

# MPI component support in CIFTS FTB

The CIFTS Team

Argonne National Laboratory  
Lawrence Berkeley National Laboratory  
Oak Ridge National Laboratory  
Indiana University  
Ohio State University  
University of Tennessee

August 3, 2010

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Fault Tolerance Backplane (FTB) support in Message Passing Interface (MPI)	3
<b>2</b>	<b>MPI FTB Events</b>	<b>4</b>
2.1	Node/Job/Rank Availability (Refer workflows in Appendix B.1, B.2)	4
2.2	Node/Job/Rank Failure Prediction	7
2.3	C/R and Process Migration (Refer workflows in Appendix B.3, B.4)	8
2.4	Checkpoint/Restart library notifications	10
2.5	Job status notifications	11
<b>A</b>	<b>Open MPI FTB Events</b>	<b>12</b>
<b>B</b>	<b>Open MPI FTB Workflows</b>	<b>14</b>
B.1	Workflow: Node Failure	16
B.1.1	Workflow: Node Failure Without Job	17
B.1.2	Workflow: Node Failure With MPI Job Aborting	18
B.1.3	Workflow: Node Failure With MPI Job Continuing	19
B.2	Workflow: Node Failure (No RM/JS)	20
B.2.1	Workflow: Node Failure With MPI Job Aborting	21
B.2.2	Workflow: Node Failure With MPI Job Continuing	22
B.3	Workflow: Checkpoint/Restart & Process Migration	23
B.3.1	Workflow: Gang Scheduling Support	23
B.3.2	Workflow: Predicted Failure, Job Suspend	24
B.3.3	Workflow: Predicted Failure, Process Migration	25
B.4	Workflow: Checkpoint/Restart & Process Migration (No RM/JS)	26
B.4.1	Workflow: Predicted Failure, Job Suspend	27
B.4.2	Workflow: Predicted Failure, Process Migration	28
B.5	Workflow: Faulty Interconnect	29
B.5.1	Workflow: Fail-over to an Alternative Device	30
B.5.2	Workflow: React to Corrupted or Missing Data	31
B.6	Workflow: Faulty Interconnect (No RM/JS)	32
B.6.1	Workflow: Fail-over to an Alternative Device	33
B.6.2	Workflow: React to Corrupted or Missing Data	34
<b>C</b>	<b>MPICH2 FTB Events</b>	<b>35</b>
<b>D</b>	<b>MVAPICH2 FTB Events</b>	<b>36</b>
<b>E</b>	<b>MVAPICH FTB Workflows</b>	<b>38</b>
E.1	Workflow: Process migration due to network failure	38
E.2	Workflow: Port Failover	38

## 1 Introduction

The Coordinated Infrastructure for Fault Tolerant Systems (CIFTS) Fault Tolerance Backplane (FTB) provides a shared software infrastructure that facilitates the exchange of fault-related information between various components that comprise the software stack of HEC systems. The primary motivation is to assist in making these software components fault-aware and preferably fault-tolerant, towards realizing the challenging goal of overall system resilience. The several library implementations of the Message Passing Interface (MPI) form a pivotal part of the software hierarchy of traditional HPC systems.

Based on the ideal workflows derived from common failure scenarios related to MPI, this document aims at standardizing a common set of events that every conforming MPI library implementation should support (either listen for, or respond with). The associated payloads corresponding to each FTB MPI event are also listed. This is to be considered as a starting point of discussion towards reaching a final consensus on the supported events and their representation. MPI/FTB developers are encouraged to propose addendum to this document to support any existing or additional failure scenarios.

### 1.1 FTB support in MPI

As a part of the CIFTS initiative, three existing MPI implementations, namely Open MPI, MPICH and MVAPICH, support interaction with the FTB in varying capacities. Each of the MPI libraries support different events, in some cases – common events with different severities or common events with different payloads. A FTB client interested in MPI-related fault information has to rely on implementation-specific details to effectively utilize this information. Thus, a FTB-aware user application written to deal with faults thrown by some MPI library X, would fail to even recognize these events or derive any semantic meaning out of them, if some other MPI library Y is used. Alternatively, the user application has to take into account all the events and their associated payloads, supported by all the FTB-supporting MPI libraries. It is important that we standardize, not only the the MPI FTB events and its payloads, but also the FTB events pertaining to other components like the scheduler and the resource manager to encompass a lot of common failure scenarios.

### Events

MPI FTB events are the failure/information events thrown or caught by the MPI library acting as a FTB client. These events are thrown under a common namespace (**ftb.mpi.\***) for the sake of offering consistent semantics to the clients interested in subscribing to MPI-related events with the FTB.

The list of events (derived from the corresponding common failure scenarios) are described in sections A, B, C, D, E of this document. Section 2 enlists the events with their associated payloads, that a MPI implementation must support in order to achieve full compliance with the FTB. These events subsume the events already supported by existing FTB-supporting MPI libraries. FTB allows each event to be accompanied by a user-specified event payload that can hold additional data associated with that event. MPI notification events typically need to convey additional information, for instance the “rank” of a dead node which forces us to standardize not only the events but also the the payload contents and its representation.

## 2 MPI FTB Events

The FTB offers two types of events: namely *Normal* events, which simply act as notifications and *Response* events which are a response or follow-up to a previously published event. With the introduction of reliable channels<sup>1</sup> to the FTB, applications can use the FTB, not only for notifications, but also for reliable invocations of services. Applications could implement “commands” over the FTB. For instance, an application wishing to know about a particular node failure can publish the corresponding event and wait for follow-up notifications from the other FTB-enabled components in the system affirming the unavailability of the node.

The following list of events are classified according to the different scenarios in which they occur and cross-referenced with their corresponding workflows. Events prefixed with **CHECK** are command events which almost always expect a follow-up notification event. These events are thus sent over a reliable channel. Events prefixed with **DO** are command events but do not usually depend much on a follow-up event. They perform a reliable invocation of a service in response to some other events. They are also typically sent over the reliable channel. All other events are simply “notificational” events and can be sent over the regular unreliable channel. Each FTB-enabled component prefixes the event with its own namespace identifier. Job schedulers (JS) and resource managers (RM) prefix events with **RM** whereas the MPI library and/or the MPI runtime throw events with the **MPI** prefix. Finally, events with the **CONFIRM** prefix are generic response events to the corresponding **CHECK** or **DO** events. Dedicated agents or external fault detectors could extrapolate information from the notifications they receive from individual components and throw these more general **CONFIRM** events.

The payload representation is in the form of comma-separated key-value pairs and follows standard set notation. Note that payload entries within square brackets [ ] are optional.

