Reliability in LAM/MPI
Requirements Specification

Andrew Lumsdaine
Jeffrey M. Squyres
Brian Barrett

June 18, 2002

Contents

1 Introduction **4**
- 1.1 Clustering and Failures . . . 4
- 1.2 Reliable LAM/MPI . . . 4

2 Related Work **5**
- 2.1 Fault Detection . . . 5
- 2.2 Message Passing Fault Handling . . . 6
- 2.3 Fault Handling and MPI . . . 6

3 Failures, Failure Models, and Reliability **7**
- 3.1 Criteria for Successful Operations . . . 7
- 3.2 Transient Failures . . . 8
- 3.3 Operating System Failures . . . 8
- 3.4 Run-Time Environment Failures . . . 9
- 3.5 Application Failures . . . 10
- 3.6 Split-Brain Behavior . . . 10

4 The LAM Run-Time Environment **11**
- 4.1 Startup / Shutdown . . . 11
- 4.2 Steady-State Operation . . . 12
- 4.3 Failures . . . 12

5 MPI **12**
- 5.1 Startup / Shutdown . . . 13
- 5.2 Steady-State Operation . . . 13
- 5.3 Failures . . . 13

6 Application **15**

7 Reliable LAM/MPI/Application Models **15**
- 7.1 Failure Scenarios . . . 15
- 7.2 MPI Layer . . . 16
- 7.3 User Programming Models . . . 17
  - 7.3.1 Asynchronous Failure Notices . . . 17
  - 7.3.2 MPI Function Failures . . . 18
  - 7.3.3 Extending MPI Function Semantics . . . 22
- 7.4 A Higher Abstraction Model . . . 23

8 Requirements **24**
- 8.1 Requirements for LAM . . . 24
- 8.2 Requirements for MPI . . . 25
- 8.3 Requirements for User MPI Applications . . . 26

List of Figures

1 The structure of a typical MPI environment. 4
2 A typical parallel run-time environment and user application. 7
3 A parallel user application running on four nodes. 9
4 A parallel application running on eight nodes with a failed network between them. 10
5 A typical LAM run-time environment configuration. 11
6 The `lamboot` command launches the overall parallel run-time environment. 11
7 Layered implementation of MPI in LAM. 13
8 A user MPI process on node A has failed. 17
9 Sample code setting MPI_COMM_WORLD's exception handler to be MPI_ERRORS_RETURN. 18
10 Sample pseudocode showing how to check for the failure of a typical point-to-point operation. 19
11 Pseudocode showing that the list of dead MPI processes can be retrieved from wounded communicators. 20
12 Example MPI_REDUCE on a communicator with a failed process. 21
13 Timeline showing possible behavior of MPI_RECV with MPI_ANY_SOURCE in the presence of failures. 21
14 A wounded communicator can be subsetted to create a new, healthy communicator. 23
15 The robust MPI library exists as a layer between the user application and the native MPI library. 23

1 Introduction

Cluster computing has recently emerged as a cost effective platform for high performance computing [20, 22]. Since it is so well-suited to the distributed memory architecture presented by clusters, message passing programming has emerged as an important paradigm for high performance applications running on clusters. The Message Passing Interface (MPI) was developed to provide a standard (and vendor-neutral) set of library functions for message passing parallel programming and is presently the de facto standard [14, 15, 16, 23].

1.1 Clustering and Failures

There are some practical aspects of clusters that can present difficulties for MPI programs. In particular, clusters (and in the limit, the Grid) tend to be more dynamic and less reliable than the massively parallel processors that were in vogue when MPI was developed. Issues such as reliability and fault-tolerance are not well defined by MPI, nor are they handled well by existing MPI implementations. Failure of a single node or network link in a large cluster means having to restart an entire parallel job, either from the most recent checkpoint or from the very beginning.

As shown in Figure 1, the structure of a typical MPI application consists of multiple instances, each comprised of several software layers. Abstractly, entities such as the application or MPI cut across the nodes. Failure of a particular node introduces challenges at all levels: in the run-time environment (shown as LAM RTE in the figure), in MPI, in the application. For the application to survive the failure of a node requires participation at all levels.

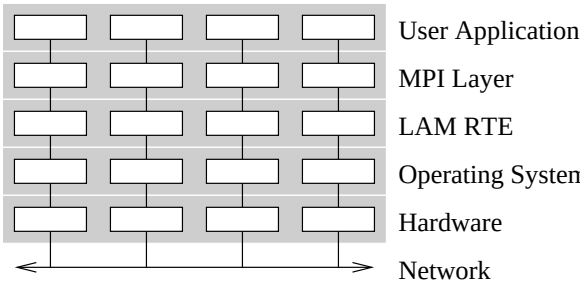

Figure 1: The structure of a typical MPI environment. Each vertical stack represents a single machine.

1.2 Reliable LAM/MPI

This project is to address some particular aspects of reliability in an MPI environment, focusing on run-time environment and its interactions with MPI and the application layer. Unlike traditional fault tolerant systems, which attempt to protect against a wide variety of errors, many of which are highly unlikely events in a cluster environment, this project seeks to allow execution to continue in the presence of a small set of common events. As the project seeks to provide a reliable environment in which to execute MPI applications, the term “reliable LAM/MPI” is used to describe the project.

Development will be in the context of LAM/MPI [5, 26], a project developed and maintained by the Open Systems Laboratory at Indiana University. LAM/MPI was originally developed at the Ohio Supercomputing Center, was adopted by the University of Notre Dame, and recently moved with its developers to its current home at Indiana University. LAM/MPI is an open source project, with binaries distributed as part of almost all major open source operating systems, including Red Hat Linux, Debian GNU/Linux, and FreeBSD. LAM/MPI contains a full implementation of MPI-1, as well as a significant portion of MPI-2.

This document describes the software requirements necessary to allow a parallel software application running on top of LAM/MPI to detect and recover from a catastrophic fault such as a compute node crash. The requirements include

1. Definition and categorization of failures to be handled by a reliable LAM/MPI application,
2. The behavioral (implementation) and interface requirements for LAM/MPI to provide reliable execution capabilities, and
3. The development of a preliminary design interface between LAM/MPI and an application wishing to recover from such an error.

The remainder of this report is organized as follows. Section 2 reviews previous work in the area of fault tolerance in parallel computing, particularly within the context of MPI. Section 3 provides an analysis of the types of failures that occur in a cluster environment and that this project will properly handle. Sections 4, 5, and 6 provide an overview of the current abilities of the LAM run-time environment, MPI layer, and user application (respectively) to determine when a failure has occurred, and how each can continue execution in the presence of failures. Section 7 presents a model for the extension of LAM/MPI to allow execution of an MPI application in a variety of failure scenarios. Finally, Section 8 provides a list of requirements for the Reliable LAM/MPI project.

2 Related Work

The problem of reliability in a distributed environment is not a new research topic. Research spanning at least three decades ranges from theoretical work on group management to checkpoint/restart systems for fault tolerant message passing environments. Specifically within MPI, a variety of methods for providing fault tolerance have been investigated, including checkpoint/restart, process notification, and redundant calculations.

2.1 Fault Detection

Fisher, Lynch, and Paterson's 1985 paper, "Impossibility of Distributed Consensus with One Faulty Process" [19], offers a formal proof that consensus on which processes are still alive in asynchronous systems is impossible when at least one process has failed (often referred to as the FLP impossibility result). Hence, within any asynchronous system (such as a point-to-point message passing system), it is impossible to provide a distributed application with consistent information on which processes are alive and which have failed.

Cristian and Schmuck [7] researched the concept of group membership in synchronous systems. They provided three protocols for solving consensus in synchronous systems in the presence of failures. Their results built upon earlier work that provided a method for implementing the needed synchronization system on a point to point network [8]. Other work [6] provides consensus algorithms with a weak level of synchronization.

Urbán, Défago, and Schiper [27] performed empirical studies on the applicability of the FLP impossibility result to real-world situations. Their results show that in a typical LAN environment and using an atomic broadcast, consensus can be correctly reached, even under extreme processor and network loads. The work shows that the FLP impossibility result often does not apply in a LAN setting, as there is some synchronous behavior in the environment.

