Proceedings of the
IEEE International Conference on Cluster Computing
Chicago, Illinois, pp. 266-274, September 2002.

Design and Validation of Portable Communication Infrastructure for Fault-Tolerant Cluster Middleware†

Ming Li, Wenchao Tao, Daniel Goldberg, Israel Hsu, Yuval Tamir
Concurrent Systems Laboratory
UCLA Computer Science Department
{mli,wenchao,dangold,israel,tamir}@cs.ucla.edu

Abstract

We describe the communication infrastructure (CI) for our fault-tolerant cluster middleware, which is optimized for two classes of communication: for the applications and for the cluster management middleware. This CI was designed for portability and for efficient operation on top of modern user-level message passing mechanisms. We present a functional fault model for the CI and show how platform-specific faults map to this fault model. Based on this fault model, we have developed a fault injection scheme that is integrated with the CI and is thus portable across different communication technologies. We have used fault injection to validate and evaluate the implementation of the CI itself as well as the cluster management middleware in the presence of communication faults.

1. Introduction

In most clusters each node runs a local copy of an off-the-shelf operating system that was not designed to manage a distributed cluster. Cluster Management Middleware (CMM) runs above the operating system and provides resource allocation, scheduling, coordination of fault tolerance actions, and coordination of the interaction between the cluster and external devices. High-performance reliable communication is critical for efficient execution of applications on the cluster as well as the operation of the CMM.

In many clusters, communication is implemented on top of TCP and Ethernet. Other clusters, especially those designed for parallel computation, use more efficient user-level message passing mechanisms [1]. In either case, the Communication Infrastructure (CI) must support interprocess communication within application tasks as well as the communication for the CMM. It is highly desirable to implement the CI in a way that facilitates efficient porting of the cluster middleware to different communication technologies.

The UCLA Fault-Tolerant Cluster Testbed (FTCT) project is focused on developing and evaluating algorithms and implementation techniques for fault tolerant cluster managers [8, 4]. A ''thin'' communication layer (CL) is a critical component of the CI for FTCT. This CL provides an interface for the rest of the middleware that need not change when porting the system to different underlying communication platforms. This is done while adding minimal additional overhead. While all communication is processed through CL, we argue for an unconventional layering structure that results in different processing for application messages and CMM messages. For both types of messages, the CI provides end-to-end reliable communication regardless of the reliability characteristics of the underlying platform [9].

Fault tolerance is a key requirement from many clusters [6, 10]. This is a focus of our work, which involves the use of active replication for the CMM in order to provide uninterrupted management for the cluster in the presence of faults [8, 4]. The operation of the cluster with respect to the applications as well as the CMM is dependent on the operation of the CI in the presence of faults. Validating the operation of the CI requires the ability to inject faults in the CI and evaluate the response of the system. We have developed a software-implemented fault injector for our CI that is portable across different platforms that may host the CI. A key to achieving this portability is a *functional* fault model that covers a wide variety of platform-specific faults that may be encountered. We have used this fault injector to validate the protocols used by CI and the entire CMM as well as to measure the performance of the system in the presence of communication faults.

An overview of our Fault-Tolerant Cluster Testbed (FTCT) is presented in Section 2. Section 3 describes the Communication Infrastructure (CI), including the key parts of its API. The functional fault model for the CI is presented in Section 4. In this section we also show how the faults that occur in commonly-utilized communication platforms map to the functional fault model. The implementation of the fault injector is described in Section 5. Our experience with the use of the fault injector to identify software bugs and evaluate the performance of our system is presented in Section 6.

† This research was supported by the Remote Exploration and Experimentation (REE) program of the Jet Propulsion Laboratory.

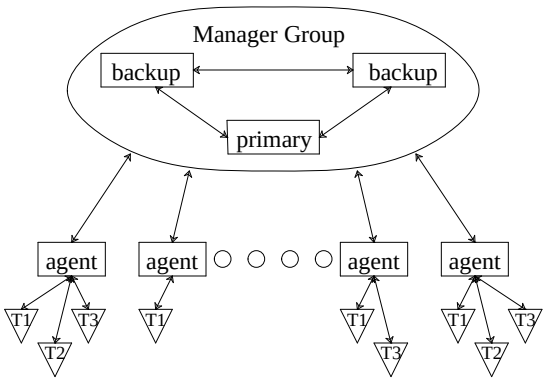

Figure 1. System structure of FTCT

2. Overview of FTCT

The overall structure of the FTCT system is shown in Figure 1 [8, 4]. The system consists of three components: a manager group, an agent process on each node, and a library for user applications. The manager group consists of three active replicas and performs cluster-level decision making. An agent process on each node sends node status information to the manager group, performs commands at the node on behalf of the manager group, and provides an interface between application processes and the CMM. A library that is linked with every application process is an important part of the CMM, providing part of the interface between the application process and the local agent. This library also provides the mechanisms needed to set up intra-task communication using MPI.

