask register (IMR) provided by the Myrinet HIC is modified to raise an interrupt when IT1 expires. So, during normal operation, *L_timer()* resets IT1 just in time to avoid an interrupt from being raised. When the LANai crashes/hangs due to a fault, the *L_timer()*

routine is not executed, causing IT1 to expire and an interrupt is raised, signaling to the host that something may be wrong with the network interface. Such a scheme allows the host to detect host interface failures with virtually no overhead and so the performance of the network is not compromised.

This detection technique assumes that a network interface hang does not affect the timer or the interrupt logic. While this assumption cannot be proved to be correct, our experimental results show that this is most often the case. In fact, this simple fault detection mechanism was able to detect all the interface hangs reported in Table 1.

4.3 Fault Recovery

The interrupt caused by the expiration of the spare interval timer is the first indication that something might be wrong with the host interface card. This FATAL interrupt is handled by GM's host device driver. While one might wish to carry out the entire fault recovery process in the interrupt handler, it is not always possible. Functions such as *sleep()* and *malloc()* which need to be called during our fault recovery process cannot be called in an interrupt handler because the interrupt handler does not run in the context of a process. We use a daemon process instead. We call this the fault tolerance deamon (FTD).

The FTD can be run anytime before fault recovery is to be achieved. The daemon, in its simplest sense, opens up a port and waits for a fault to occur. The device driver, on receiving the FATAL interrupt, simply wakes up the FTD which then starts the fault recovery procedure. First, the FTD checks to make sure that the LANai has indeed failed. This it does by writing a "magic" word to a location in the LANai SRAM which would have been cleared by the LANai had it been functioning correctly. If the location is not cleared, the FTD assumes that the interface has hung. The FTD then disables the interrupts, unmaps the IO and reset the interface card. It is assumed that the fault causing the upset is transient and that a card reset will cause all the components on the card to reset to a non-faulty state. The FTD then clears the LANai SRAM and reloads the MCP. The DMA engine is restarted and the interrupts re-enabled. This brings the state of the LANai to the same state it reaches when the GM driver is loaded. The next step would be to restore the LANai state using the copy stored in the host. Firstly, the LANai is notified of the page hash table maintained by the host. The page hash table keeps track of the mappings of virtual addresses for each port to DMA addresses. It is big, so it is stored in host memory and the MCP caches entries into the LANai SRAM. Next, the FTD restores the mapping and routing table information in the LANai. Finally, the FTD posts a FAULT_DETECTED event in the receive queue of all open ports, before rewinding and standing guard for the recovery of the next fault.

4.4 Transparency

The asynchronous nature of communication in GM requires a user process to occasionally poll the receive queue for new events. As indicated earlier, all GM internal events are required to be passed to a special GM library function called *gm_unknown()*. Transparency of fault recovery is achieved by modifying this function to handle the FAULT_DETECTED event. A series of actions are carried out in the handler to get the process back on track. After some initial cursory checks, the LANai send and receive token queue is restored using the process' backup copy. Note that the send tokens contain the sequence numbers of the messages that have not been acknowledged, while the receive tokens correspond to the host buffers that have not received messages. The process then updates the LANai with the last sequence number received on each stream, one for each (connection,port) pair. This ensures that the LANai ACKs the right messages and NACKs those that arrive out-of-order. Finally, the process clears its receive queue before notifying the LANai to "reopen" the port. The LANai initializes the per-port state and, as usual, starts sending and receiving messages for the port.

5 Implementation and Performance Results

It is important to see how our design requires no changes to be made to previously-written GM applications to take advantage of the fault tolerance features. We incorporated our fault recovery scheme into a recent version of the GM software (GM-1.5.1). We will refer to this modified software as FTGM. We were able to implement all the required changes into GM library functions making the failure of the network interface completely transparent to the applications, or middleware software, such as MPI. What is required however is for the application to be recompiled with the new GM library.

An important design criteria during implementation was to use resources sparingly so that the performance is not impacted greatly. Hash tables were used wherever searching over large arrays was required. The extra static memory usage in the LANai was around

$100KB$ while a process used up extra virtual memory in the order of $20KB$. It is important to evaluate the impact of the implemented fault tolerance scheme on the performance of the network. We present this analysis next.

5.1 Performance Impact

The performance of a network system is usually measured using three principal metrics:

- **Bandwidth** is typically equated to the sustained data rate available for large messages.
- **Latency** is usually calculated as the time to transmit small messages from source to destination.
- **Host-CPU utilization** measures the overhead borne by the host-CPU in sending or receiving a message.