### 2.1 Node/Job/Rank Availability (Refer workflows in Appendix B.1, B.2)

	<b>FTB Event</b>	<b>Thrown by</b>	<b>Caught by</b>	<b>Channel</b>
1	CHECK_NODES_DEAD	Monitoring system, Application	External agent, RM/JS, Autonomic script	Reliable

*Nodes  $N^+$  are suspected as unavailable. Check if nodes  $N^+$  are really dead. Since the component throwing this event expects a response from the components across the other layers, this event has to be sent over a reliable channel.*

#### **Payload**

nodes:  $\{N^+\}$

2	CONFIRM_NODES_DEAD	External agent, Autonomic script	Monitoring system, Application	Unreliable
---	--------------------	----------------------------------	--------------------------------	------------

*Confirm that nodes  $N^+$  are indeed dead. This notification event can be caught by any component including the application itself. It is typically thrown by an external agent after it ascertains that a node is really dead.*

<sup>1</sup>See the discussion at <http://wiki.mcs.anl.gov/cifts/index.php/Reliability>

**Payload**nodes:  $\{N^+\}$ 

- 
- |   |                     |                                  |                                |            |
|---|---------------------|----------------------------------|--------------------------------|------------|
| 3 | CONFIRM.NODES_ALIVE | External agent, Autonomic script | Monitoring system, Application | Unreliable |
|---|---------------------|----------------------------------|--------------------------------|------------|

*Confirm that nodes  $N^+$  are alive. This event is thrown by the external agent or the monitoring system in response to CHECK\_NODES\_DEAD if the node is responsive. It can also be thrown as a purely notifiational event which adds the nodes to the unallocated resource pool if they were previously identified as dead.*

**Payload**nodes:  $\{N^+\}$ 

- 
- |   |              |       |                                       |          |
|---|--------------|-------|---------------------------------------|----------|
| 4 | RM.JOBS_DEAD | RM/JS | MPI, External agent, Autonomic script | Reliable |
|---|--------------|-------|---------------------------------------|----------|

*Confirm that jobs  $J^+$  running on nodes  $N^+$  are dead. This event could be thrown by the RM/JS in response to the CHECK\_NODES\_DEAD event.*

**Payload**nodes:  $\{N^+\}$ , jobs:  $\{J^+\}$ 

- 
- |   |                 |       |                                       |          |
|---|-----------------|-------|---------------------------------------|----------|
| 5 | RM.JOBS_RUNNING | RM/JS | MPI, External agent, Autonomic script | Reliable |
|---|-----------------|-------|---------------------------------------|----------|

*Confirm that jobs  $J^+$  running on nodes  $N^+$  are running.*

**Payload**nodes:  $\{N^+\}$ , jobs:  $\{J^+\}$ 

- 
- |   |                 |                                |   |          |
|---|-----------------|--------------------------------|---|----------|
| 6 | CHECK_JOBS_DEAD | Monitoring system, Application | RM/JS, External agent, Autonomic script | Reliable |
|---|-----------------|--------------------------------|---|----------|

*Jobs  $J^+$  are suspected as unavailable. Check if jobs  $J^+$  are really dead.*

**Payload**jobs:  $\{J^+\}$ 

- 
- |   |                   |   |                                |            |
|---|-------------------|---|--------------------------------|------------|
| 7 | CONFIRM.JOBS_DEAD | RM/JS, External agent, Autonomic script | Monitoring system, Application | Unreliable |
|---|-------------------|---|--------------------------------|------------|

*Confirm that jobs  $\mathbf{J}^+$  are indeed dead. This notification event can be caught by any component including the application itself. It is typically thrown by an external detector after making sure that the jobs are dead.*

**Payload**

jobs:  $\{J^+\}$

---

8	CONFIRM_JOBS_RUNNING	RM/JS, agent, script	External Monitoring system, Autonomic Application	Unreliable
---	----------------------	----------------------	---	------------

*Confirm that jobs  $\mathbf{J}^+$  are running. This notification event is typically thrown by the RM/JS if the jobs are running.*

**Payload**

jobs:  $\{J^+\}$

---

9	MPI_RANKS_DEAD	MPI library/runtime	External agent, Monitoring system	Unreliable
---	----------------	---------------------	-----------------------------------	------------

*Confirm that MPI ranks  $\mathbf{R}^+$  belonging to jobs  $\mathbf{J}^+$  running on nodes  $\mathbf{N}^+$  are dead. The MPI runtime (usually mpirun) throws this event in response to either CHECK\_NODES\_DEAD (when the MPI runtime can form a relation between the nodes and corresponding ranks) or CHECK\_JOBS\_DEAD or simply as a notification.*

**Payload**

nodes:  $\{N^+\}$ , [ jobs:  $\{J^+\}$ , ] ranks:  $\{R^+\}$

---

10	MPI_RANKS_ALIVE	MPI library/runtime	External agent, Monitoring system	Unreliable
----	-----------------	---------------------	-----------------------------------	------------

*Confirm that MPI ranks  $\mathbf{R}^+$  belonging to jobs  $\mathbf{J}^+$  running on nodes  $\mathbf{N}^+$  are alive. The MPI runtime (usually mpirun) throws this event in response to either CHECK\_NODES\_DEAD (when the MPI runtime can form a relation between the nodes and corresponding ranks) or CHECK\_JOBS\_DEAD or simply as a notification.*

**Payload**

nodes:  $\{N^+\}$ , [ jobs:  $\{J^+\}$ , ] ranks:  $\{R^+\}$

---

11	CHECK_RANKS_DEAD	Monitoring system, Application	External agent, Autonomic script, MPI	Reliable
----	------------------	--------------------------------	---------------------------------------	----------

*Ranks  $\mathbf{R}^+$  are suspected as unavailable. Check if ranks  $\mathbf{R}^+$  are really dead.*

**Payload**ranks:  $\{R^+\}$ 


---

12	CONFIRM_RANKS_DEAD	MPI, External agent, Autonomic script	Monitoring system, Application	Unreliable
----	--------------------	--	-----------------------------------	------------

*Confirm that ranks  $R^+$  are indeed dead. This event is typically thrown by MPI or an external detector after making sure that the ranks are dead.*

**Payload**ranks:  $\{R^+\}$ 


---

13	CONFIRM_RANKS_ALIVE	MPI, External agent, Autonomic script	Monitoring system, Application	Unreliable
----	---------------------	--	-----------------------------------	------------

*Confirm that ranks  $R^+$  are alive. This event is thrown by MPI or an external detector after making sure that the ranks are alive.*

**Payload**ranks:  $\{R^+\}$ **2.2 Node/Job/Rank Failure Prediction**


---

	FTB Event	Thrown by	Caught by	Channel
1	PREDICT_NODES_FAILURE	External agent, Au- tonomic script	RM/JS, MPI, Moni- toring system	Unreliable

*Depending on collected heuristics, predict imminent failure for nodes  $N^+$ .*

**Payload**nodes:  $\{N^+\}$ 


---

2	PREDICT_JOBS_FAILURE	RM/JS	MPI, External agent, Autonomic script	Unreliable
---	----------------------	-------	--	------------

*Depending on collected heuristics, predict imminent failure for jobs  $J^+$  running on nodes  $N^+$ .*

**Payload**nodes:  $\{N^+\}$ , jobs:  $\{J^+\}$ 


---

3	PREDICT_RANKS_FAILURE	MPI	External agent, Au- tonomic script, Ap- plication	Unreliable
---	-----------------------	-----	---	------------

Depending on collected heuristics, predict imminent failure for ranks  $\mathbf{R}^+$  spanning jobs  $\mathbf{J}^+$  running on nodes  $\mathbf{N}^+$ .