CMM communication (as opposed to application communication) is based on *authenticated* (signed[7]) messages. Errors in the CMM are detected using heartbeats, comparison of messages generated by the manager replicas, and standard reliable communication mechanisms [8, 4]. Under certain conditions, members of the manager group enter ''self diagnosis,'' where their states are compared and, possibly, a faulty replica is identified and terminated. Once a faulty manager replica is identified, a new manager replica is restarted using the states of the remaining two manager replicas.

3. The Communication Infrastructure

The idea of using a communication layer to facilitate porting is not new. For example, Glunix [3] provides such a layer. However, the Glunix portability layer is tuned for an implementation over TCP/IP and leads to inefficiencies when implemented over state-of-the-art communication platforms.

High-performance communication requires avoiding system calls and eliminating local message copies [11]. Hence, modern designs of NIC (network interface card) hardware allow user-level send and receive primitives, allow the NIC and host to access shared memory, and expose buffer management to the application [13]. The design of the FTCT CI was done with these considerations in mind — if the underlying communication platform provides these high performance features, our CI must be able to take advantage of them. Since the networking hardware available to us was Myrinet [1], our low-level CL was inspired by Myricom's GM library and has a similar, though not identical, ''look-and-feel.'' However, CL can be easily ported to other user-level communication platforms as well as more traditional platforms, such as TCP over Ethernet. We have working implementations on top of Myricom's GM as well as on top of UDP and sockets.

CL provides connectionless communication with no reliability guarantees. Basically, CL provides a way to get bits (most of the time) from one node to another with minimal overhead above the overhead of the underlying platform. CL uses (node id, port number) pairs to identify communication endpoints. In order to be able to efficiently utilize modern user-level message-passing mechanisms, the CL API exposes message buffer management to the application. The CL_send() operation is nonblocking and a *status* handle provided by the caller contains the status of the operation. The message to be sent *must* be in memory allocated using the CL API. CL calls can fail in a variety of ways. Table 1 summarizes the most important error conditions that may be flagged as a result of a call to a CL procedure.

The cluster must support the requirements of two classes of communication: application-level communication and CMM communication. Both classes require reliable communication. In general, the CMM uses aggressive fault tolerance mechanisms to ensure that the cluster as a whole remains usable. On the other hand, in our application domain, the failure of an application task is not quite as critical. Accordingly, we defined a Management Message Layer (MML) for CMM communication and a platform for application-level communication, called Reliable CL (RCL).

In order to meet its reliability requirement, MML provides authentication as well as the standard integrity checks. In addition, it must facilitate reliable atomic multicast required for maintaining active replication. Some of the CMM communication is of the form request-reply, or request-acknowledge. For example, the manager group directs an agent to start a process and expects a confirmation that the process has been started within some timeout. For this style of communication, end-to-end arguments lead to the conclusion that low-level (message layer) acknowledgements and retransmissions are not necessary. Regardless of the communication subsystem, the sender must be ready to take corrective action if it does not receive a high-level

Table 1. Key error conditions flagged by calls to CL procedures

Error Code	Description
LOCAL_RESOURCE	Required resources on the local node for completing the operation are currently not available
REMOTE_RESOURCE	Required resources on the remote node for completing the operation are currently not available (applies to send)
INVALID_BUFF	A buffer address passed to CL is invalid
INVALID_ENDPOINT	A specified communication endpoint (node id, port number) is invalid
INVALID_ARGS	Invalid arguments (not covered by the above entries) are passed to CL
DEST_REJECT	Message rejected by destination
UNREACHABLE	Unreachable destination (network problem, closed port at destination, etc)
NOT_INITIALIZED	Attempt to use an uninitialized communication layer
INTR	An asynchronous signal occurred during the CL call.

confirmation that the requested operation has been performed.

The considerations above lead to an MML that is quite complex due to all the functionality it must provide. For example, MML actually supports three types of messages: UNR — unreliable message where no acknowledgements are required, ACK — reliable message that requires a positive acknowledgement, and NACK — reliable message that does not require positive acknowledgement (negative acknowledgements used to flag lost messages). Heartbeats use UNR messages while the request-confirm scenario above uses NACK messages. MML is implemented strictly above CL, using only the CL API for communication. MML provides an API that is used by the management middleware (CMM).