2.2 Message Passing Fault Handling

PVM [3, 4, 9, 11] was the first widely used message passing system to implement a fault handling system. PVM allows processes to register with the local PVM daemon (the "pvmd") in order to be made aware of the failure of another process in the overall parallel job. When a pvmd is alerted to a failure in the run-time environment, it sends messages to the PVM user processes that registered for failure notices. Once alerted, the user process can perform appropriate recovery actions. PVM avoids the FLP impossibility result by never needing to reach consensus on group membership for its fault tolerance model. Instead, PVM only guarantees that each user process will *eventually* be notified of the failure.

CoCheck [24] provides a method for checkpointing a PVM job, allowing restart of a job after a failure. CoCheck uses the Condor [25] checkpoint library and focuses on allowing process migration in a network of workstations environment. Later releases of CoCheck also allow rudimentary checkpoint/restart and migration of MPI applications. While the focus is on migration, the checkpoint system provided by CoCheck also allows a job to be restarted in the presence of failures.

2.3 Fault Handling and MPI

One of the significant issues in making MPI more resistant to faults is the static nature of MPI communicators. William Gropp and Ewing Lusk proposed extensions to the MPI-1 standard [12] to allow dynamic process control from an MPI application. In particular, MPI_COMM_MODIFY allowed the dynamic resizing of a communicator. While many of the proposed extensions where added in the MPI-2 standard, MPI_COMM_MODIFY was not added, leaving communicators defined as static structures.

Other groups tried different approaches towards handling faults in MPI. Egida [21] is a toolkit for low-overhead recovery using checkpoint/restart and message logging/rollbacks. The project's goal is to provide a framework for developing and studying rollback recovery protocols in a variety of environments. In order to increase use, a modified version of the MPICH [13, 17] implementation of the MPI standard which utilizes the Egida library is provided. The Egida version of MPI provides the ability to restart after a failure without imposing high overhead operations during normal execution. After a failure, a watchdog script is used to restart the application using the logged data.

Starfish [1] is an attempt to provide a high-performance fault tolerant parallel environment. Starfish applications benefit from the checkpoint/restart abilities of the Starfish run-time environment. A set of daemons form a run-time environment capable of restarting applications after a failure, controlling checkpointing, and message passing. Since a significant portion of the infrastructure resides in the daemons, the size of the data that must be checkpointed is greatly reduced from other systems. In addition to checkpointing, Starfish provides a mechanism to register failure notification handlers that allow applications to re-partition themselves to continue without the need to restart.

The FT-MPI [10] project provides the ability to continue execution in the presence of failures. The project redefines MPI communicators, allowing them to be "healed" when a failure occurs. MPI functions return an error when communication is attempted with a failed process. Unlike Starfish, the notification of a failure is not asynchronous – it occurs only while the application makes an MPI call. Upon notification of the failure, the user application is then responsible for "healing" the communicator and continuing execution.

MPI/FT^{TM} [2] proposes a model in which MPI is extended to provide fault tolerance in an unreliable environment. In addition to traditional checkpoint/restart failure protection, MPI/FT provides redundant calculations, voting, and guarantees about the internal state of the MPI layer in the presence of application failures. While receiving maximum fault tolerance requires using non-portable MPI/FT calls, the system provides some failure protection against legacy MPI applications.

All existing fault tolerance models have their drawbacks for the problem this project is attempting to solve. Egida does not provide the possibility of continuing after a failure, only restarting. Starfish's asyn-

chronous notification scheme is only intended for trivially parallel applications and does not provide a well-defined process for continuing after a failure. FT-MPI defines a method for rebuilding communicators, but requires non-portable extensions to MPI. In addition, the project only implemented a small subset of the MPI standard. MPI/FT is intended for environments much more hostile than the large cluster of dedicated hardware this project targets. As a result, use of MPI/FT involves dedicating a large portion of the available resources to redundant calculations in case a failure occurs.

3 Failures, Failure Models, and Reliability

A failure can occur in any of the layers of a parallel application. Generally, failures that occur in lower layers will propagate to upper layers. The sections below discuss failures that occur in the context of a parallel run-time environment and parallel user application running on multiple nodes. Each section attempts to answer the following questions:

1. What happens to the parallel run-time environment and application when a failure occurs?
2. Can the parallel run-time environment and/or application recover? If so, how?

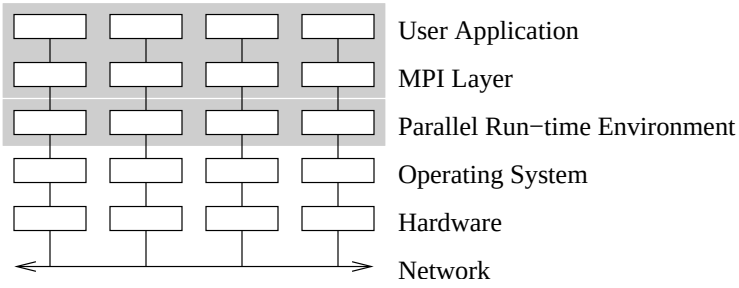

Figure 2: A typical parallel run-time environment and user application, each comprised of the union of local instances, potentially distributed across multiple nodes. User application instances each span two layers – MPI is, by definition, a library, and therefore shares the same Unix process space with the user application.

Figure 2 shows a typical parallel run-time environment and user application. The overall parallel run-time environment is comprised of the set of local instances of run-time environments, one on each node. Similarly, the user MPI application is comprised of the union of individual instances of the user program (which may be distributed across multiple nodes).

Section 3.1 discusses the conditions necessary for successful operation of a parallel run-time environment. Sections 3.2 through 3.6 discusses several types of failures in a parallel run-time environment, and the resulting behavior of the top layers from Figure 2.

3.1 Criteria for Successful Operations

There are many conditions that can cause failures for a distributed parallel run-time system or parallel application. Rather than define a large number of failure-causing errors, it is easier to discuss the criteria required for success for the lowest layer of software in the system. Any condition not meeting that criteria is therefore a failure.

The lowest software layer of interest is the parallel run-time system itself. Its criteria for successful operation are:

- An instance of the run-time environment executing on each node
- Continuous network connectivity between each pair of individual run-time environment instances

Any condition that causes either of these criteria to not be met will cause a failure in the overall run-time system. Examples of such conditions include (but are not limited to):

- Death of an entire node (i.e., killing all processes that were running on it), such as a power failure, manual reboot, or other hardware / operating system reset
- Death of the local run-time environment process on a node, although the node itself is otherwise functional
- Interruption of network services between nodes, such as hardware failure, heavy congestion and data loss, or physical network media disconnect

As such, the layers under the run-time environment are treated as a black box system. Failures that propagate from anywhere in the black box into the run-time system are examined only in the context of the run-time system itself. This is, after all, the purpose of a layered architecture.

While the exact condition and cause of the error is unimportant to the run-time environment itself, several broad types of errors must be examined to define the desired behavior of the run-time environment and the user application.

3.2 Transient Failures

Transient failures are defined as "temporary" errors that occur in layers below the run-time system. These types of failures appear for some length of time, and then correct themselves with no outside assistance. An example of a transient failure is a network that drops packets when congested (such as a TCP/IP network). This can be seen when network connections are rejected even though both the client and server nodes are both available and functioning properly.

While the hardware and operating system will propagate these kinds of errors up into the run-time system, there is usually no way for the run-time system to know that the failure is only temporary. Indeed, there is typically no way to distinguish it from other kinds of lower-layer errors. Typical approaches to this problem are to try the failed action multiple times over a defined timeout period. If the action finally succeeds, no failure needs to propagate to a higher level. If the action does not succeed within the timeout period, it is deemed a failure, and the appropriate error condition must be propagated up to a higher level.

Transient failures are addressed in the run-time system itself. It will retry actions (as appropriate) over a timeout period before declaring that the action has failed. The parallel user application then only needs to receive the ultimate "success" or "failure" result and not be concerned with retrying an action or timeout periods.

3.3 Operating System Failures

Operating system failures are rare, but can occur when an operating system either has a bug or makes specific policy or resource decisions while operating. Bugs in operating system library or system calls result in unreliable or erroneous behavior of both run-time environment and user programs (both serial and parallel). Since the failure is a bug, there is no guaranteed reliable way to handle the error at run-time, and therefore the failure must be fixed by altering the application code to not invoke the bug.