**Payload**

nodes:  $\{N^+\}$ , [jobs:  $\{J^+\}$ ], ranks:  $\{R^+\}$

## 2.3 C/R and Process Migration (Refer workflows in Appendix B.3, B.4)

	FTB Event	Thrown by	Caught by	Channel
1	DO_NODE_MIGRATE	External agent, Autonomic script	RM/JS, MPI	Reliable

*This event is thrown by the external agent or the autonomic script after it predicts a node failure. It involves the migration of all the active services from the source node to the destination node. Each component subscribed to this event further throws events to migrate its resources from the dying node. For instance, a RM/JS catching this event throws a follow-up event to migrate all the jobs associated with this node to other node(s). Note that the destination node field in the payload is optional. If no destination node is specified, the component has to “figure out” an unallocated destination node by external means.*

**Payload**

snode:  $SN$  [, dnode:  $DN$ ]

2	RM_JOBS_MIGRATE	RM/JS	MPI, External agent, Autonomic script	Reliable
---	-----------------	-------	---------------------------------------	----------

*This is a command event thrown by the RM/JS to initiate a request to migrate jobs  $\mathbf{J}^+$  from source node  $\mathbf{SN}$  to destination node  $\mathbf{DN}$ .*

**Payload**

snode:  $SN$ , dnode:  $DN$ , jobs:  $\{J^+\}$

3	MPI_RANKS_MIGRATE	MPI	External agent, Autonomic script, Application	Unreliable
---	-------------------	-----	---	------------

*This event is thrown by MPI after migrating ranks  $\mathbf{R}^+$  belonging to jobs  $\mathbf{J}^+$  from the source node  $\mathbf{SN}$  to destination node  $\mathbf{DN}$ .*

**Payload**

snode:  $SN$ , dnode:  $DN$ , [jobs:  $\{J^+\}$ ], ranks:  $\{R^+\}$

---

4	CONFIRM_NODE_MIGRATE	External agent, Autonomous script	Monitoring system, Application	Unreliable
<i>Confirm that all the active services have been migrated from the source node <b>SN</b> to destination node <b>DN</b>. Note that the destination node is no longer optional in the payload.</i>				
<b>Payload</b>				
snode: <i>SN</i> , dnode: <i>DN</i>				
<hr/>				
5	DO_JOBS_SUSPEND	RM/JS	External agent, Autonomous script, MPI	Reliable
<i>This is a command event that initiates a request to suspend jobs <b>J</b><sup>+</sup>.</i>				
<b>Payload</b>				
jobs: { <i>J</i> <sup>+</sup> }				
<hr/>				
6	MPI_RANKS_SUSPEND	MPI	External agent, Autonomous script	Unreliable
<i>This event is a notification event thrown by MPI after it suspends (mostly by taking a checkpoint) ranks <b>R</b><sup>+</sup> belonging to jobs <b>J</b><sup>+</sup>.</i>				
<b>Payload</b>				
jobs: { <i>J</i> <sup>+</sup> }, ranks: { <i>R</i> <sup>+</sup> }				
<hr/>				
7	CONFIRM_JOBS_SUSPEND	External agent, Autonomous script	Monitoring system, Application	Unreliable
<i>Confirm that all the jobs <b>J</b><sup>+</sup> have been suspended.</i>				
<b>Payload</b>				
jobs: { <i>J</i> <sup>+</sup> }				
<hr/>				
8	DO_JOBS_RESUME	RM/JS	External agent, Autonomous script, MPI	Reliable
<i>This is a command event that initiates a request to resume jobs <b>J</b><sup>+</sup>.</i>				
<b>Payload</b>				
jobs: { <i>J</i> <sup>+</sup> }				
<hr/>				
9	MPI_RANKS_RESUME	MPI	External agent, Autonomous script	Unreliable

*This event is a notification event thrown by MPI after it resumes (mostly by initiating a restart) ranks  $\mathbf{R}^+$  belonging to jobs  $\mathbf{J}^+$ .*

**Payload**

jobs:  $\{J^+\}$ , ranks:  $\{R^+\}$

---

10	CONFIRM_JOBS_RESUME	External agent, Autonomic script	Monitoring system, Application	Unreliable
----	---------------------	----------------------------------	--------------------------------	------------

*Confirm that all the jobs  $\mathbf{J}^+$  have been resumed and in a running state.*

**Payload**

jobs:  $\{J^+\}$

---

11	MPI_RANKS_CKPT	MPI	External agent, Autonomic script	Unreliable
----	----------------	-----	----------------------------------	------------

*This event is a notification event thrown by MPI after it checkpoints the ranks  $\mathbf{R}^+$ .*

**Payload**

ranks:  $\{R^+\}$

---

12	MPI_RANKS_RESTART	MPI	External agent, Autonomic script	Unreliable
----	-------------------	-----	----------------------------------	------------

*This event is a notification event thrown by MPI after it restarts previously checkpointed ranks  $\mathbf{R}^+$ .*

**Payload**

ranks:  $\{R^+\}$

## 2.4 Checkpoint/Restart library notifications

---

	FTB Event	Thrown by	Caught by	Channel
1	CR_CKPTS_BEGIN	C/R library	MPI, Application, External agent	Unreliable

*This is a notification event to indicate that the C/R library has begun checkpointing processes  $\mathbf{P}^+$ .*

**Payload**

processes:  $\{P^+\}$

---

2	CR_CKPTS_END	C/R library	MPI, Application, External agent	Unreliable
---	--------------	-------------	----------------------------------	------------

*This is a notification event to indicate that the C/R library has finished checkpointing processes  $P^+$ .*

**Payload**

processes:  $\{P^+\}$

---

3	CR_RESTARTS_BEGIN	C/R library	MPI, Application, External agent	Unreliable
---	-------------------	-------------	-------------------------------------	------------

*This is a notification event to indicate that the C/R library has begun restarting processes  $P^+$ .*

**Payload**

processes:  $\{P^+\}$

---

4	CR_RESTARTS_END	C/R library	MPI, Application, External agent	Unreliable
---	-----------------	-------------	-------------------------------------	------------

*This is a notification event to indicate that the C/R library has finished restarting processes  $P^+$ .*

**Payload**

processes:  $\{P^+\}$

## 2.5 Job status notifications

---

	FTB Event	Thrown by	Caught by	Channel
1	RM_JOBS_CREATED	RM/JS	MPI, External agent, Autonomic script	Reliable

*New jobs  $J^+$  have been created. This event could be thrown by the RM/JS as a notification event.*

**Payload**

jobs:  $\{J^+\}$

---

2	RM_JOBS_COMPLETED	RM/JS	MPI, External agent, Autonomic script	Reliable
---	-------------------	-------	--	----------

*Jobs  $J^+$  have completed. This event could be thrown by the RM/JS as a notification event.*

**Payload**

jobs:  $\{J^+\}$

---

## A Open MPI FTB Events

The FTB notifier component throws events in the event namespace `ftb.mpi.openmpi`. At the moment, it does not subscribe to or act upon any events from the FTB.

The supported FTB publishable events for Open MPI are listed in Table 1. These events are mapped to the corresponding error codes in the ORTE framework. The FTB notifier component translates the ORTE error codes to a corresponding FTB event and publishes them to the FTB. For instance, the OpenIB BTL throws an event `COMM_FAILURE` when the InfiniBand retry count between two MPI processes exceeds a specified threshold. Since all the supported events are normal events, they have an `'error_type'` as 1. These events also have an associated payload that describes the cause of the error.

Table 1: Supported publishable events for the OMPI FTB component

<b>FTB Event</b>	<b>Severity</b>	<b>Corresponding ORTE error</b>
UNKNOWN_ERROR	error	
OUT_OF_RESOURCES	error	ORTE_ERR_OUT_OF_RESOURCE ORTE_ERR_TEMP_OUT_OF_RESOURCE
UNREACHABLE	error	ORTE_ERR_CONNECTION_REFUSED ORTE_ERR_CONNECTION_FAILED ORTE_ERR_UNREACH
COMM_FAILURE	error	ORTE_ERR_COMM_FAILURE
FATAL	fatal	ORTE_ERR_FATAL

### Possible Events

This section lists the possible events likely to be supported by the Open MPI FTB component in future. These events might most likely change per community feedback and specific requirements of components relying on Open MPI for fault information. Table 2 lists the FTB events that the component might want to subscribe to whereas Table 3 lists the possible publishable FTB events for the OMPI-FTB component.

For a general overview of how these events coordinate with the other FTB-enabled components, please refer to the IU CIFTS workflows.