RCL is a simple, conventional, reliable communication layer. Since high performance is a key requirement from RCL, we have chosen not to implement RCL (as we did MML) strictly above CL. In some cases, such strict layering would lead to inefficiencies due to lack of access to the internal workings of CL by the reliable communication routines. For example, with a Myrinet GM backend and strict layering above CL, RCL would have to perform periodic probing of the status of control messages, such as ACKs, in order to determine when this data can be garbage collected. Our implementation of RCL is integrated with the implementation of CL. Hence, the reliable communication routines have full access to the internals of CL. In the example above, this allows RCL to garbage-collect the status data structures and free the control packet's buffer in the callback routines that are called by GM without any probing.

The structure of the FTCT CI is shown in Figure 2. The MML is tightly-coupled with the CMM and strictly layered on top of CL. RCL functionality is integrated with CL and provides a high-performance reliable communication platform for implementing application-level communication. A retargeted version of MPICH provides the MPI API for the application.

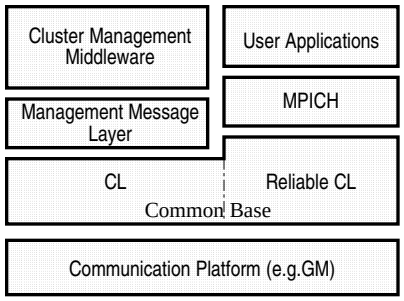

Figure 2. The structure of the FTCT Communication Infrastructure

4. A functional fault model for the FTCT CI

As mentioned earlier, our goal was to implement a portable fault injector for the communication infrastructure and use it to validate the protocols used in FTCT and evaluate their performance in the presence of communication errors. The basis of a portable fault injector must be a ''portable fault model'' that does not depend on the specific characteristics of the underlying communication platform. Since we have a communication infrastructure (CI) that wraps the communication platform with a standard interface, there is an opportunity to use this interface to develop a functional fault model that can meet the portability requirements. In Subsection 4.1 we define this functional fault model. In Subsection 4.2 we show how the specific faults of the Myrinet/GM communication platform maps to our functional fault model.

4.1. Defining a functional fault model

Software-implemented fault injection involves adding to a working system special software that perturbs the normal operation of the system, emulating the results of faults. Software-implemented fault injectors have been used by many systems to emulate communication faults [2, 5, 12]. Typically, the emulated faults are at a relatively high functional level, such as dropping, modifying, duplicating, and delaying messages. In this

subsection we define such a fault model that goes one step further, to include faults that cannot be covered by a strictly message-oriented high level model.

Our functional fault model includes the following types of faults: *message faults*, *invocation faults*, and *operation faults*. Message faults are faults that affect particular messages, including dropping, delaying, modifying, reordering, and injecting messages. Invocation faults are faults that happen (or are manifested) at the time of CL API invocation. Specifically, the CL invocation fails to perform its function and/or returns one of the error status codes described in Table 1. Operation faults are faults that occur during the actual transmission operation — the CL API invocation proceeds normally but there is an error indication later that is not a simple change of one message. For example, a message is sent but is never delivered due to full buffers on the receiver side.

All faults can be specified with a probability distribution of the fault recurrence interval. For message delaying faults, the duration of the delay can also be specified with a probability distribution. Message injecting faults include sending a previously-sent message to an arbitrary destination. The combination of message injecting faults and messages modifying faults also covers the more general case of sending an arbitrary message to an arbitrary destination.

Communication errors can also result in resource ‘‘leakage’’ that can have consequences that are not covered by the message-oriented faults. Specifically, resources like low level memory buffers and tokens used for flow-control can be leaked due to various communication faults. With the CL API, such resource leakage is eventually manifested at the functional level as send failure, receive failure, or allocation failure at the time of CL API invocation. Hence, these faults are covered by the *invocation faults* mentioned above. Invocation faults also include faults that cause CL API call failure due to parameter validation errors (e.g., a message buffer address that is not accessible by the network interface hardware).

With the CL API, the *send* call specifies a *status* variable where the result of the operation (success or failure) is eventually stored. This status mechanism can be used to emulate the faults that occur during the send operation but are not covered by the message faults. For example, with GM/Myrinet the *status* is set to failure if the receiver is out of buffer space for receiving the message and causes the sender to timeout. A list of the functional faults are summarized in Table 2. Each fault type is associated with parameters such as the probability distribution of the fault’s recurrence interval.