Indeed, many portable software projects are distributed with a GNU "configure" script that attempts to discover characteristics of the operating system that it is running under, and configure the software as

appropriate. It is not uncommon to see configure scripts that check for specific operating system bugs in order to activate alternate code paths to workaround known erroneous behavior.

An operating system may also decide to kill an application and/or reduce the resources that have been allocated to it (perhaps while under heavy load). In such cases, operating systems typically send some kind of asynchronous notification to the process indicating that reduction that will occur. It is up to the application to handle these failures; specific code must be supplied that can catch, recognize, and handle these operating system-specific events.

For example, if a local instance of the parallel run-time environment is notified that it is going to be killed, appropriate handling may include killing all of the local user applications on that node before dying.

3.4 Run-Time Environment Failures

The run-time environment acts as a distributed service provider between the user and a collection of nodes and operating systems. It therefore only relays service requests to the appropriate destination and returns exit statuses from the underlying services. Hence, all errors in the parallel run-time environment are the result of failures that have propagated up from lower levels – no errors are originated from within a properly functioning run-time environment itself.

For example, a network error may indicate that a remote node is no longer alive. The run-time environment must handle these kinds of errors by closing down stale network connections, cleaning up all orphaned state, freeing excess memory, marking remote nodes as "failed," etc.

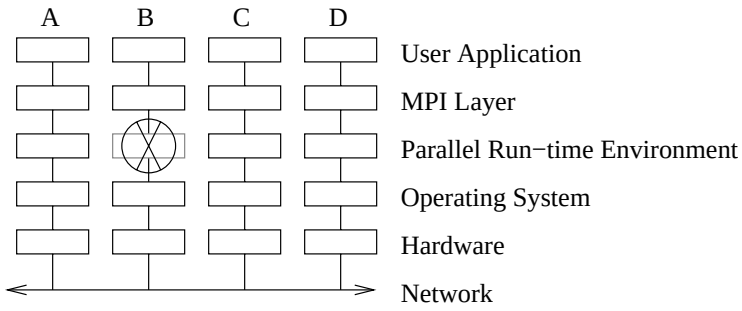

Figure 3: A parallel user application running on four nodes. Although node B itself does not fail, the run-time environment fails on B, which will propagate up to the user application and cause it to fail as well.

Figure 3 shows a failed local instance of the run-time environment on node B. When the local instance of the run-time environment on A fails to communicate with its peer on node B (after a timeout and/or series of retried communications), A will declare node B "failed."

Note that this may happen even if neither node B nor the user application running on node B have failed. For example, if the run-time environment's network connectivity success condition is not met, the other nodes in the run-time environment will assume that *everything* on node B has failed. This is consistent with the stack-based model shown in Figure 3 – if a module in the stack fails, all levels above that module must also fail. Hence, the failure of the run-time environment is an unrecoverable error for a user application.

The remaining local run-time environment instances need to propagate this failure to the surviving instances of the user application. The user application must take appropriate actions to handle the failure event. This may include peer data structure management and cleanup, attempting to start a new user application instance, etc.

3.5 Application Failures

Failures in the application, for the purposes of this document, are considered bugs. This class of failures include (but is not limited to): segmentation faults, buffer overflows, data mis-management, not checking the return codes from system, library, and parallel run-time API calls, etc. These all need to be handled by the application itself – the parallel run-time environment is not directly involved. If errors are not handled by the application, the operating system defaults will be used instead, which usually terminates the process.

If a process terminates without properly shutting itself down to the parallel run-time environment, it is the run-time environment's responsibility to clean up any state associated with that process, and notify other instances of the user application that the local instance has failed.

3.6 Split-Brain Behavior

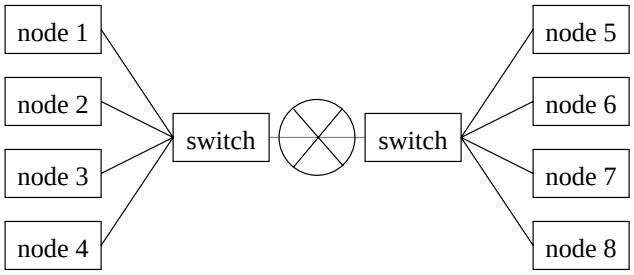

Figure 4: A parallel application running on eight nodes with a failed network between them. Each of the two groups think that the other group has failed. This is called "split brain" behavior.

A notable type of error is called "split brain behavior." Figure 4 shows a parallel user application running on eight nodes with a failed network between them, effectively splitting the parallel application into two groups. Each group thinks that the nodes in the other group have failed, when only the interconnecting network has failed – not any nodes or processes.

Hence, no nodes have actually failed, but it *appears* to each group that the other has failed. This kind of error spans both the run-time environment and the user application – both may potentially need to execute specific recovery steps. Since the network is part of the black box beneath the run-time system, it can be quite difficult to distinguish between the failure of remote nodes from the failure of intermediate hardware.

A typical solution to avoid split brain behavior is for the run-time environment to designate one node as a "master" node (e.g., node 1 in Figure 4). If any local run-time environment instance can no longer communicate with the master node, it will abort. Adhering to the stack model, this will also abort any user processes on that node. Hence, in the case of a network failure (as shown in Figure 4), split brain behavior will not occur because nodes 5 through 8 will abort when they realize that they can no longer communicate with the master node (node 1).

While this approach is effective, it does create a single point of failure. Indeed, if the master node itself fails, it will cause the rest of the run-time environment (and user application) to abort. Although this is undesirable, protecting against failure on just the one node can be relatively inexpensive as compared to "hardening" all nodes.[1] Additionally, the relative cost of trying to reassemble fragmented results from a parallel job that degenerated into split-brain behavior could be much higher than simply re-running the entire job.

[1]Common protections against failure include uninterruptable power supplies, RAID filesystems, redundant power supplies, processors, RAM, etc.

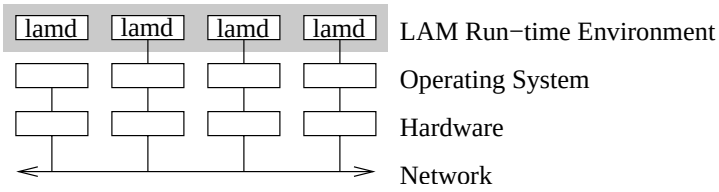

Figure 5: A typical LAM run-time environment configuration. A `lamd` on each node is the local instance of the run-time environment, the union of which comprises the overall parallel run-time environment.

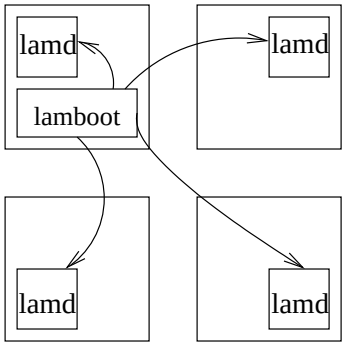

Figure 6: The `lamboot` command launches a `lamd` locally on each node to create the overall parallel run-time environment.

4 The LAM Run-Time Environment

LAM is a user-level run-time environment that provides a framework for running parallel programs. LAM utilizes user-level daemons on each machine to create a persistent run-time environment. Figure 5 shows a typical configuration, with a local instance of the run-time environment running on each node; each instance is referred to as a `lamd`. While a `lamd` may actually be one or more Unix processes, it is always discussed as a single entity in this document.

4.1 Startup / Shutdown

The collection of `lamd`s is started by the `lamboot` command. `lamboot` is invoked with a list of hosts on which to create the run-time environment. The `lamboot` command dispatches requests on each machine (typically via `rsh` or `ssh`) to individually start each `lamd`. Figure 6 shows this process.

As each `lamd` is started, it is seeded with a routing table containing the addresses of all the other `lamd` processes. In this manner, each `lamd` becomes aware of its peers and starts coordinating on a global scale. This transforms the group of individual `lamd` processes into a unified parallel run-time environment.

The `lamhalt` command is used to shut down a LAM run-time environment; it sends a shutdown control messages to each `lamd`. A notification of the global shutdown is first sent to each node so that each `lamd` knows that its peers will be shutting down and it should not generate failure notices. This is followed by sending a “shut yourself down” message to each `lamd`. Upon receipt of this message, each `lamd` kills all user processes, releases resources, and generally exits in an orderly fashion.