Table 2: Possible subscribable events for the OMPI FTB module

<b>FTB Event</b>	<b>Type</b>	<b>Description</b>
NODE_DEAD	response	Check if node X is unreachable
MPLNODE_DEAD	normal	Node X is unreachable
NODE_RESTORED	response	Add node X as an unallocated resource
MPLNODE_RESTORED	normal	Return node X to the available resource pool
NODE_MIGRATE	response	Migrate all ranks from Node X to Node Q
JOB_ABORT	response	Suspend or terminate job Z
JOB_RESUME	normal	Bring back job Z to a running state
IFACE_DEAD	response	Physical interface P has failed
IFACE_RESTORED	normal	Physical interface P is back to service
MPLMSG_CORRUPT	normal	Message corruption on interface P

Table 3: Possible publishable events for the OMPI FTB module

<b>FTB Event</b>	<b>Severity</b>	<b>Description</b>
MPLINIT	info	Initialize the MPI execution environment
MPLFINALIZE	info	Finalize the MPI execution environment
MPLNODE_DEAD	error	Node X is dead
MPLNODE_RESTORED	info	Node X is back to service
MPLRANK_DEAD	error	Rank Y (on Node X) is presumably dead
MPLRANK_RESTORED	info	Rank Y (on Node X) is back to service
MPLNODE_MIGRATE_DONE	info	Ranks migrated from Node X to Node Q
MPLJOB_ABORT_CMD	error	Command to abort MPI Job Z
MPLJOB_RESUME_CMD	info	Command to resume MPI Job Z
MPLJOB_ABORT	error	MPI Job Z has been aborted
MPLJOB_RESUME	info	MPI Job Z has been resumed
MPLMSG_CORRUPT	error	Message corruption on interface P
MPLIFACE_DEAD	error	Mark physical interface P as dead
MPLIFACE_RESTORED	error	Add P to available physical interfaces

## B Open MPI FTB Workflows

We believe that many of the workflows in the proposal are still applicable, but with some slight modifications. Most of the workflow discussion in this document focuses on the role of MPI (particularly Open MPI). Further iterations are needed to refine these workflows such that they are correct for other components of the CIFTS FTB.

For each workflow below we have also attempted to provide an alternative workflow that work around non-FTB-aware RM/JS. To do so we rely on a `mpirun` process that has knowledge of the global state of the MPI job and available resources. Open MPI has such a process that usually resides on the login node when a new job is launched.

In the non-FTB-aware RM/JS scenarios we chose to pass the resource management and event escalation duties to the `mpirun` process. We do this because the `mpirun` process has global knowledge of the running processes and available resources to its job, and can more abstractly escalate events than, say, the application. Normally we prefer to use an FTB-aware RM/JS since it may handle non-MPI jobs and has the potential to do more interesting actions in response to failure since it has knowledge beyond that of a single job.

In the event that the MPI *and* the RM/JS are non-FTB-aware, the application will have to assume the responsibilities of the RM/JS and MPI to whatever degree that it is able. This is following a general rule that the responsibility of resource management and event escalation should be pushed to the FTB-aware component with (a) the largest amount of global knowledge regarding resource availability, and (b) significantly low-level enough to meaningfully escalate events originating from specific devices (e.g., Escalating “IB NIC failure” to “Node failure” to “MPI Rank x,y,z Failed”).

**NOTE:** For the Workflows that attempt to work around a non-FTB-aware RM/JS, *MPI\** denotes `mpirun` playing the role of RM/JS. In some cases *MPI\** is throwing events that are only caught by *MPI* (and visa versa). The reason for this is to maintain generality so that when a FTB-aware RM/JS is plugged in then the MPI implementation should be able to more easily transition to using it.

### Workflow: Node Failure

All of these workflows detail a response to a detected node failure.

- Section B.1 Details the registered events for various components.
- Section B.1.1 Node failure without a job
- Section B.1.2 Node failure with MPI job aborting
- Section B.1.3 Node failure with MPI job continuing

### Workflow: Node Failure (No RM/JS)

All of these workflows detail a response to a detected node failure. They also work around the need for a FTB aware RM/JS by relying on the `mpirun` process to handle many of these activities. Note the Section B.1.1 has no corresponding section here since MPI is not involved.

- Section B.2 Details the registered events for various components.
- Section B.2.1 Node failure with MPI job aborting
- Section B.2.2 Node failure with MPI job continuing

### **Workflow: Checkpoint/Restart & Process Migration**

All of these workflows detail a response to a predicted node failure. So with advance notice of a failure, preventative actions are triggered to mitigate the impact of the failure. Additionally a RM/JS might wish to trigger a checkpoint to provide a coarse-grained, gang scheduling type of functionality.

- Section B.3 Details the registered events for various components.
- Section B.3.1 Gang Scheduling Support
- Section B.3.2 Predicted node failure, resulting in a full job suspension/shutdown
- Section B.3.3 Predicted node failure, resulting in process migration

### **Workflow: Checkpoint/Restart & Process Migration (No RM/JS)**

All of these workflows detail a response to a predicted node failure. So with advance notice of a failure, preventative actions are triggered to mitigate the impact of the failure. They also work around the need for a FTB aware RM/JS by relying on the `mpirun` process to handle many of these activities. Section B.3.1 is not represented since MPI is not able to control multiple 'jobs'.

- Section B.4 Details the registered events for various components.
- Section B.4.1 Predicted node failure, resulting in a full job suspension/shutdown
- Section B.4.2 Predicted node failure, resulting in process migration

### **Workflow: Interconnect Failure**

All of these workflows detail a response to a faulty interconnect.

- Section B.5 Details the registered events for various components.
- Section B.5.1 Fail-over to an alternative device.
- Section B.5.2 React to corrupted or missing data

### **Workflow: Interconnect Failure (No RM/JS)**

All of these workflows detail a response to a faulty interconnect. They also work around the need for a FTB aware RM/JS by relying on the `mpirun` process to handle many of these activities.

- Section B.6 Details the registered events for various components.
- Section B.6.1 Fail-over to an alternative device.
- Section B.6.2 React to corrupted or missing data

### **Workflow: Task Farm**

The task farm workflow concerns an MPI application that operates in a manager/worker model. This workflow still needs to be more concretely specified in a later draft.

## B.1 Workflow: Node Failure

The following table details the events that each component will want to either throw or catch.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Initialization &amp; Job Launch</i>		
0	RM/JS	Register	Check Problem Node (node *)
0	RM/JS	Register	Dead Physical Node (node *)
0	RM/JS	Register	Dead MPI Node (node *: job z)
0	RM/JS	Register	Restored Node (node *)
0	RM/JS	Register	Restored MPI Node (node *: job z)
0	Monitoring System	Register	Check Problem Node (node *)
0	Monitoring System	Register	Dead Physical Node (node *)
0	Monitoring System	Register	Restored Node (node *)
0	Autonomic Script	Register	Check Problem Node (node *)
0	Autonomic Script	Register	Dead Physical Node (node *)
0	MPI	Register	Dead MPI Node (node *: job z)
0	MPI	Register	Restored MPI Node (node *: job z)
0	MPI	Register	Dead MPI Rank (node x: job z: rank n-m)
0	Application	Register	Dead MPI Rank (node x: job z: rank n-m)
0	Application	Register	Restored MPI Node (node x: job z)