Table 2. Functional fault model for FTCT CI

Fault Type	Description
Drop	Message is dropped
Delay	Message is delayed
Modify	Message is modified
Reorder	Message is reordered
Inject	A previously sent message is sent to an arbitrary destination.
Invocation	Invocation faults — often due to lack of resources or bad parameters
Operation	Operation faults — *send* operation properly invoked but fails to complete correctly (e.g., receiver is out of buffer space)

4.2. Mapping communication platform faults to the functional fault model

The proposed functional fault model is quite general and provides good ‘‘coverage’’ of the errors that can occur with a wide variety of back-ends to the CL API. We do not attempt to prove this assertion but simply show the mapping for the GM/Myrinet platform. Mapping for the sockets/UDP platform has also been done but is omitted here due to space limitations.

GM over Myrinet provides user-level message-passing — no system calls are involved in the send or receive operations. Tokens are used for flow control, limiting the number of outstanding messages from any sender. On the receive side, tokens are used to regulate the number of active receive buffers. Some of the host memory is shared with the Network Interface Card (NIC). In order for a message to be sent, it must be placed in this shared space. The application ‘‘provides’’ the NIC with buffers in the shared space to be used for receiving messages. After a message is received and the data is read by the application, the buffer space is ‘‘provided back’’ to the NIC for future use.

The GM API for Myrinet is event-driven. Events are ‘‘returned’’ as a result of the application invoking gm_receive(). For example, to send a message, the application invokes gm_send_with_callback(). At some later point, when the application invokes gm_receive(), a *sent_tokens_event* is returned, causing a callback routing passed by the application to be invoked and reclaiming the send token. The event *recv_event* is returned when a message is received. Upon the receipt of this event, the application can extract the message from the receive buffer.

The operation of GM can lead to complex fault scenarios. For example, consider the consequences of a dropped message. If the message is dropped on the remote (receiver) NIC, the callback for the send may still be invoked as usual, the message is then marked as successfully sent, and a send token is reclaimed. If the message is dropped at the sender NIC, the callback for the

Table 3. Mapping of GM/Myrinet faults to functional faults

Communication Platform API	Communication Platform Failure Mode	Functional Fault
gm_send_with_callback	Send completes but doesn't call callback. Depletion of send tokens.	Eventual CL_send failure — eventual invocation/operation fault.
	Doesn't send and doesn't call callback. Depletion of send tokens.	Eventual CL_send failure — invocation fault, Drop
	Doesn't send and calls callback	Drop
	Send completes and calls callback but error on remote side. Includes no available buffers, port closed, port unreachable, etc.	Operation fault.
	Send wrong data, length	Modify
	Send to wrong location	Inject,Drop
gm_alloc_send_token	No tokens available	CL_send failure — invocation
gm_receive	Doesn't return message event	Drop, eventual invocation fault (buffer not provided back)
	Returns incorrect message event	Inject, eventual invocation fault (wrong buffer provided back)
	Doesn't return sent tokens event. Depletion of send tokens.	Eventual invocation fault
gm_provide_receive_buffer	Doesn't provide buffer	Eventual invocation fault
gm_unknown	Fails to call callback (or with correct arguments)	Eventual invocation and/or operation fault
gm_dma_malloc	No available memory	Invocation fault
gm_init, gm_open, gm_get_node_id, etc. (called at init)	GM failure	CL_init fails — invocation

send is never invoked, leading to the loss of a token. The possible send errors consist of all the combinations of sending or not sending the message and having the callback eventually invoked or never invoked. In addition, messages can be sent to the wrong location or with the wrong data. Sends can also fail because of a resource error — running out of tokens.

Table 3 shows the mapping of possible GM/Myrinet platform faults to our functional fault model. As a result of a fault, events that are supposed to be received may never arrive or may arrive with incorrect values. If a *recv_event* is lost, the results are a dropped message and a receive buffer that is not provided back to GM. If an incorrect *recv_event* is generated, the result may be the injection of a bogus message to the host. This could also cause the CI implementation to provide the wrong buffer back to GM, resulting in fewer buffers of a particular size being available. With GM, when a message send operation is completed, the appropriate callback function should be invoked after a *sent_tokens_event* is received by gm_receive(). If the *sent_tokens_event* is lost, the send buffer is never released for reuse and, eventually, the sender runs out of buffers for sending messages. If corrupt information is returned with the *sent_tokens_event*, resources may not be properly reclaimed and/or the *status* of the send may be incorrect (*operation* functional faults).

Based on the discussion above, the more complex faults in GM/Myrinet are likely to result in resource leakage that is eventually manifested as invocation faults or operation faults. For example, if a node runs out of buffer space, that may be manifested as an invocation fault when trying to allocate a new buffer for a send. Running out of buffer space may also be manifested and as an operation fault for a remote node trying to send to the node with no buffer space.