The LogP model [5] specifies similar type of metrics but for this analysis we will use the ones stated above.

GM provides a set of programs that can be used to evaluate these metrics. Our experimental setup consisted of two Pentium III machines each having 256MB of memory, a 33MHz PCI bus and running RedHat Linux 7.2. The Myrinet host interface cards were LANai9-based PCI64B cards and the Myrinet switch was type M3M-SW8. Figure 7 compares the bandwidths obtained with GM and FTGM for different message lengths. The workload for these experiments involved both the hosts sending and receiving messages at the maximum rate possible (as in *gm_allsize*). For each message length, a large number of messages (we used 1000) were sent repeatedly and results averaged. For small message sizes, the data-rate performance is limited by the number of DMA transfers and packets the interface can handle per unit time. Longer messages convey bytes in larger units, hence, can use up the bandwidth provided by the links more efficiently. The figure shows that the sustained bidirectional data rate for GM as well as FTGM approaches an asymptotic value of $\sim 92MB/s$ for long messages. FTGM follows very close on the heels of GM and for all practical purposes, imposes no appreciable performance degradation with regards to bandwidth.

The reason for the jagged pattern in the middle of the curve is because GM fragments large messages into packets of at most $4KB$ at the sender and reassembles them into messages at the receiver. This fragmentation and reassembly is performed in order to limit the packet size in the network, so that a long message will not block a channel for an extended period, but will allow other packets to be interleaved on the channel.

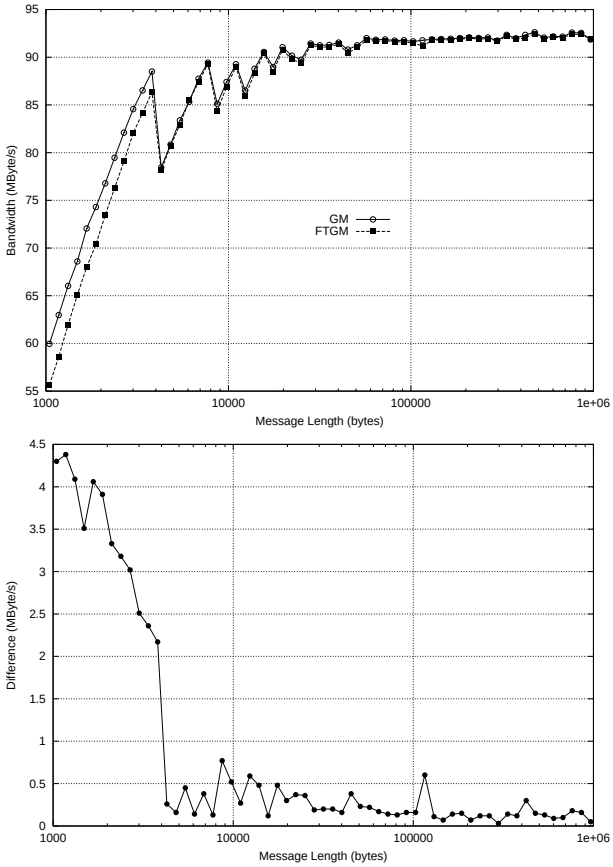

Figure 7. Bandwidth comparison of the original GM and FTGM

Figure 8 compares the point-to-point half round trip latency of messages of different lengths. The measurement was performed as a repetitive "ping-pong" exchange of messages between processes in the two machines, with the one-way latency for each message length plotted as half of the average round-trip time.

Here again, the performance of FTGM is not far behind the original GM. The short-message latency, a critical metric for many distributed-computing applications, is about $11.5\mu s$ for GM and $13.0\mu s$ for FTGM, averaged over message lengths ranging from 1 byte to 100 bytes. These latencies are the sum of a host component and a network interface component. While the host component is a combination of the host-CPU execution time and the PCI latency, the network interface component is a combination of the LANai execution time and the packet interface latency. FTGM was designed to minimize the amount of extra information being DMAed from the host memory to the LANai memory. Moreover, there is absolutely no change in the packet header and no extra information is sent with the