### B.1.1 Workflow: Node Failure Without Job

A node failure can occur without any jobs running on the failed node.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Node x Fails, no job running on node x</i>		
1	Monitoring System	Throw	Check Problem Node (node x)
	<i>Suspect problem with node x</i>		
2(a)	RM/JS	Catch	Check Problem Node (node x)
	<i>Suspend scheduling on node x (suspect failure)</i>		
2(b)	Autonomic Script	Catch	Check Problem Node (node x)
	<i>Attempt to confirm node x failed</i>		
3	Autonomic Script	Throw	Dead Physical Node (node x)
	<i>Confirmed node x failed</i>		
4(a)	RM/JS	Catch	Dead Physical Node (node x)
	<i>Remove node x from resource pool</i>		
4(b)	Monitoring System	Catch	Dead Physical Node (node x)
	<i>Remove node x from set of monitored resources</i>		
4(c)	Autonomic Script	Catch	Dead Physical Node (node x)
	<i>Notify sysadmin, trigger full diagnosis after hard reboot</i>		
	<i>Archives system logs, begin stress test, bring online spare nodes</i>		
	<i>Refund CPU accounting units, reschedule job</i>		
	<i>Time passes, machine returned to service</i>		
5	Autonomic Script	Throw	Restored Node (node x)
	<i>Sysadmin uses script to notify services of node recovery</i>		
6(a)	RM/JS	Catch	Restored Node (node x)
	<i>Return node x to resource pool</i>		
6(b)	Monitoring System	Catch	Restored Node (node x)
	<i>Return node x to the set of monitored resources</i>		

### B.1.2 Workflow: Node Failure With MPI Job Aborting

A node failure occurs while a job is running on the failed node. The policy expressed by the application through the MPI interface is that the MPI abort on such a failure.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Node x Fails, job z running on allocation including node x</i>		
1	Monitoring System	Throw	Check Problem Node (node x)
	<i>Suspect problem with node x</i>		
2(a)	RM/JS	Catch	Check Problem Node (node x)
	<i>Mark node x as (suspect failure)</i>		
2(b)	Autonomic Script	Catch	Check Problem Node (node x)
	<i>Attempt to confirm node x failed</i>		
3	Autonomic Script	Throw	Dead Physical Node (node x)
	<i>Confirmed node x failed</i>		
4(a)	RM/JS	Catch	Dead Physical Node (node x)
	<i>Remove node x from resource pool</i>		
4(b)	Monitoring System	Catch	Dead Physical Node (node x)
	<i>Remove node x from set of monitored resources</i>		
4(c)	Autonomic Script	Catch	Dead Physical Node (node x)
	<i>Notify sysadmin, trigger full diagnosis after hard reboot</i>		
	<i>Archives system logs, begin stress test, bring online spare nodes</i>		
	<i>Refund CPU accounting units, reschedule job</i>		
5	RM/JS	Throw	Dead MPI Node (node x: job z)
	<i>Translates node x to job z</i>		
6	MPI	Catch	Dead MPI Node (node x: job z)
	<i>MPI prints console error, aborts job z</i>		
	<i>Time passes, machine returned to service</i>		
7	Autonomic Script	Throw	Restored Node (node x)
	<i>Sysadmin uses script to notify services of node recovery</i>		
8(a)	RM/JS	Catch	Restored Node (node x)
	<i>Return node x to unallocated resource pool</i>		
8(b)	Monitoring System	Catch	Restored Node (node x)
	<i>Return node x to the set of monitored resources</i>		

### B.1.3 Workflow: Node Failure With MPI Job Continuing

A node failure occurs while a job is running on the failed node. Node failure policy is that MPI should continue with holes in communicators. Node recovery policy is that MPI adds resources to internal pool to support application directed re-spawning of processes.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Node x Fails, job z running on allocation including node x</i>		
1	Monitoring System	Throw	Check Problem Node (node x) <i>Suspect problem with node x</i>
2(a)	RM/JS	Catch	Check Problem Node (node x) <i>Mark node x as (suspect failure)</i>
2(b)	Autonomic Script	Catch	Check Problem Node (node x) <i>Attempt to confirm node x failed</i>
3	Autonomic Script	Throw	Dead Physical Node (node x) <i>Confirmed node x failed</i>
4(a)	RM/JS	Catch	Dead Physical Node (node x) <i>Remove node x from resource pool</i>
4(b)	Monitoring System	Catch	Dead Physical Node (node x) <i>Remove node x from set of monitored resources</i>
4(c)	Autonomic Script	Catch	Dead Physical Node (node x) <i>Notify sysadmin, trigger full diagnosis after hard reboot</i> <i>Archives system logs, begin stress test, bring online spare nodes</i> <i>Refund CPU accounting units, reschedule job</i>
5	RM/JS	Throw	Dead MPI Node (node x: job z) <i>Translates node x to job z</i>
6	MPI	Catch	Dead MPI Node (node x: job z) <i>Translate (node x:job z) to ranks m-n</i> <i>Replace ranks m-n with MPI_PROC_NULL, call application error handlers</i>
7	MPI	Throw	Dead MPI Rank (node x: job z: rank n-m) <i>Translate (node x:job z) to ranks m-n</i>
8	Application	Catch	Dead MPI Rank (node x: job z: rank n-m) <i>Work around 'blank' ranks n-m in the MPI communicators</i>
	<i>Time passes, machine returned to service</i>		
9	Autonomic Script	Throw	Restored Node (node x) <i>Sysadmin uses script to notify services of node recovery</i>
10(a)	RM/JS	Catch	Restored Node (node x) <i>Return node x to resource pool for job z</i>
10(b)	Monitoring System	Catch	Restored Node (node x) <i>Return node x to the set of monitored resources</i>
11	RM/JS	Throw	Restored MPI Node (node x: job z) <i>Translates node x to job z</i>
12(a)	MPI	Catch	Restored MPI Node (node x: job z) <i>Add node x as an unallocated resource</i>
12(b)	Application	Catch	Restored MPI Node (node x: job z) <i>If needed, use MPI_Comm_spawn to create new processes</i>

## B.2 Workflow: Node Failure (No RM/JS)

The following table details the events that each component will want to either throw or catch. *MPI\** denotes *mpirun* playing the role of RM/JS.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Initialization &amp; Job Launch</i>		
0	MPI*	Register	Check Problem Node (node *)
0	MPI*	Register	Dead Physical Node (node *)
0	MPI*	Register	Dead MPI Node (node *: job z)
0	MPI*	Register	Restored Node (node *)
0	MPI*	Register	Restored MPI Node (node *: job z)
0	Monitoring System	Register	Check Problem Node (node *)
0	Monitoring System	Register	Dead Physical Node (node *)
0	Monitoring System	Register	Restored Node (node *)
0	Autonomic Script	Register	Check Problem Node (node *)
0	Autonomic Script	Register	Dead Physical Node (node *)
0	MPI	Register	Dead MPI Node (node *: job z)
0	MPI	Register	Restored MPI Node (node *: job z)
0	MPI	Register	Dead MPI Rank (node x: job z: rank n-m)
0	Application	Register	Dead MPI Rank (node x: job z: rank n-m)
0	Application	Register	Restored MPI Node (node x: job z)