Some possible faults can have unexpected consequences, such as causing the host system to hang or crash. For example, it is undefined to send a message over GM/Myrinet if there is no available send token. The GM API requires the application using GM to ensure that a token exists before calling gm_send_with_callback(). However, the application's accounting of the available tokens could be incorrect. For example, the *sent_tokens_event* could be replicated, thus causing the application (in this case, the CI implementation) token management routine to operate as though there are more tokens than there actually are. Hence, a gm_send_with_callback() may be called even though there really are no tokens available. This could hang the host. Faults like this that can hang or crash the host are not considered in our fault model or validation.

5. A portable fault injector for the FTCT CI

The *message* functional faults can be implemented on either the send or receive side. The *invocation* faults must be implemented locally — on the node where the fault is being emulated. These faults are easy to emulate

by simply not performing the requested operation and returning an error code. The *operation* faults can be emulated on the sender side by returning incorrect information in the *status* variable passed to the send operation. In general, the implementation of the fault injector was guided by two key goals: ease of porting to different communication platforms and minimal disruption to the normal operation of the CI.

Table 4. Fault injector implementation

Fault Type	General Implementation
Drop	Do not call underlying send; set the status for this send as successful.
Inject	Call the underlying send with a copy of a previously sent message.
Delay	Hold on to the received message and do not deliver. Each time into CL_receive, deliver the delayed message if one exists before calling underlying receive.
Modify	Randomly modify bits in the buffer returned from underlying receive.
Reorder	Hold on to the received message and call the underlying receive again. Deliver the new message before delivering the message being held.
Invocation	Return with a resource error condition without calling the underlying communication function (send/receive/memory allocate)
Operation	Set status with random error. Possibly drop message.

Table 4 summarizes how the functional faults are implemented. The functions for fault injection exist at the CL interface, just above the calls to the communication platform's send and receive functions and just below the reliability features of RCL. The fault injection code is separated from the code for CL implementation.

It is straightforward to implement message dropping on the sender side by simply not sending the message. Dropping a message on the receiver side is more complex — with a user-level network protocol, such as GM, messages must be received and discarded and buffers must then be released back to the network layer. If messages are dropped on the receive side, they still use resources (receive buffers) and potentially cause side effects due to resource exhaustion. Such side effects are emulated separately as invocation and operation faults in our functional fault model.

Message injection is also simpler to implement on the sender side. When injecting a message, a sender creates a new message and sends it to a random destination. The content of the message is copied from a previously sent message.

Message delaying and reordering are implemented on the receive side. Implementing message delaying on the send side would involve holding a message in order to send it at a later time while reporting it as a successful send to the application. However, the fault injection mechanism must then ensure that the message is actually sent successfully at the later time. If, at this later time, there are problems completing the delayed send (due to a lack of resources, for example), the application cannot be alerted since the delayed message has already been reported as sent. In general, it is difficult to guarantee the completion of a send operation for all possible CL implementations. This problem is avoided on the receive side, where messages can be easily reordered or delayed.

In order to avoid the complexity of dealing with timers, we implemented a restricted version of message delaying and reordering. Specifically, on the receive side, when a message is selected for delaying or reordering, it is buffered, the underlying receive is called again, and nothing is delivered to the application until the next message arrives. For delaying, the first message is then delivered to the application. For reordering, the second message is delivered to the application first.

Message modifying is implemented by changing a random bit in a message on the receive side. Sender-based emulation of message corruption during transmission would require making a copy of the original message — if the original copy of the message is modified on the sender, the faulty message would be repeatedly retransmitted.

The fault injector implementation is not completely portable. For each underlying platform it is likely that some platform-specific code needs to be developed. For example, with the GM/Myrinet platform, the code needs to deal with callbacks and token allocation. If the fault being emulated is a message drop at the receive GM endpoint, the send actually completes successfully so the callback for the send operation must be invoked. With our fault injector, for a dropped message the underlying send is never called. Hence, the fault injection code itself has to perform the function of the callback function. Specifically, it must set the status of that send to indicate completion. Similarly, GM requires that a token be allocated for each send. The token is later freed by the callback function. When emulating message dropping, since the underlying send is not invoked, the token allocation is also skipped. Thus the normal token allocation is not disrupted. Despite the need for such platform-specific code in the fault injector, the majority of the fault injection code is portable.

Reliable CL (RCL) is *not* strictly layered on top of CL — the implementation of RCL procedures interacts directly with the underlying communication platform's API, not the CL API. Hence, implementing fault injection for CL does not automatically provide fault injection for RCL. However, both CL and RCL procedures interact with the platform API in a similar

way. For both CL and RCL procedures, the faults occur at the point of interaction with the platform API. For example, the message drop is implemented by essentially skipping over the call to the underlying send. Hence, the same fault injection code used to ‘‘instrument’’ the CL implementation can be used with essentially no modifications in RCL.