4.2 Steady-State Operation

Once a LAM run-time environment has been successfully booted, it enters steady-state operation. Two kinds of actions occur during steady-state operation:

- User-initiated actions
- Failure detection and router table maintenance

User-initiated actions are commands (or library calls) issued to the LAM run-time environment. These commands are requests from the user to specific parts of the run-time environment to perform some action, and typically involve sending some control messages between `lamd` instances. The `lamexec` command, for example, launches arbitrary executables on nodes in the run-time environment. This entails sending a short control message to each node that the executable is to be run on containing the name of the executable, command line arguments, relevant environment variables, etc. Other user-initiated actions include direct message passing, manipulation of files on remote nodes, and relaying `stdin`, `stdout`, and `stderr` between processes.

Failure detection is periodically performed by each `lamd` with "heartbeat" messages and normal communication traffic. Each `lamd` monitors the "health" of all the other `lamd` instances. The health of a node is judged by its ability to continue to send and receive run-time environment control messages. Healthy nodes can send and receive message traffic (and therefore assumedly correctly process the individual messages). Unhealthy nodes either do not respond to control messages or do not send any control messages.

An obvious problem with this approach is that each `lamd` is responsible for monitoring the health of all other nodes. This leads to scalability problems, mainly in the form of propagation delays since each node individually determines when another node has failed. The heartbeat mechanism does have a scalable bandwidth-limiting mechanism, although with the tradeoff that it may take a long time to find out if a given node is down. This can lead to inconsistent state in the run-time environment about what nodes are available and what nodes have failed, race conditions with failure notices, and deadlock conditions in some cases.

4.3 Failures

LAM's current heartbeat mechanism will eventually discover when a node has failed. Upon detection, the run-time environment will heal itself and continue operating correctly. In most cases, however, the run-time environment will not propagate the error to the MPI layer and user applications. As such, the MPI layer (and user application) may deadlock while waiting for a failed process because it is not aware that the process has died.

Another problem is that once the run-time environment on a node dies, there is no mechanism to recover that node back into the overall run-time system once it has been "fixed". For example, if a failed node gets rebooted and is capable of handling computations again, LAM currently provides no way to bring that node back into the run-time environment. Indeed, upon detection of a failed node, LAM will completely remove that node from its routing tables. This precludes the possibility of cleanly "bringing a node back from the dead" because the identity of that node has been removed.

5 MPI

LAM also provides an implementation of MPI. MPI is layered on top of the LAM parallel run-time system and uses many of the services that LAM provides, including process control, out-of-band message passing, and meta-data storage. Although not shown in Figure 7, the MPI layer does directly interact with the

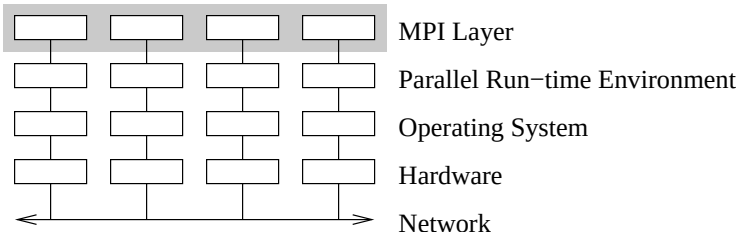

Figure 7: Layered implementation of MPI in LAM.

operating system to maintain its own communication channels for MPI processes (mainly for optimization reasons).

5.1 Startup / Shutdown

An MPI process is started with the LAM command `mpirun`. This invokes a complicated sequence of events that includes:

1. Passing the executable name, relevant command line parameters, environment variables, and other meta data to each of the target nodes
2. Loading and starting the executable on each of the target nodes
3. Synchronization and mutual self-awareness of the new MPI processes

Process control is mainly handled by the LAM run-time environment. The synchronization and mutual self-awareness phase is handled within MPI_INIT. Once MPI_INIT returns, all MPI processes have become aware of each other and have formed a consistent MPI_COMM_WORLD.

The MPI standard mandates that all of the the MPI processes in MPI_COMM_WORLD must invoke MPI_FINALIZE before exiting. MPI_FINALIZE performs the bookkeeping of shutting down the MPI layer, shutting down the LAM layer, and exiting in an orderly fashion such that its death will not cause failure errors in its peer MPI processes.

5.2 Steady-State Operation

Once MPI_INIT has returned, the user application is free to perform its own computations. MPI provides a number of API function calls to effect message passing, such as MPI_SEND and MPI_RECV. LAM's MPI layer will choose one of the available channels for message passing: TCP sockets, shared memory, Myrinet, or the so-called "`lamd`" channel. The `lamd` channels routes all MPI traffic through the `lamd` instances.

Message passing failure detection is handled by the MPI layer, or propagated up lower layers. As such, user MPI programs do not need to be concerned with retransmission, timeouts, network errors, etc. MPI_SEND and MPI_RECV will either succeed or fail – user applications are not required to provide extra code to assist MPI_SEND/MPI_RECV.

5.3 Failures

It is important to note that the MPI-1 and MPI-2 documents make few stipulations about run-time errors. Indeed, MPI-1 states that implementations are free *not* to handle errors. Failures are completely left up to the implementation to decide how to handle (or not). MPI-1 section 7.2 states:

> "After an error is detected, the state of MPI is undefined. That is, using a user-defined error handler, or MPI_ERRORS_RETURN, does not necessarily allow the user to continue to use MPI after an error is detected. ... An MPI implementation is free to allow MPI to continue after an error but is not required to do so."

Hence, any additional guarantees that LAM/MPI is able to provide about failures are not required, but will be compliant with the MPI standards. LAM/MPI's default error handler, however, will still be MPI_ERRORS_ABORT – which will abort the user program – as mandated by the MPI standard. To use LAM's additional error-handling capabilities, another error handler (such as MPI_ERRORS_RETURN) must be used.

If an MPI function involving another MPI process returns a failure indicating that the remote process has failed (i.e., something other than MPI_SUCCESS), the user application can safely assume that the target MPI process will be unavailable for the remainder of the computation.[2] Indeed, the default error handler for MPI programs (including LAM/MPI programs) is to abort on any error, so many MPI programs do not attempt to handle errors at all.

Failures will either be propagated up from the LAM layer or operating system, or will be detected by the MPI layer itself.

The LAM run-time system detects an error. In most cases, LAM will kill the entire parallel user application.[3] There are some cases where LAM will not kill the user application (e.g., where the error handler has been set to MPI_ERRORS_RETURN), and the MPI layer will allow all the other MPI processes to continue. In these cases, MPI calls to communicate with the dead MPI process(es) will return appropriate error codes. The state of MPI should be stable, but the code has not yet been audited closely to verify this.

The MPI layer detects an error. The MPI layer has a secondary mechanism for discovering that MPI processes have failed. This mechanism relies on the communication channel itself reporting errors. Each time an MPI communication is started, the initiating MPI process checks the status of the communication channel between itself and its peer. This works well with TCP sockets because the operating system will report if the remote peer has closed the socket.

It does not always work well for connectionless communication channels such as Myrinet, shared memory, or the "`lamd`" mode communication. This is because these channels will not report if a remote process dies because there is no concept of a single remote peer. Hence, there are failure cases where LAM will hang indefinitely because the MPI layer may be unable to detect the error. This is a shortcoming of the current LAM failure detection model; it only happens when the LAM run-time environment also does not detect the error immediately (e.g., when both the user application and the `lamd` are simultaneously killed on a node when a node is rebooted).

If the MPI layer detects a failure, the error handler actions are the same as if the LAM layer detects a failure – if the error handler is MPI_ERRORS_ABORT, the entire user parallel process will will be killed. If the error handler is MPI_ERRORS_RETURN, the process may continue with the same stipulations noted above.

[2] Errors returned from MPI collective calls do not necessarily inform the caller which MPI process has failed.

[3] Note that this is a completely asynchronous action – even though LAM is a single-threaded MPI implementation, the remaining MPI processes will be killed regardless of whether they are in MPI functions or not.

6 Application

MPI applications running under existing MPI implementations have little ability to withstand failures in the run-time environment. Most MPI implementations, including LAM/MPI and MPICH, default to aborting execution of an entire application if any one process in the application fails. The best-case scenario for many MPI applications is to be killed – the worst-case scenario is a deadlocked application that will never complete.