### B.2.1 Workflow: Node Failure With MPI Job Aborting

A node failure occurs while a job is running on the failed node. The policy expressed by the application through the MPI interface is that the MPI abort on such a failure.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Node x Fails, job z running on allocation including node x</i>		
1	Monitoring System	Throw	Check Problem Node (node x)
	<i>Suspect problem with node x</i>		
2(a)	MPI*	Catch	Check Problem Node (node x)
	<i>Mark node x as (suspect failure)</i>		
2(b)	Autonomic Script	Catch	Check Problem Node (node x)
	<i>Attempt to confirm node x failed</i>		
3	Autonomic Script	Throw	Dead Physical Node (node x)
	<i>Confirmed node x failed</i>		
4(a)	MPI*	Catch	Dead Physical Node (node x)
	<i>Remove node x from internal resource pool</i>		
4(b)	Monitoring System	Catch	Dead Physical Node (node x)
	<i>Remove node x from set of monitored resources</i>		
4(c)	Autonomic Script	Catch	Dead Physical Node (node x)
	<i>Notify sysadmin, trigger full diagnosis after hard reboot</i>		
	<i>Archives system logs, begin stress test, bring online spare nodes</i>		
	<i>Refund CPU accounting units, reschedule job</i>		
5	MPI*	Throw	Dead MPI Node (node x: job z)
	<i>Translates node x to job z (value stored internally)</i>		
6	MPI	Catch	Dead MPI Node (node x: job z)
	<i>MPI prints console error, aborts job z</i>		
	<i>Time passes, machine returned to service</i>		
7	Autonomic Script	Throw	Restored Node (node x)
	<i>Sysadmin uses script to notify services of node recovery</i>		
	<i>Sysadmin manually adds node to RM/JS</i>		
8	Monitoring System	Catch	Restored Node (node x)
	<i>Return node x to the set of monitored resources</i>		

### B.2.2 Workflow: Node Failure With MPI Job Continuing

A node failure occurs while a job is running on the failed node. Node failure policy is that MPI should continue with holes in communicators. Node recovery policy is that MPI adds resources to internal pool to support application directed re-spawning of processes.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Node x Fails, job z running on allocation including node x</i>		
1	Monitoring System	Throw	Check Problem Node (node x) <i>Suspect problem with node x</i>
2(a)	MPI*	Catch	Check Problem Node (node x) <i>Mark node x as (suspect failure)</i>
2(b)	Autonomic Script	Catch	Check Problem Node (node x) <i>Attempt to confirm node x failed</i>
3	Autonomic Script	Throw	Dead Physical Node (node x) <i>Confirmed node x failed</i>
4(a)	MPI*	Catch	Dead Physical Node (node x) <i>Remove node x from internal resource pool</i>
4(b)	Monitoring System	Catch	Dead Physical Node (node x) <i>Remove node x from set of monitored resources</i>
4(c)	Autonomic Script	Catch	Dead Physical Node (node x) <i>Notify sysadmin, trigger full diagnosis after hard reboot</i> <i>Archives system logs, begin stress test, bring online spare nodes</i> <i>Refund CPU accounting units, reschedule job</i>
5	MPI*	Throw	Dead MPI Node (node x: job z) <i>Translates node x to job z (value stored internally)</i>
6	MPI	Catch	Dead MPI Node (node x: job z) <i>Translate (node x:job z) to ranks m-n</i> <i>Replace ranks m-n with MPI_PROC_NULL, call application error handlers</i>
7	MPI	Throw	Dead MPI Rank (node x: job z: rank n-m) <i>Translate (node x:job z) to ranks m-n</i>
8	Application	Catch	Dead MPI Rank (node x: job z: rank n-m) <i>Work around 'blank' ranks n-m in the MPI communicators</i>
	<i>Time passes, machine returned to service</i>		
9	Autonomic Script	Throw	Restored Node (node x) <i>Sysadmin uses script to notify services of node recovery</i> <i>Sysadmin manually adds node to RM/JS, allocate to job z</i>
10(a)	MPI*	Catch	Restored Node (node x) <i>Return node x to resource pool for job z (MPI looks up new resources)</i>
10(b)	Monitoring System	Catch	Restored Node (node x) <i>Return node x to the set of monitored resources</i>
11	MPI*	Throw	Restored MPI Node (node x: job z) <i>Translates node x to job z (value stored internally)</i>
12(a)	MPI	Catch	Restored MPI Node (node x: job z) <i>Add node x as an unallocated resource</i>
12(b)	Application	Catch	Restored MPI Node (node x: job z) <i>If needed, use MPI_Comm_spawn to create new processes</i>

### B.3 Workflow: Checkpoint/Restart & Process Migration

All of these workflows detail a response to a predicted node failure. So with advance notice of a failure, preventative actions are triggered to mitigate the impact of the failure. Additionally a RM/JS might wish to trigger a checkpoint to provide a coarse-grained, gang scheduling type of functionality.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Initialization &amp; Job Launch</i>		
0	RM/JS	Register	Restored Node (node *)
0	RM/JS	Register	Suspend Job (job z)
0	RM/JS	Register	Resume Job (job z)
0	RM/JS	Register	Resume Job Cmd (job z)
0	RM/JS	Register	Predict Problem Node (node *)
0	RM/JS	Register	Migrate Node (job z: node x,q)
0	RM/JS	Register	Migrate Node Done (job z: node x,q)
0	RM/JS	Register	Restored Node (node *)
0	RM/JS	Register	Restored MPI Node (node *: job z)
0	Autonomic Script	Register	Restored Node (node *)
0	Autonomic Script	Register	Predict Problem Node (node *)
0	MPI	Register	Suspend Job (job z)
0	MPI	Register	Resume Job (job z)
0	MPI	Register	Resume Job Cmd (job z)
0	MPI	Register	Migrate Node (job z: node x,q)
0	MPI	Register	Migrate Node Done (job z: node x,q)
0	MPI	Register	Restored MPI Node (node *: job z)

#### B.3.1 Workflow: Gang Scheduling Support

Gang scheduling support. The RM/JS suspends and resumes entire jobs using a checkpoint/restart technique in cooperation with the MPI implementation.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>RM/JS decides to suspend job z using CPR</i>		
1	RM/JS	Throw	Suspend Job (job z)
	<i>Suspend job z</i>		
2	MPI	Catch	Suspend Job (job z)
	<i>Coordinate a global checkpoint operation. Suspend/Terminate job z</i>		
3	MPI	Throw	Resume Job Cmd (job z)
	<i>Provide RM/JS with the command needed to resume job z</i>		
4	RM/JS	Catch	Resume Job Cmd (job z)
	<i>Store command with information for job z</i>		
	<i>RM/JS decides to resume job z from CPR</i>		
5	RM/JS	Throw	Resume Job (job z)
	<i>Use stored resume information for job z to restart job</i>		
6	MPI	Catch	Resume Job (job z)
	<i>Bring job z back into a running state</i>		

### B.3.2 Workflow: Predicted Failure, Job Suspend

A monitoring system predicts a node failure based on heuristic information gathered from the operating system, network card, and other system resources. The job is suspended and rescheduled for later execution.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>RM/JS decides to suspend job z using CPR</i>		
1	Autonomic Script	Throw	Predict Problem Node (node x)
	<i>Information gathered indicates emanate failure of node x</i>		
2	RM/JS	Catch	Predict Problem Node (node x)
	<i>Suspend scheduling on node x (predicted failure)</i>		
	<i>Translate node x to job z</i>		
3	RM/JS	Throw	Suspend Job (job z)
	<i>Suspend job z</i>		
4	MPI	Catch	Suspend Job (job z)
	<i>Coordinate a global checkpoint operation. Suspend/Terminate job z</i>		
5	MPI	Throw	Resume Job Cmd (job z)
	<i>Provide RM/JS with the command needed to resume job z</i>		
6	RM/JS	Catch	Resume Job Cmd (job z)
	<i>Store command with information for job z</i>		
	<i>Reschedule job z</i>		
	<i>Job z becomes runnable once again</i>		
7	RM/JS	Throw	Resume Job (job z)
	<i>Use stored resume information for job z to restart job</i>		
8	MPI	Catch	Resume Job (job z)
	<i>Bring job z back into a running state</i>		
	<i>Time passes, node x returned to service</i>		
9	Autonomic Script	Throw	Restored Node (node x)
	<i>Information gathered indicates node x is stable again</i>		
10	RM/JS	Catch	Restored Node (node x)
	<i>Return node x to resource pool</i>		