6. Experimental results

This section describes early experience with the use of the fault injector. It has been used for debugging the FTCT system as well as for evaluating the system’s performance in the presence of faults.

6.1. Identifying software bugs

Using random inputs is a well-known technique for testing the robustness of software and identifying problems dealing with unexpected inputs. Fault injection in the CI has been extremely helpful for improving the CI itself as well as the cluster management middleware (CMM) as a whole. To illustrate the usefulness of the fault injection for debugging, we present, as examples, two bugs that were uncovered using the fault injector.

In our original implementation of RCL, we used the message length in the message header to determine how many bytes should be included in the checksum calculation used to validate the message. When fault injection corrupted this message length field, the checksum routine attempted to access protected addresses, leading to a segmentation fault. The obvious fix was to use the *minimum* of the message length field in the message and the length of the buffer delivered by the communication layer.

Our cluster management middleware is based on a triplicated central manager. The manager replicas monitor each other by exchanging periodic heartbeats. Since heartbeats are transmitted as ‘‘unreliable’’ messages, some heartbeats are lost in transmission. This led to a situation where heartbeats from one manager replica reached a second replica but failed to reach the third replica. This situation ‘‘emulated’’ a non-fail-stop failure mode of the first replica and uncovered a flaw in our original implementation of the middleware.

6.2. The performance of Reliable CL

Application communication is implemented on top of Reliable CL (RCL). Hence, the performance of RCL is critical. In this section we report on the performance of RCL both without faults and with varying fault rate. All the measurements were conducted on a cluster whose nodes are dual CPU PCs with 350 MHz Pentium II processors running Solaris 8. The network is Myrinet with Lanai 4.3 processors (M2F-PCI32c Lanai cards). Note that this hardware is quite old so the results are not competitive with current hardware.

Table 5 shows the one way latencies for five different ping-pong micro-benchmarks: three that use GM alone, one that uses CL, and one that uses reliable CL. The first GM program is a program provided by Myricom to showcase their software. The second GM program is written by us and does not use any buffer management (it provides back to GM the same buffer that the message arrives in). The third GM program includes standard GM buffer management so that it can handle more than one message at a time. The overhead of RCL increases with increasing message size due to the time it takes to compute the checksum.

Table 5. One way message latencies (μsec)

API/ Benchmark	Message Size (bytes)			
	4	64	1024	4096
GM (Myricom)	25.1	26.56	54.78	120.7
GM (No BM)	26.1	27.75	55.1	122.8
GM	27.5	29.35	58	128
CL	26.5	28.6	56.5	127
RCL	30.9	32.16	72	170

Comparing RCL to CL, the RCL overhead includes extra processing at the sender and receiver as well as the handling of a periodic alarm interrupt used to check for timeouts. However, our measurement shows the overhead of handling of periodic timer events is not significant. The processing overhead on the sender side includes mainly extra header processing (approximately 150 clock cycles for an eight byte message). In addition, a timer event must be set up (for timeout) but that is done after the underlying send is invoked so that the send and timer setup are done mostly in parallel. At the receiver, extra processing includes checking the checksum, checking whether the message is in order, setting a timer event (for a delayed ack) and updating data structures (approximately 370 clock cycles for an eight byte message).

Table 6. Communication throughput (MB/sec)

API/ Benchmark	Message Size (bytes)			
	4	64	1024	4096
GM (Myricom)	0.16	2.44	30.73	73
GM (No BM)	0.151	2.33	30.62	72.63
GM	0.15	2.27	29.32	71.08
CL	0.147	2.28	30.27	72.46
RCL	0.12	1.94	25.3	67

Table 6 shows the communication throughput of micro-benchmarks designed to maximize throughput. For large messages, the throughput overhead of RCL is lower than the latency overhead since much of the processing overhead for a message can be done in parallel with the transmission of other messages.

To measure the performance of RCL in the presence of faults, two sample tasks were used. The first task is communication-bound and consists of a sender and a receiver. The sender sends as quickly as it can and the receiver receives as quickly as it can. The second task is CPU bound and also consists of two processes. Each process performs 12 sends, computes for enough time to allow for acknowledgments to arrive, and performs 12 receives. For each task, results were obtained based on five runs for each fault type using different random number seeds in the fault injector. The execution times of the CPU-bound task runs were all within 5% of the average. The communication-bound task had up to 15% variation in the execution times among runs. The execution time of the communication-bound task is highly dependent on the types of faults that occur and on the types of messages involved. This accounts for the larger variability in execution times between successive runs. The ability to evaluate the efficiency of RCL under different types of faults is important since it allows RCL to be fine-tuned to deal with the faults expected to occur in the system.