The LAM/MPI implementation allows the use of MPI-2 dynamic process control to avoid the death of an entire job when one process fails. LAM/MPI will not kill the entire application when a process created by MPI_COMM_SPAWN dies. The solution is imperfect, however, because it makes no guarantees about notification of the failure and does not lend itself to all parallel computing models. In addition, not all common implementations of the MPI standard have a full implementation of the dynamic process control, so code utilizing MPI_COMM_SPAWN currently has limited portability.

Checkpoint/restart mechanisms offer a practical method of surviving failures. When integrated into the MPI implementation itself, the user program does not need to be modified – in many cases, the entire application can be checkpointed and restarted without the application's knowledge. User-level checkpointing is more common, however, where only critical user data is saved (this also provides portability to non-checkpointing MPI implementations).

Neither checkpointing model alone, however, allows the application to automatically recover from a failure and continue execution. Additional mechanisms must be in place for restarting the application after a failure has occurred. This can be implemented in the MPI system itself (and/or paired up with a batch queueing system), or be an external user mechanism (which may be as simple as a shell script).

Checkpoint/restart mechanisms provide a convenient method of saving the state of an application and restarting after a failure. However, if the application is to continue execution without a complete restart, the application must be informed that a failure has occurred so that it can re-partition itself in order to avoid relying on the dead process for data or computation. The MPI interface and underlying run-time environment must also ensure that it does not cause a process to "hang" as the result of a failure – otherwise applications will require human intervention to restart.

7 Reliable LAM/MPI/Application Models

The LAM implementation of MPI aims to provide a level of reliability that enables the LAM run-time environment and a user's MPI application to survive some kinds of failures. Additionally, it is desirable to provide this reliability in an MPI-compliant fashion. That is, user MPI code that exploits the reliable functionality in LAM/MPI should be able to be written portably; when running under LAM/MPI, the failure detection and handling code is used at run time. But when running under another MPI that does not support the same reliability model, the failure detection and handling code is effectively ignored, or never triggered.

Sample failure scenarios are described below (with respect to previous failure definitions) to provide a framework for a discussion of the role of the MPI layer in failure detection, as well as the programming models that are necessary to recover from such failures.

7.1 Failure Scenarios

Several common scenarios are described, as well as their ramifications and possible recovery options. Note that all of the scenarios assume that the both the LAM run-time environment is running in "reliable" mode[4]

[4]There is long-standing precedent in parallel run-time environments that the failure of one parallel process will kill the entire rest of the parallel application. LAM's default behavior therefore will still utilize this model; LAM must be specifically told to run

and that the user's parallel application is an MPI job that has been written to utilize the LAM failure detection mechanism and includes application-specific recovery logic.

Complete failure of a node. A power supply failure causes the complete failure of a node in a cluster. A user's MPI application was running under LAM on the node at the time of the failure.

A total power failure guarantees that all processes on the machine at the time of failure are lost. The LAM run-time environment will notice that the `lamd` on the failed node is no longer responding and will reorganize the run-time environment to exclude the failed node. The LAM run-time environment will notify the remaining MPI processes in the MPI application. The notification will convey the fact that the failed user application instances disappeared, as opposed to terminating abnormally or exiting gracefully. The surviving processes in the user's MPI application can take whatever action is appropriate, including reorganization of internal data structures, aborting, spawning a new MPI process to replace the failed processes, etc.

Node becomes disconnected from the network. A cabling failure results in two groups of nodes being disconnected from each other (see Figure 4, on page 10).

The system has essentially been divided into two groups of nodes, each with separate MPI jobs running under their control. Only one of the two groups will contain the master node.[5] The run-time environment instances in the non-master group will realize that they can no longer communicate with the master run-time environment instance, and therefore abort. This will cause any user applications running on these nodes to also abort.

The run-time environment instances in the master group will realize that they can no longer communicate with the nodes in the other group, and will declare those nodes as failed. This will trigger the standard notification mechanism to all the surviving MPI processes.

User application failure. A programming error causes some of the MPI processes in a parallel application to abort from a segmentation fault.

The LAM run-time environment will be alerted to the abnormal death of the MPI processes by means of normal Unix process control semantics. LAM will notify the surviving MPI processes by the same mechanism described above. The fact that the failed MPI processes died abnormally will be conveyed to the survivors, since it may influence their decision as to whether to try to recover and continue or to outright abort.

Note that the default action for an MPI application is to abort upon process failure, which may be the safest alternative for the application, given that an abnormal death may indicate that the application itself is at fault.

7.2 MPI Layer

The MPI layer will be notified of failures in peer MPI processes in at least one of two ways: the communications channel to the remote process will indicate that the remote process has disconnected, or the LAM run-time environment will asynchronously signal that a specific MPI process has failed. Figure 8 shows both mechanisms – a user application on node A fails, triggering a notification from the `lamd` on node A to the `lamd` on node B, which is then propagated up to the MPI layer. Additionally, the MPI layer may be notified directly by the communications channel that it has been broken.

in "reliable" mode where it will not kill an entire parallel job when one MPI process fails. This will likely be through a command line switch to `lamboot` or `mpirun`.

[5] Note that LAM/MPI implicitly defines the master node as the node on which `lamboot` was executed to launch the overall run-time environment.

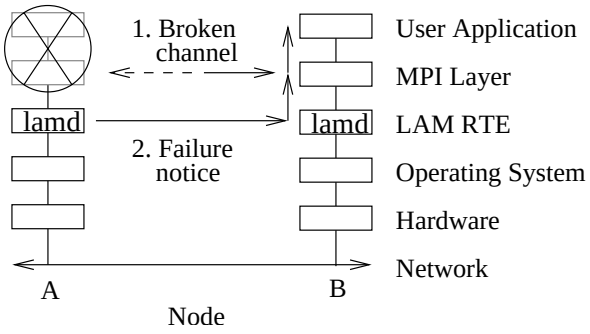

Figure 8: A user MPI process on node A has failed. If supported, the communications channel on node B will notify the MPI layer that its peer on node A has unexpectedly shut down. The `lamd` on A will notice that the application has failed, and send a failure notice to the `lamd` on B, which will be propagated up to the MPI layer. The user application may also choose to receive the asynchronous failure notices (but from the MPI layer, not from the run-time environment directly).

Note that if node A completely fails, the end result is the same (the MPI layer on node B is asynchronously notified of the failure of the user application on node A), but the mechanism is different. In this case, there will be no `lamd` on A to send a failure notice. Instead, the `lamd` on node B will notice that it can no longer communicate with the `lamd` on A, and therefore directly propagate a failure notice up to the MPI layer.

The user application may register a callback with the MPI layer to receive failure notifications. The MPI layer will invoke the user callback(s) when it is notified of process failures from the LAM layer.

7.3 User Programming Models

Reliable MPI user applications must be written to handle the fact that MPI processes may fail. User applications are notified via two different mechanisms when one of its peers has failed: user-supplied callback functions that are asynchronously invoked when the MPI layer learns of a process failure, and MPI functions returning error codes other than MPI_SUCCESS (and therefore triggering an MPI exception).

7.3.1 Asynchronous Failure Notices

An asynchronous failure notice mechanism can be used by the user application to discover peer MPI process failures. The user application can register a callback function that will be invoked when the MPI layer is notified of the failure of a remote process. Each MPI process failure will eventually trigger the invocation of callbacks on all of its peer processes that have registered to receive failure notices. The callback function is associated with a communicator; the callback will be invoked (on each process) once for each "wounded" communicator that the failed MPI process was in.

The semantics of the callback function are similar to a Unix signal handler – it can be called at any time, and only async-signal safe functions are allowed to be invoked from within the handler. No MPI functions can be invoked from the callback. Its main purpose is to record the failure in some global state that can be seen on the main code path.

The callback is a *local* action. The exact timing of the callback across all processes in the parallel user application is not rigidly defined. The callback mechanism may be viewed as a lazy updater providing out-of-band information about the state of the user application.

In practice, the callback will be "soon" after the actual failure – the main delays are the detection time required by the LAM run-time environment, and the propagation time throughout the rest of the surviving processes. The lazy updating mechanism provides the following guarantee: upon return from MPI_BARRIER invoked on wounded communicator A, callbacks will have been invoked for each failed process in A on each of the surviving processes in A.