### B.3.3 Workflow: Predicted Failure, Process Migration

A monitoring system predicts a node failure based on heuristic information gathered from the operating system, network card, and other system resources. Affected processes are migrated to alternative resources provided by the RM/JS.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>RM/JS decides to suspend job z using CPR</i>		
1	Autonomic Script	Throw	Predict Problem Node (node x) <i>Information gathered indicates emanate failure of node x</i>
2	RM/JS	Catch	Predict Problem Node (node x) <i>Suspend scheduling on node x (predicted failure)</i> <i>Translate node x to job z</i>
3	RM/JS	Throw	Migrate Node (job z: node x,q) <i>Allocate spare node q to job z</i> <i>Migrate processes from job z on node x to new node q</i>
4	MPI	Catch	Migrate Node (job z: node x,q) <i>Coordinate a global checkpoint operation.</i> <i>Migrate ranks from node x to new node q. Resume application</i>
5	MPI	Throw	Migrate Node Done (job z: node x,q) <i>Tell RM/JS that migration is finished</i>
6	RM/JS	Catch	Migrate Node Done (job z: node x,q) <i>Receive confirmation that node x no longer contains MPI ranks</i>
	<i>Time passes, node x returned to service</i>		
7	Autonomic Script	Throw	Restored Node (node x) <i>Information gathered indicates node x is stable again</i>
8	RM/JS	Catch	Restored Node (node x) <i>Return node x to resource pool for job z</i>
9	RM/JS	Throw	Restored MPI Node (node x: job z) <i>Translates node x to job z</i>
10	MPI	Catch	Restored MPI Node (node x: job z) <i>Add node x as an unallocated resource</i>

### B.4 Workflow: Checkpoint/Restart & Process Migration (No RM/JS)

All of these workflows detail a response to a predicted node failure. So with advance notice of a failure, preventative actions are triggered to mitigate the impact of the failure. *MPI\** denotes *mpirun* playing the role of RM/JS.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Initialization &amp; Job Launch</i>		
0	MPI*	Register	Restored Node (node *)
0	MPI*	Register	Suspend Job (job z)
0	MPI*	Register	Resume Job Cmd (job z)
0	MPI*	Register	Predict Problem Node (node *)
0	MPI*	Register	Migrate Node (job z: node x,q)
0	MPI*	Register	Migrate Node Done (job z: node x,q)
0	MPI*	Register	Restored Node (node *)
0	MPI*	Register	Restored MPI Node (node *: job z)
0	Autonomic Script	Register	Restored Node (node *)
0	Autonomic Script	Register	Predict Problem Node (node *)
0	MPI	Register	Suspend Job (job z)
0	MPI	Register	Resume Job (job z)
0	MPI	Register	Resume Job Cmd (job z)
0	MPI	Register	Migrate Node (job z: node x,q)
0	MPI	Register	Migrate Node Done (job z: node x,q)
0	MPI	Register	Restored MPI Node (node *: job z)

### B.4.1 Workflow: Predicted Failure, Job Suspend

A monitoring system predicts a node failure based on heuristic information gathered from the operating system, network card, and other system resources. The job is checkpointed and terminated. The user can manually resubmit the job at a later time.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>MPI decides to suspend job z using CPR to avoid node failure</i>		
1	Autonomic Script	Throw	Predict Problem Node (node x) <i>Information gathered indicates emanate failure of node x</i>
2	MPI*	Catch	Predict Problem Node (node x) <i>Translate node x to job z (value stored internally)</i>
3	MPI*	Throw	Suspend Job (job z) <i>Suspend job z</i>
4	MPI	Catch	Suspend Job (job z) <i>Coordinate a global checkpoint operation. Suspend/Terminate job z</i>
5	MPI	Throw	Resume Job Cmd (job z) <i>Provide MPI* with the command needed to resume job z</i>
6	MPI*	Catch	Resume Job Cmd (job z) <i>Print command for user to resubmit job z at a later time</i>
	<i>Time passes, node x returned to service</i>		
7	Autonomic Script	Throw	Restored Node (node x) <i>Information gathered indicates node x is stable again</i> <i>Sysadmin manually adds node to RM/JS</i> <i>User can manually resubmit job using resume command provided by MPI*</i>

### B.4.2 Workflow: Predicted Failure, Process Migration

A monitoring system predicts a node failure based on heuristic information gathered from the operating system, network card, and other system resources. Affected processes are migrated to alternative resources from the list of resources known to `mpirun` (this may also oversubscribe nodes depending on policy).

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>MPI decides to suspend job z using CPR to avoid node failure</i>		
1	Autonomic Script	Throw	Predict Problem Node (node x)
	<i>Information gathered indicates emanate failure of node x</i>		
2	MPI*	Catch	Predict Problem Node (node x)
	<i>Translate node x to job z (value stored internally)</i>		
3	MPI*	Throw	Migrate Node (job z: node x,q)
	<i>Determine target node q from known pool of nodes in job z</i>		
	<i>Migrate processes from job z on node x to new node q</i>		
4	MPI	Catch	Migrate Node (job z: node x,q)
	<i>Coordinate a global checkpoint operation.</i>		
	<i>Migrate ranks from node x to new node q. Resume application</i>		
5	MPI	Throw	Migrate Node Done (job z: node x,q)
	<i>Tell MPI* that migration is finished</i>		
6	MPI*	Catch	Migrate Node Done (job z: node x,q)
	<i>Receive confirmation that node x no longer contains MPI ranks</i>		
	<i>Time passes, node x returned to service</i>		
7	Autonomic Script	Throw	Restored Node (node x)
	<i>Information gathered indicates node x is stable again</i>		
	<i>Sysadmin manually adds node to RM/JS, reallocate to job z</i>		
8	MPI*	Catch	Restored Node (node x)
	<i>Return node x to resource pool for job z (MPI looks up new resources)</i>		
9	MPI*	Throw	Restored MPI Node (node x: job z)
	<i>Translates node x to job z (value stored internally)</i>		
10	MPI	Catch	Restored MPI Node (node x: job z)
	<i>Add node x as an unallocated resource</i>		
	<i>May also re-load balance</i>		

## B.5 Workflow: Faulty Interconnect

The following table details the events that each component will want to either throw or catch.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Initialization &amp; Job Launch</i>		
0	RM/JS	Register	Failed Physical Interface (iface *: node *)
0	RM/JS	Register	Failed MPI Physical Interface (iface *: node *: job *)
0	RM/JS	Register	Restored Physical Interface (iface *: node *)
0	RM/JS	Register	Restored MPI Physical Interface (iface *: node *: job *)
0	RM/JS	Register	MPI Message Corruption (node *: job *)
0	IB Fault Monitor	Register	Failed Physical Interface (iface *: node *)
0	IB Fault Monitor	Register	Restored Physical Interface (iface *: node *)
0	IB Fault Monitor	Register	Check Physical Interface (iface *: node *)
0	Autonomic Script	Register	Failed Physical Interface (iface *: node *)
0	Autonomic Script	Register	Restored Physical Interface (iface *: node *)
0	Autonomic Script	Register	Check Physical Interface (iface *: node *)
0	MPI	Register	Failed MPI Physical Interface (iface *: node *: job z)
0	MPI	Register	Restored MPI Physical Interface (iface *: node *: job z)
0	MPI	Register	MPI Message Corruption (node *: job z)