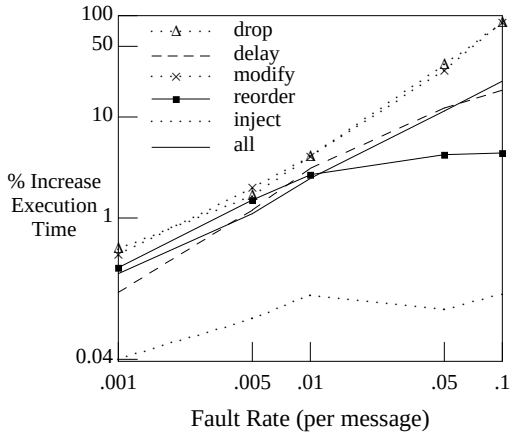

Figure 3. The effect of faults on the execution time of a communication-bound task.

Figure 3 shows the impact of faults on the performance of the communication-bound task. As expected, when faults increase the effective communication latency, the result is a dramatic increase in the execution time of the communication-bound task. Inject faults have minimal impact since they only require low overhead for eliminating duplicate messages or discarding messages that arrive at the wrong destinations. Figure 4 shows the impact of faults on the performance of the CPU-bound task. In this case retransmissions can occur in parallel with the computation, so the extra overhead in the presence of faults is small.

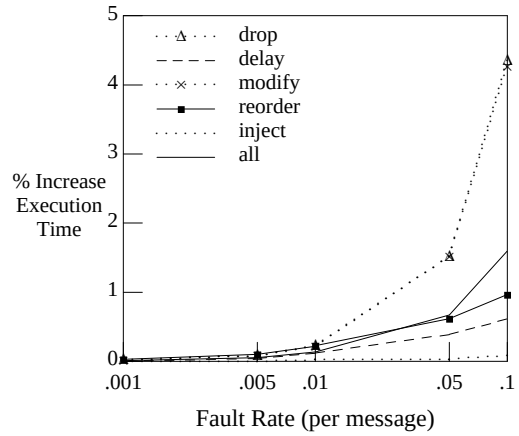

Figure 4. The effect of faults on the execution time of a CPU-bound task.

6.3. Fault injection with the CMM

As part of the evaluation and validation of the FTCT Cluster Management Middleware, we used the CI fault injector to inject faults in MML. As discussed earlier, MML messages include both reliable messages and unreliable messages (mainly heartbeats). If a fault affects a reliable message transmission or the corresponding acknowledgement, the sender may have to retransmit the same message and the receiver may receive the message more than once. Communication faults may also cause false alarms regarding node failures if the manager group misses two consecutive heartbeats from the agent on a node, thus invoking the node probing procedure to check the health of the node. It is also possible that communication faults cause the managers to initiate the manager self-diagnosis procedure. This occurs when two consecutive heartbeat messages from a manager replica are lost or delayed, or an agent gets inconsistent or belated messages from the managers [8, 4].

Figure 5 shows the consequences of communication faults under different fault injection rates. In each case, the number of consequences is normalized to the total number of faults injected. The normalized numbers of message retransmissions and duplicated messages are constant under different fault rates, indicating that the number of these events increases proportionally with the number of faults injected. However, with increasing fault rate there are smaller increases in the rate of node failure false alarms and the rate of managers entering self-diagnosis. The reason for this is that the CMM protocols are designed to mask (tolerate) some faults without resorting to high-level recovery or diagnosis actions. For example, an alarm regarding the possible failure of a node is raised only if there are *two consecutive* missing or delayed heartbeat messages.

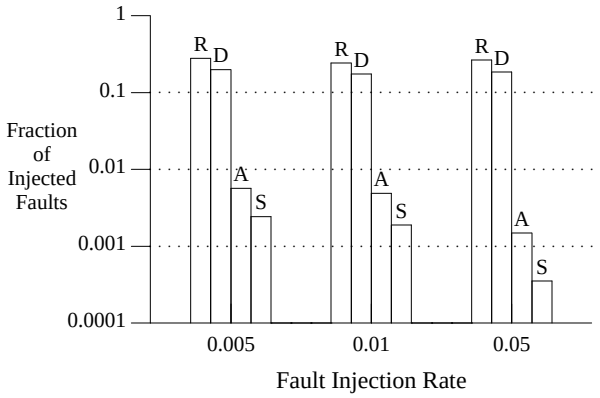

Figure 5. The effects of fault injection with different rates. (R) Retransmissions; (D) Duplicated messages; (A) False alarms on node failure; (S) Manager self-diagnosis.