In the event of multiple failures, or where one MPI process was in multiple MPI communicators (each of which was registered to receive failure notices), the ordering of callbacks is undefined.

7.3.2 MPI Function Failures

Regardless of whether the user application utilizes the asynchronous notification method or not, MPI communication functions involving dead processes will fail.

MPI Exceptions. If an MPI process fails, its peer processes will generate an MPI exception when attempting to communicate with it. Examples of actions that will generate an MPI exception include (but are not limited to): calling MPI_SEND to send a message to a failed MPI process, calling MPI_RECV to receive a message from a failed process, calling MPI_BARRIER on a wounded communicator. Once the MPI layer marks a process as "failed," all future MPI communications involving that process will fail immediately.[6]

The default MPI exception handler is MPI_ERRORS_ABORT, which kills the entire MPI application. Hence, an important setup step in a reliable MPI program is to assign a different MPI exception handler that will not kill the entire application, such as MPI_ERRORS_RETURN or MPI::ERRORS_THROW_EXCEPTIONS (C++ only). This will enable failed MPI calls to return, allowing the user application to catch the error and attempt to handle it. See Figure 9.

```
int main(int argc, char* argv[]) {
  MPI_Init(&argc, &argv);
  MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
  // ...rest of program...
  MPI_Finalize();
  return 0;
}
```

Figure 9: Sample code setting MPI_COMM_WORLD's exception handler to be MPI_ERRORS_RETURN. All communicators derived from MPI_COMM_WORLD will inherit this error handler setting.

Non-blocking MPI functions can also cause exceptions. For example, MPI_ISEND could trigger an MPI exception when used to send to a process that is already marked as failed. However, if MPI_ISEND is used to send a message to a process that later fails (but before the message is actually transferred), the MPI exception won't be triggered until the MPI request is checked via MPI_TEST or MPI_WAIT.

[6] A subtle side-effect of failure detection is that any "unexpected" messages that have already been received from the failed process by the local MPI layer will be discarded. Unexpected messages are messages that have been received and buffered by the MPI layer even though no matching receive has been posted by the user application.

Point-to-Point Failures. Failed calls to point-to-point functions that do not use MPI_ANY_SOURCE[7] will return an error code that is not equal to MPI_SUCCESS. This indicates that the remote MPI process has failed.

Notification of process failures are delivered to the MPI layer in each MPI process, and may arrive before, during, or after the point-to-point function call:

1. If the failure notification is delivered before point-to-point function is invoked, the point-to-point function can immediately invoke the appropriate MPI error handler without attempting any communication.

2. If the failure notification is delivered after the point-to-point function started, but *before* the message transfer has completed, the point-to-point function will abort the message transfer, clean up any auxiliary state, and then invoke the appropriate MPI error handler.

3. If the failure notification is delivered *after* the message transfer has completed, the point-to-point function will return MPI_SUCCESS and defer the failure notification until the next invocation of a communications function involving the failed process.

In all cases, however, any future point-to-point (or collective) message passing involving the failed process will trigger an MPI exception. Figure 10 shows sample pseudocode checking the return status of a call to MPI_SEND. Point-to-point failures involving the use of MPI_ANY_SOURCE have semantics similar to failed collectives, and are described below (see page 21).

```
if (MPI_Send(..., dest, tag, mycomm) != MPI_SUCCESS) {
  // ...error handling here...
  // These calls will immediately fail
  MPI_Send(..., dest, tag, mycomm);
  MPI_Barrier(mycomm);
}
```

Figure 10: Sample pseudocode showing how to check for the failure of a typical point-to-point operation. Also note that the calls to MPI_SEND and MPI_BARRIER will immediately fail because the `dest` process is now known to have failed.

Collective Failures. A collective function that fails with an error code not equal to MPI_SUCCESS indicates that at least one MPI process in the communicator has failed. Any future collective message passing on that wounded communicator will fail. The fact that a collective operation has failed does not necessarily indicate *which* process (or processes) on the wounded communicator has failed – it only indicates that one or more have failed.

The only reliable way for the application to discover which MPI process has failed is to attempt a point-to-point communication with every other process. Since this can be expensive for large communicators, LAM will provide an alternative local mechanism to retrieve the information. An attribute (or set of attributes) will be set on wounded communicators, indicating which process(es) have failed. The user process

[7] Receiving messages from MPI_ANY_SOURCE shares behavioral characteristics with collective functions; see page 21 for discussion of calls to point-to-point functions that use MPI_ANY_SOURCE.

can retrieve this attribute (or attributes) to discover which process(es) had failed by the end of the failed collective operation. Figure 11 shows example pseudocode for this process.

```
if (MPI_Barrier(MPI_COMM_WORLD) != MPI_SUCCESS) {
  MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_FAILED_LIST, &dead, &f);
  if (f == 1) {
    MPI_Comm_get_attr(MPI_COMM_WORLD, MPI_FAILED_SIZE, &size, &f);
    for (i = 0; i < size; ++i)
      print("Rank %d has died\n", dead[i]);
  }
}
```

Figure 11: Pseudocode showing that the list of dead MPI processes can be retrieved from wounded communicators. This is example pseudocode only, meant to show that standard MPI mechanisms will be used to retrieve this type of information. MPI_FAILED_LIST, MPI_FAILED_SIZE, and an array of rank numbers are only one possibility for how the MPI implementation may provide the information to the user application.

Asynchronous Collectives. MPI allows for collective operations to return different error codes on participating MPI processes. Specifically, it is legal for some participating processes to return successfully while other processes trigger MPI exceptions. MPI-1 section 4.1 states:

> Collective routine calls can (but are not required to) return as soon as their participation in the collective communication is complete. The completion of a call indicates that the caller is now free to access locations in the communication buffer. It does not indicate that other processes in the group have completed or even started the operation (unless otherwise indicated in the description of the operation). Thus, a collective communication call may, or may not, have the effect of synchronizing all calling processes. This statement excludes, of course, the barrier function.

Hence, any collective that does not imply a global synchronization may allow some MPI processes to return successfully, even on wounded communicators. Only global synchronization functions such as MPI_BARRIER are guaranteed to behave identically (in terms of detecting failures) across the set of surviving processes in a wounded communicator.

For example, non-root processes in an MPI_REDUCE call only need to send their contribution to one other process; if each non-root process can communicate with its designated reduction peer, it will not notice that any other process has failed because no global synchronization is required. Figure 12 shows that processes C and D complete MPI_REDUCE successfully, even though process B has failed. Only the root, process A, is made aware of the failure.

In general, rooted collective communication functions exhibit this behavior, mainly for performance optimization reasons.[8] User applications that require uniform success/failure semantics from collectives should follow calls to rooted collectives with a call to MPI_BARRIER. If an error occurs in MPI_BARRIER, appropriate error handling measures can then be taken relative to the rooted collective operation.

[8] Imposing synchronization on all collective operations would significantly degrade message passing performance.

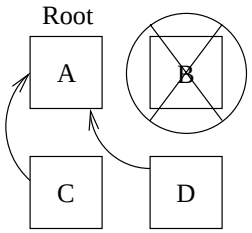

Figure 12: In this example, four processes are in a common communicator. Process B has failed. Processes A, C, and D call MPI_REDUCE on the common communicator. It is possible that processes C and D will perform their part of MPI_REDUCE and complete successfully, and that only A will recognize that B has failed.

Point-to-Point Failures with MPI_ANY_SOURCE. Receiving messages using MPI_ANY_SOURCE is similar to asynchronous collectives – failures may or may not be detected. Since failures are only guaranteed to be detected during synchronizing communications, previously undetected failures may not cause errors if the receive operation can complete with local information.

Figure 13 shows an example scenario where an MPI_RECV using MPI_ANY_SOURCE may succeed even though a failure has already occurred. At $t = 0$, process C sends a message to A. All three processes synchronize at the next timestep, which allows process A to receive the message from C into its unexpected message queue. At $t = 2$, process B fails. The LAM run-time environment initiates the failure notification to processes A and C, but it takes time to propagate to its destinations.