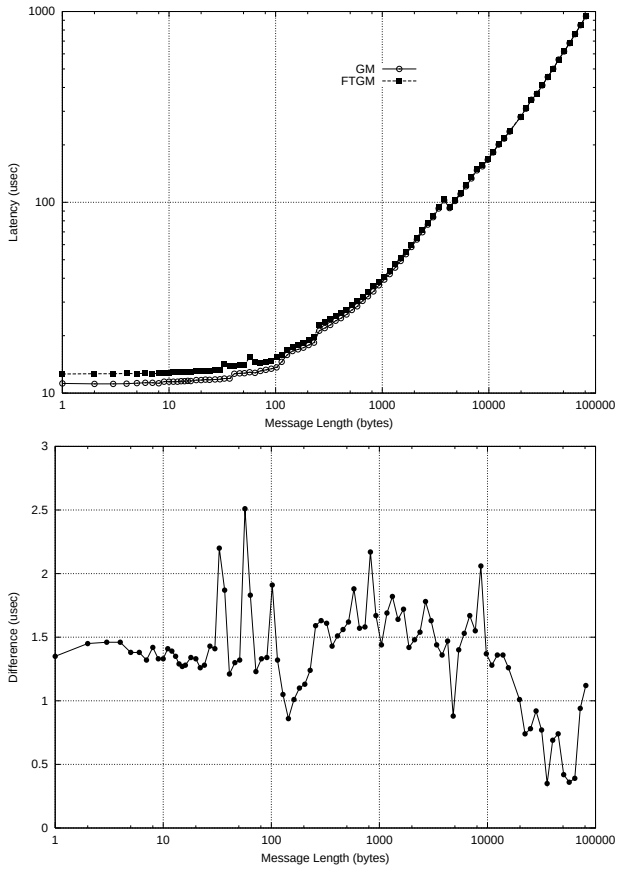

Figure 8. Latency comparison of the original GM and FTGM

packet. Therefore, the effect on the PCI latency and the packet interface latency in the LANai would be minimal, if at all. The modification in the MCP that affects the critical path the most is the delaying of sending the ACK to after the DMA is complete. Since the ACK needs to be delayed only when a receive token is returned to the user, a multiple-packet message can be made to take full advantage of the network bandwidth by not waiting for the DMA to be complete, thus allowing several packets of the same message to be in-flight at the same time. For small messages, however, the extra delay comes mainly from the host-CPU utilization. This factor is most predominant in protocols employing a host-level credit scheme for flow control [2], such as FM. Minimizing the host-CPU utilization was one of our principal design objectives. Information posted on the Myricom website indicates that the measured overhead on the host for sending a message is about $0.3\mu s$ and for receiving a message is $0.75\mu s$. In FTGM, the send and receive token housekeeping contributes the greatest to the increase in delay. It is around $0.25\mu s$ for the send and around $0.4\mu s$ for the receive. The extra overhead for the receive is because the receiver has to update two hash tables for every receive: the hash table containing the recv tokens and the hash table containing ACK numbers for each stream. Table 2 summarizes the results presented in this section.

Table 2. Comparison of various performance metrics between GM and FTGM

Performance Metric	GM	FTGM
Bandwidth	$92.4MB/s$	$92.0MB/s$
Latency	$11.5\mu s$	$13.0\mu s$
Host util. (send)	$0.30\mu s$	$0.55\mu s$
Host util. (recv)	$0.75\mu s$	$1.15\mu s$
LANai util.	$6.0\mu s$	$6.8\mu s$

5.2 Recovery Time and Effectiveness

The complete recovery time is the sum of the fault detection time and the times spent in the FTD and the user process' fault handler for restoring the state, as shown in Figure 9. The fault detection time was measured as the time from the fault injection to then time when the FTD is woken up by the driver. It is a function of the maximum time in between *L_timer()* invocations and the interrupt latency. We shall ignore the interrupt latency, because it is negligible ($\sim 13\mu$s) compared to 800μs for the watchdog timer interval. The FTD recovery time consists of time required to reload the MCP and restore routing and page hash tables and posting the FAULT_DETECTED event in each open port's receive queue. Averaging over a number of experiments revealed a value of $\sim 765000\mu$s for the FTD recovery time, with $\sim 500000\mu$s being spent in reloading the MCP.

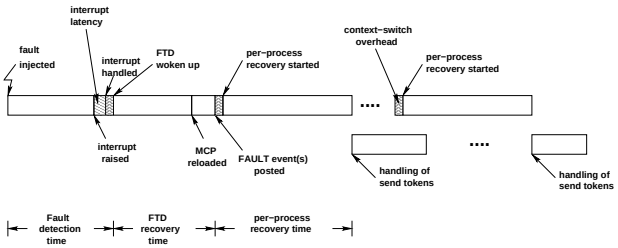

Figure 9. The timeline of the fault recovery process

The rest of the recovery time depends on the number of open ports at the time of failure. The per-port recovery time is primarily a function of the execution time

of the FAULT_DETECTED event handler. Our experimental results show that this value is $\sim 900,000\mu$s. It is arguable whether the time for handling the restored send tokens by the LANai needs to be accounted for in the recovery time. This would however be a function of the number of send tokens that have been restored. Table 3 gives a breakdown of the complete fault recovery time.