7. Conclusion

Robust operation in the presence of communication errors is key to the reliability of any distributed system. We have designed and implemented a portable reliable Communication Infrastructure (CI) for MPI applications as well as a message layer optimized specifically for a cluster manager. The cluster manager's message layer reduces performance penalties by not repeating at lower levels that which needs to be done at the higher level anyway. Fault injection experiments are critical for validating fault tolerance protocols and evaluating performance in the presence of faults. In order to implement a fault injector that is largely portable to different communication platforms, we have developed a functional fault model. While previous models covered *message* faults, such as dropped messages, our model includes *invocation* and *operation* faults, providing better coverage of faults that can occur in modern high-performance message-passing mechanisms. Based on this functional fault model, we have implemented a portable fault injection mechanism that is placed in a common abstract communication layer used by applications as well as the cluster manager. We used this fault injector to debug and validate the operation of our cluster middleware. Preliminary fault injection experiments provide data on the impact of fault rates on overhead and performance. Our evaluation of the CMM under faults demonstrates the complexity of the relationship between fault rates and error rates in such a system. Future work will involve enhancing the ability of the fault injector to emulate more complex fault scenarios and evaluating the operation of our cluster middleware under different workloads and fault patterns.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, ‘‘Myrinet: A Gigabit-per-Second Local Area Network,’’ *IEEE Micro*, vol.15, no.1, pp. 29-36, February 1995.

[2] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung, ‘‘Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection,’’ *Proceedings of the International Symposium on Fault-Tolerant Computing*, Sendai, Japan, pp. 404-414, June 1996.

[3] D. P. Ghormley, D. Petrou, S. H. Rodrigues, A. M. Vahdat, and T. E. Anderson, ‘‘GLUnix: A Global Layer Unix for a Network of Workstations,’’ *Software - Practice and Experience*, vol.28, no.9, pp. 929-961, July 1998.

[4] D. Goldberg, M. Li, W. Tao, and Y. Tamir, ‘‘The Design and Implementation of a Fault-Tolerant Cluster Manager,’’, Computer Science Department Technical Report CSD-010040, University of California, Los Angeles, CA, October 2001. (Presented at the IEEE Cluster 2001 Conference, October 2001, Newport Beach, CA).

[5] S. Han, K. G. Shin, and H. A. Rosenberg, ‘‘DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems,’’ *Proceedings of International Computer Performance and Dependability Symposium*, Erlangen, Germany, pp. 204-213, April 1995.

[6] Z. Kalbarczyk, R. K.Iyer, S. Bagchi, and K. Whisnant, ‘‘Chameleon: A Software Infrastructure for Adaptive Fault Tolerance,’’ *IEEE Transactions on Parallel and Distributed Systems*, vol.10, no.6, pp. 560-579, June 1999.

[7] L. Lamport, R. Shostak, and M. Pease, ‘‘Byzantine Generals Problem,’’ *ACM Transactions on Programming Languages and Systems*, vol.4, no.3, pp. 382-401, July 1982.

[8] M. Li, D. Goldberg, W. Tao, and Y. Tamir, ‘‘Fault-Tolerant Cluster Management for Reliable High-Performance Computing,’’ *Proceedings of International Conference on Parallel and Distributed Computing and Systems*, Anaheim, CA, pp. 480-485, August 2001.

[9] R. Minnich, ‘‘Bipolar Disorder in Cluster Networking,’’ *Proceedings of the IEEE Cluster 2001 Conference*, Newport Beach, CA, October 2001. http://www.cacr.caltech.edu/cluster2001/program/talks/minnich.pdf

[10] S. H. Russ, K. Reece, J. Robinson, B. Meyers, R. Rajan, L. Rajagopalan, and C.-H. Tan, ‘‘Hector: An Agent-Based Architecture for Dynamic Resource Management,’’ *IEEE Concurrency*, vol.7, no.2, pp. 47-55, April-June 1999.

[11] P. A. Steenkiste, ‘‘A Systematic Approach to Host Interface Design for High-Speed Networks,’’ *Computer*, vol.27, no.3, pp. 47-57, March 1994.

[12] D. T. Stott, B. Floering, D. Burke, Z. Kalbarczyk, and R. K. Iyer,, ‘‘NFTAPE: A Framework for Assessing Dependability in Distributed Systems with Lightweight Fault Injectors,’’ *Proceedings of the 4th IEEE International Computer Performance and Dependability Symposium (IPDS-2K)*, Chicago, IL, pp. 91-100, March 2000.

[13] T. von Eicken and W. Vogels, ‘‘Evolution of the Virtual Interface Architecture,’’ *IEEE Computer*, vol.31, no.11, pp. 61-68, November 1998.