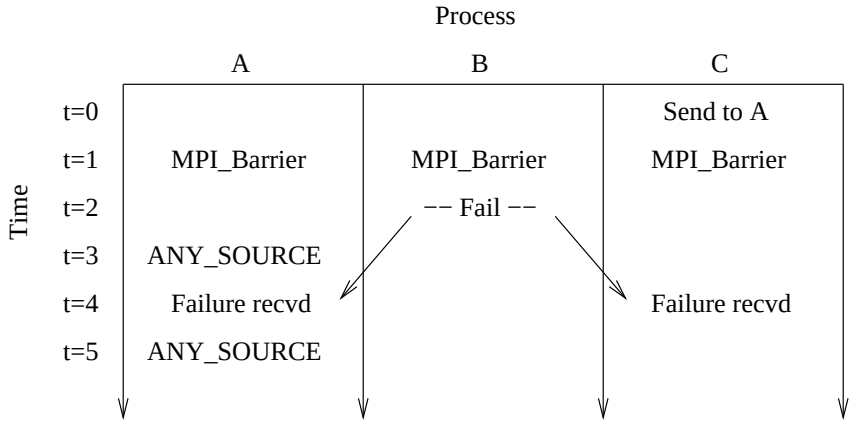

Figure 13: Timeline showing possible behavior of MPI_RECV with MPI_ANY_SOURCE in the presence of failures.

Meanwhile, at $t = 3$, process A calls MPI_RECV with MPI_ANY_SOURCE. A sees that it has a matching unexpected message buffered from C, and therefore returns successfully. The asynchronous failure notice is finally delivered at $t = 4$, and the MPI layer in processes A and C mark process B as "failed." When process A calls MPI_RECV with MPI_ANY_SOURCE again at $t = 5$, a failure will be returned immediately since process B has been marked as "failed."

7.3.3 Extending MPI Function Semantics

Two MPI functions need to be extended in order to allow effective use of the reliable MPI semantics described in previous sections: MPI_COMM_FREE and MPI_COMM_SPLIT.

MPI Communicator Destructors. When an MPI process dies, its peers will likely need to free state that was associated with the dead process – including MPI communicators. MPI-1 section 5.4.3 defines MPI_COMM_FREE as a collective operation, meaning that by definition, MPI_COMM_FREE must fail on communicators that contain failed processes. But a reliable MPI implementation clearly must allow the successful completion of MPI_COMM_FREE on wounded communicators, or possibly face a deadlock situation during MPI_FINALIZE.

An "advice to implementors" section follows the definition of MPI_COMM_FREE in MPI-1, stating that MPI_COMM_FREE is normally expected to be implemented as a local operation, and therefore not require any communication. Given this stipulation, allowing MPI_COMM_FREE to free wounded communicators is clearly in the spirit of the original definition. Indeed, MPI_COMM_FREE can be considered an asynchronous collective – each process just happens no to notice the other failed processes in the communicator, and therefore completes successfully.

MPI Communicator Constructors. If surviving MPI processes need to use collective operations, wounded communicators must be subsetted to exclude the failed processes and create healthy communicators that can be used for future collective operations. MPI includes several communicator constructor functions, each of which are collective operations. By definition, they all must fail when used with a wounded source communicator. A reliable MPI implementation must therefore change the definition of at least one of the communicator constructors to allow creating healthy communicators from wounded communicators.

There are three constructor functions that could be modified to allow reliable behavior:

1. Changing MPI_COMM_DUP to allow copying wounded communicators would result in two wounded communicators, which is not useful.

2. MPI's rich set of algebraic group operations could be used to create an MPI group representing the set of surviving processes. This group could then be given to a modified MPI_COMM_CREATE to create a new communicator containing just the surviving set of processes. This is difficult because each surviving process must atomicly agree on exactly which processes are still alive when building the new group. As proven in [19], building such consensus in an asynchronous environment is impossible.

3. MPI_COMM_SPLIT is the functional opposite of MPI_COMM_CREATE – no group needs to be constructed ahead of time. Instead, each process that invokes MPI_COMM_SPLIT effectively says "I am alive." This is therefore a synchronization point between surviving members of the wounded communicator, and a distributed consensus of which processes are still alive can be achieved.

Hence, a reliable MPI extends the definition of MPI_COMM_SPLIT as follows:

- The `comm` argument may be a wounded communicator
- Failed processes will implicitly supply a `color` of MPI_UNDEFINED, and therefore not be included in any communicators returned on the surviving nodes.

Using these extended MPI functions, user applications that utilize collective operations can reliably continue in the presence of faults by creating new communicators. Figure 14 shows pseudocode using the

two extended functions to subset a wounded communicator into a new, healthy communicator. Note that the new communicator will potentially change the rank numbers of surviving MPI processes – the user application will need to map old rank numbers to new rank numbers.

```
MPI_Comm subset_comm(MPI_Comm orig_comm) {
  MPI_Comm_split(orig_comm, 0, 0, &new_comm);
  MPI_Free(&orig_comm);
  return new_comm;
}
```

Figure 14: A wounded communicator can be subsetted to create a new, healthy communicator with the extended definitions of MPI_COMM_SPLIT and MPI_COMM_FREE.

7.4 A Higher Abstraction Model

While a reliable MPI implementation provides a base layer of abstraction and the necessary functionality to implement a reliable MPI application, the data structure bookkeeping and maintenance necessary to effect the required programming models is both complex and repetitive. Details such as obtaining correct results from collective operations in an unreliable environment, automatic subsetting and/or re-launching of failed processes, and maintaining rank-to-process mappings even in the presence of process failures and communicator subsetting can all be maintained by a standalone library. Specifically, some higher-level abstractions are required to provide a reliable and easy-to-use basis for user applications.

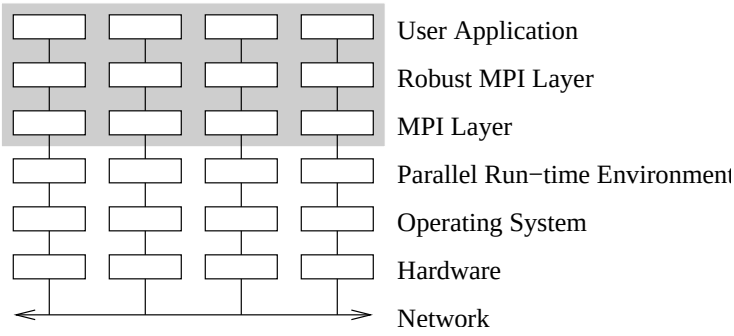

Figure 15: The robust MPI library exists as a layer between the user application and the native MPI library. Hence, the user application process on each node now spans three layers.

A library implemented on top of a reliable MPI implementation can provide both the required functionality while hiding the necessary data structure bookkeeping behind abstraction. As shown in Figure 15, this “robust MPI” library will exist as a layer between the user application and the native MPI library. This library may provide the following:

- An interface similar (if not identical) to MPI itself, thereby streamlining the process of adapting user applications to exploit robust functionality.

- Automatic detection and handling of most MPI exceptions, eliminating the need for laborious return code checking and error handling.
- Refined semantics of collective operations in MPI to effectively allow collective operations on communicators with failed processes by transparently providing services to rebuild communicators over the set of surviving processes (as described in Section 7.3.3).
- Bookkeeping services to maintain mappings of communicator/rank to MPI process, particularly when failed processes cause the underlying MPI to renumber ranks in a rebuilt communicator.
- Portability services that transparently enable robust behavior when used with reliable MPI implementations, and automatically disable most functionality (effectively becoming a thing passthru layer) when used with an unreliable MPI implementation.
- Full source code compatibility so that user applications can compile and link on multiple MPI implementations (regardless of whether the MPI implementation supports reliable behavior or not) with no code modifications.

This library will be a standalone component – separate from LAM/MPI. It will be fully compatible with any MPI implementation, although it may fall back to "abort on failure" behavior for unreliable MPI implementations. Hence, although robust behavior will be exhibited on reliable MPI implementations, user applications will be fully portable to any MPI implementation.

8 Requirements

The requirements outlined in this section allow for user-written MPI programs to execute in a reliable manner. They are broken down into three component areas: requirements for the LAM run-time environment, requirements for the MPI layer, and requirements for the user application.

8.1 Requirements for LAM

The LAM run-time environment requires several changes and new functional aspects in order to operate in a reliable manner.