Table 3. Components of the fault recovery time

Component	Value(μs)
Fault Detection Time	800
FTD Recovery Time	765000
Per-process Recovery Time	900000

The experiments reported in Table 1 were repeated using FTGM. While all the network interface hangs were correctly detected, there was only five cases out of the 286 hangs that FTGM was not able to properly recover from. We are currently investigating these cases.

6 Conclusions

This paper describes a low-overhead network interface failure recovery scheme for Myrinet. The crux of the scheme is to keep a copy of just the right amount of network interface state information in the host so that the state of the network interface can be restored on failure. Furthermore, we show how this can be achieved by keeping the fault tolerance completely transparent to the user. Implementation results are very promising. Fault detection can be achieved in less than a millisecond, whereas the complete fault recovery typically takes less than 2 seconds. Such a quick recovery can be obtained with just a 1.5μs overhead in normal operation. Depending on the application, this small overhead could be well worth paying for, considering the high availability that can be obtained.

The basic idea of such type of fault recovery is quite generic and can be applied to almost all user-space communication protocols [1]. Most of these high-speed communication protocols use some kind of token system for flow control. As we have seen, maintaining a copy of the outstanding tokens will not incur a substantial overhead. Thus, all these protocols can stand to gain from such a scheme, when it comes to fault tolerance. Our fault detection and recovery scheme also takes advantage of the autonomy between the host processor and network processor. This is a feature present in Myrinet and many modern microprocessor-based network interfaces, such as Infiniband [9], Gigabit Ethernet [16, 14], QsNet [17] and ATM [7].

References

[1] S. Araki, A. Bilas, C. Dubnicki, J. Edler, K. Konishi, and J. Philbin. User-space communication: A quantitative study. In *Supercomputing'98*, Nov. 1998.

[2] R. Bhoedjang, T. Rühl, and H. Bal. User-level network interface protocols. *Computer*, 31(11):53–60, Nov. 1998.

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. K. Su. Myrinet — A gigabit-per-second local-area-network. *IEEE Micro*, 15(1):29–36, Feb 1995.

[4] D. Culler, S. Goldstein, K. Schauser, and T. Eicken. Active messages: A mechanism for intgerated communication and computation. Technical Report UCB/CSD 92/675, University of California, Berkeley, Mar 1992.

[5] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, July 1993.

[6] R. Ferraro. NASA Remote Exploration and Experimentation Project. http://www-ree.jpl.nasa.gov/.

[7] A. T. M. Forum. *ATM User-Network Interface Specification*. Prentice Hall, 1998.

[8] T. Halfhill. Intel network processor targets routers. *Microprocessor Report*, 13-12, Sept. 1999.

[9] Infiniband Trade Association . http://www.infinibandta.com/.

[10] H. T. Kung and R. Morris. Credit-based flow control for ATM networks. *IEEE Network*, 9(2):40–48, Mar. 1995.

[11] Myricom Inc. http://www.myri.com/.

[12] S. Pakin, V. Karamcheti, and A. A. Chien. Fast messages: Efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency: Parallel Distributed & Mobile Computing*, 5(2):60–73, Apr-Jun 1997.

[13] L. Prylli and B. Tourancheau. BIP: A new protocol designed for high performance networking on myrinet. *Lecture Notes in Computer Science*, 1388:472–480, 1998.

[14] P. Shivam, P. Wyckoff, and D. Panda. EMP: Zero-copy OS-bypass NIC-driven Gigabit Ethernet message passing. In *SC2001: High Performance Networking and Computing*, 2001.

[15] D. T. Stott, M.-C. Hsueh, G. L. Ries, and R. K. Iyer. Dependability analysis of a commercial high-speed network. In *Proceedings of The Twenty-Seventh Annual International Symposium on Fault-Tolerant Computing (FTCS'97)*, pages 248–257, June 1997.

[16] The Gigabit Ethernet Alliance . http://www.gigabit-ethernet.com/.

[17] The QsNet High Performance Interconnect . http://www.quadrics.com/.