1. The orderly shutdown of a `lamd` by any of the LAM utility commands listed below will not cause spurious asynchronous failure notices between peer `lamd` processes (any other method of shutting down a `lamd` is considered an "unexpected death"):
   (a) `lamhalt`
   (b) `wipe`
2. Each `lamd` must become aware of the unexpected death of any other `lamd` in the run-time environment. As a result of becoming aware of a peer `lamd`'s unexpected death, each `lamd` will:
   (a) Not leak memory or file descriptors
   (b) Continue correct operation of the overall run-time environment without causing deadlock or livelock

(c) Provide diagnostic information in the form of an error message giving, at a minimum, the name of the node where the `lamd` failed and the date/timestamp when the local `lamd` became aware of the failure

(d) Cause any pending and future operations with the failed `lamd` to fail and return appropriate error statuses

3. If any `lamd` cannot communicate with the `lamd` on the origin node,[9] it will kill all user applications on its node and then abort

4. The LAM utility commands will be hardened to allow for fault detection and recovery, including (but not limited to):

   (a) No LAM utility command may "hang" indefinitely; all operations/commands will complete within finite time

   (b) The `lamnodes` command will indicate which nodes have been marked "down" by the run-time environment

   (c) The `lamhalt` command will ensure that all nodes remaining in the run-time environment will kill themselves, even in the presence of failed nodes; if it cannot guarantee that all nodes have been shut down, a warning message will be displayed

5. When an MPI process dies and meets the criteria listed below, the local `lamd` will notify all of its peer MPI processes that it failed

   (a) The MPI process did not call MPI_FINALIZE

   (b) The MPI process was not killed by the `lamclean` command

8.2 Requirements for MPI

The MPI layer that is part of each user MPI process requires several changes and new functional aspects in order to operate in a reliable manner. All of the requirements listed below assume that user MPI applications are correct MPI programs, and, at a minimum, invoke MPI_INIT and MPI_FINALIZE.

The requirements below only apply to the MPI library layer in each of the user MPI processes. For example, even correct MPI programs can leak memory that the MPI library layer has no control over.

"Failed MPI processes" are defined as processes that terminate before calling MPI_FINALIZE.

1. The MPI layer in each MPI process will not leak memory or file descriptors, even in the presence of faults of peer MPI processes

2. The MPI library layer in each MPI process will be able to receive asynchronous notifications of unexpected MPI process failures from the local `lamd`

   (a) The asynchronous failure notifications will not affect the correctness of MPI functions

3. After the MPI layer in an MPI process has been notified that a peer MPI process has failed, all MPI operations involving communicaton with the failed process will fail immediately and return an error code that is not equal to MPI_SUCCESS

[9] The "origin node" in LAM terminology means the node that `lamboot` was invoked from.

(a) MPI operations involving a failed peer MPI process will not "hang" (deadlock or livelock)

4. The MPI library layer will provide infrastructure for MPI processes to optionally register a callback function in order to receive asynchronous notifications of unexpected MPI process failures

   (a) Callback functions will be associated with communicators

   (b) Callback functions will have, at a minimum, the same restrictions as Unix signal handlers

   (c) Callback functions will be invoked in each MPI process exactly once for each failed MPI process / communicator pair, where the communicator both contains the failed MPI process and registered a callback function

   (d) Callback functions are guaranteed to be invoked before the completion of an MPI_BARRIER on any communicator that includes a failed MPI process

5. The MPI library layer will be complaint with the MPI-1 and MPI-2 standards, except for the following:

   (a) The definition of MPI_COMM_SPLIT will be extended/changed per Section 7.3.3.

   (b) The definition of MPI_COMM_FREE will be extended/changed per Section 7.3.3.

6. User MPI processes will be able to perform a local operation on a communicator to obtain a list of ranks which correspond to the failed MPI processes in that communicator

7. The orderly shutdown of an MPI process by any of the MPI function calls listed below will not cause spurious asynchronous failure notices between peer MPI processes:

   (a) MPI_ABORT

   (b) MPI_FINALIZE

8. The `mpirun` command will provide additional infrastructure for reliable functionality

   (a) A command line switch will specify that a parallel MPI application should be run in "reliable" mode

   (b) When running in "reliable" mode, tolerate MPI process failures without killing the entire application

   (c) Process failure status information will be provided through optional error messages or `mpirun`'s Unix exit status

8.3 Requirements for User MPI Applications

User applications must be written specifically to allow continued operations in the presence of peer MPI process failures. The requirements listed below do not include discussion of fault recovery methods, such as user-level checkpointing, restarting user applications, etc.

1. User MPI applications will notify the MPI system to run in "reliable" mode[10]

2. User MPI applications will set the MPI error handler to an error handler other than MPI_ERRORS_ABORT on all communicators that require reliable operation

[10] The requirements in Section 8.2 dictate that this is actually a command line switch to `mpirun` and not included directly in the user program; it is mentioned here only because it is a necessary precondition for reliable behavior.

3. If required, user applications will register a callback function on relevant communicators to be asynchronously notified of peer MPI process failures
4. When notified of failures, the user application will subset wounded communicators with MPI_COMM_SPLIT to create healthy communicators if future collection operations are required
5. User applications will check the returned error code of all MPI functions and take appropriate recovery action if the returned error code is not MPI_SUCCESS

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhu, A. Skjellum, Y. Dandass, and M. Apte. MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance - Portable Parallel Computing. In George Mohay Rajkumar Buyya and Paul Roe, editors, *1st IEEE International Symposium of Cluster Computing and the Grid*, pages 26–33, Los Alamitos, CA, May 2001. IEEE Computer Society.

[3] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, S. Otto, and J. Walpole. PVM: Experiences, Current Status and Future Direction. In *Supercomputing'93 Proceedings*, pages 765–6, 1993.

[4] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *IEEE Computer*, 26(6):88–95, June 1993.

[5] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In John W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[6] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The Weakest Failure Detector for Solving Consensus. In Maurice Herlihy, editor, *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92)*, pages 147–158. ACM Press, 1992.

[7] F. Cristian. Reaching Agreement on Processor-Group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4:175–188, 1991.

[8] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proc. 15th Int. Symp. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206. IEEE Computer Society Press, 1985.

[9] J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. Integrated PVM Framework Supports Heterogeneous Network Computing. *Computers in Physics*, 7(2):166–75, April 1993.

[10] G. Fagg and a Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. EuroPVM/MPI User's Group Meeting 2000, Springer-Verilag, Berlin, Germany, 2000, pp.346-353.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: A Parallel Virtual Machine*. Scientific and Engineering Computation Series. MIT Press, 1994.

[12] W. Gropp and E. Lusk. Dynamic process management in an MPI setting. In *Proceedings / Seventh IEEE Symposium on Parallel and Distributed Processing, October 25–28, 1995, San Antonio, Texas*, pages 530–534. IEEE Computer Society Press, 1995.

[13] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[14] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, , and Marc Snir. *MPI — The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998.

[15] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[16] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[17] William D. Gropp and Ewing Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[18] Geoffrey James. *The Tao of Programming*. Info Books, Santa Monica, CA, 1987.

[19] N. A. Lynch M. J. Fisher and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.

[20] Gregory F. Pfister. *In Search of Clusters*. Prentice Hallt PTR, Upper Saddle River, NJ, 2 edition, 1998.

[21] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: An Extensible Toolkit for Low-Overhead Fault-Tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[22] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.

[23] Marc Snir, Steve W. Otto, Steve Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[24] G. Stellner. Cocheck: Checkpointing and Process Migration for MPI. In *Proc. of the Int'l Par. Proc. Symp.*, pages 526–531, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[25] T. Tannenbaum and M. Litzkow. The Condor Distributed Processing System. *Dr. Dobb's Journal*, (227):40–48, February 1995.

[26] The LAM Team. *Getting Started with LAM/MPI*. University of Notre Dame, Department of Computer Science, `http://www.lam-mpi.org/`, 1998.

[27] P. Urbán, X. Défago, and A. Schiper. Chasing the FLP Impossibility Result in a LAN or How Robust Can a Fault Tolerant Server Be? Technical Report DSC/2001/037, École Polytechnique Fédérale de Lausanne, Switzerland, August 2001.