### B.5.1 Workflow: Fail-over to an Alternative Device

A physical network interface fails, MPI fails-over to an alternative device and continues.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
			<i>Interface p fails on node x, job z running on node x</i>
			<i>IB Fault Monitor is first to detect</i>
1	IB Fault Monitor	Throw	Failed Physical Interface (iface p: node x) <i>Interface p on node x has failed</i>
2(a)	RM/JS	Catch	Failed Physical Interface (iface p: node x) <i>Translate node x to job z</i>
2(b)	Autonomic Script	Catch	Failed Physical Interface (iface p: node x) <i>Attempt diagnose and clean up IB routes and switches</i>
3	RM/JS	Throw	Failed MPI Physical Interface (iface p: node x: job z) <i>Notify MPI of failed interface</i>
4	MPI	Catch	Failed MPI Physical Interface (iface p: node x: job z) <i>Mark interface p as down</i> <i>If possible, use an alternative interface</i> <i>If not, suspend communication until interface restored</i>
			<i>Interface p returned to service on node x</i>
5	Autonomic Script	Throw	Restored Physical Interface (iface p: node x) <i>Interface p has been restored to service on node x</i>
6(a)	IB Fault Monitor	Catch	Restored Physical Interface (iface p: node x) <i>Confirm interface is restored</i>
6(b)	RM/JS	Catch	Restored Physical Interface (iface p: node x) <i>Translate node x to job z</i>
7	RM/JS	Throw	Restored MPI Physical Interface (iface p: node x: job z) <i>Notify MPI of restored/new interface p</i>
8	MPI	Catch	Restored MPI Physical Interface (iface p: node x: job z) <i>Add interface p back to the possible interfaces for communication</i>

### B.5.2 Workflow: React to Corrupted or Missing Data

A physical network interface is dropping or corrupting packets. MPI takes corrective action to mask such fails. At some point MPI may decide to remove the interface from service similar to Section B.5.1

	Component	Action	Message
<i>Interface p dropping or corrupting packets on node x</i>			
<i>MPI is first to detect</i>			
1	MPI	Throw	MPI Message Corruption (node x: job z)
<i>MPI detects message corruption</i>			
<i>Continue masking corruption while interfaces are inspected</i>			
2	RM/JS	Catch	MPI Message Corruption (node x: job z)
<i>Translate node x to iface p-q</i>			
3	RM/JS	Throw	Check Interface (iface p-q: node x)
<i>Ask script to check interfaces for suspected failure</i>			
4(a)	Autonomic Script	Catch	Check Interface (iface p-q: node x)
<i>Checks interfaces</i>			
4(b)	IB Fault Monitor	Catch	Check Interface (iface p-q: node x)
<i>Checks interfaces</i>			
5	Autonomic Script	Throw	Failed Physical Interface (iface p: node x)
<i>Notify of confirmed failed interface</i>			
6	RM/JS	Catch	Failed Physical Interface (iface p: node x)
<i>Translate node x to job z</i>			
7	RM/JS	Throw	Failed MPI Physical Interface (iface p: node x: job z)
<i>Notify MPI of failed interface</i>			
8	MPI	Catch	Failed MPI Physical Interface (iface p: node x: job z)
<i>Mark interface p as down</i>			
<i>If possible, use an alternative interface</i>			
<i>If not, suspend communication until interface restored</i>			
<i>Interface p returned to service on node x</i>			
9	Autonomic Script	Throw	Restored Physical Interface (iface p: node x)
<i>Interface p has been restored to service on node x</i>			
10(a)	IB Fault Monitor	Catch	Restored Physical Interface (iface p: node x)
<i>Confirm interface is restored</i>			
10(b)	RM/JS	Catch	Restored Physical Interface (iface p: node x)
<i>Translate node x to job z</i>			
11	RM/JS	Throw	Restored MPI Physical Interface (iface p: node x: job z)
<i>Notify MPI of restored/new interface p</i>			
12	MPI	Catch	Restored MPI Physical Interface (iface p: node x: job z)
<i>Add interface p back to the possible interfaces for communication</i>			

## B.6 Workflow: Faulty Interconnect (No RM/JS)

The following table details the events that each component will want to either throw or catch. *MPI\** denotes `mpirun` playing the role of RM/JS.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Initialization &amp; Job Launch</i>		
0	MPI*	Register	Failed Physical Interface (iface *: node *)
0	MPI*	Register	Failed MPI Physical Interface (iface *: node *: job *)
0	MPI*	Register	Restored Physical Interface (iface *: node *)
0	MPI*	Register	Restored MPI Physical Interface (iface *: node *: job *)
0	MPI*	Register	MPI Message Corruption (node *: job *)
0	IB Fault Monitor	Register	Failed Physical Interface (iface *: node *)
0	IB Fault Monitor	Register	Restored Physical Interface (iface *: node *)
0	IB Fault Monitor	Register	Check Physical Interface (iface *: node *)
0	Autonomic Script	Register	Failed Physical Interface (iface *: node *)
0	Autonomic Script	Register	Restored Physical Interface (iface *: node *)
0	Autonomic Script	Register	Check Physical Interface (iface *: node *)
0	MPI	Register	Failed MPI Physical Interface (iface *: node *: job z)
0	MPI	Register	Restored MPI Physical Interface (iface *: node *: job z)
0	MPI	Register	MPI Message Corruption (node *: job z)

### B.6.1 Workflow: Fail-over to an Alternative Device

A physical network interface fails, MPI fails-over to an alternative device and continues.

	<b>Component</b>	<b>Action</b>	<b>Message</b>
	<i>Interface p fails on node x, job z running on node x</i>		
	<i>IB Fault Monitor is first to detect</i>		
1	IB Fault Monitor	Throw	Failed Physical Interface (iface p: node x) <i>Interface p on node x has failed</i>
2(a)	MPI*	Catch	Failed Physical Interface (iface p: node x) <i>Translate node x to job z (value stored internally)</i>
2(b)	Autonomic Script	Catch	Failed Physical Interface (iface p: node x) <i>Attempt diagnose and clean up IB routes and switches</i>
3	MPI*	Throw	Failed MPI Physical Interface (iface p: node x: job z) <i>Notify MPI of failed interface</i>
4	MPI	Catch	Failed MPI Physical Interface (iface p: node x: job z) <i>Mark interface p as down</i> <i>If possible, use an alternative interface</i> <i>If not, suspend communication until interface restored</i>
	<i>Interface p returned to service on node x</i>		
5	Autonomic Script	Throw	Restored Physical Interface (iface p: node x) <i>Interface p has been restored to service on node x</i>
6(a)	IB Fault Monitor	Catch	Restored Physical Interface (iface p: node x) <i>Confirm interface is restored</i>
6(b)	MPI*	Catch	Restored Physical Interface (iface p: node x) <i>Translate node x to job z</i>
7	MPI*	Throw	Restored MPI Physical Interface (iface p: node x: job z) <i>Notify MPI of restored/new interface p</i>
8	MPI	Catch	Restored MPI Physical Interface (iface p: node x: job z) <i>Add interface p back to the possible interfaces for communication</i>

### B.6.2 Workflow: React to Corrupted or Missing Data

A physical network interface is dropping or corrupting packets. MPI takes corrective action to mask such fails. At some point MPI may decide to remove the interface from service similar to Section B.6.1

	Component	Action	Message
	<i>Interface p dropping or corrupting packets on node x</i>		
	<i>MPI is first to detect</i>		
1	MPI	Throw	MPI Message Corruption (node x: job z)
	<i>MPI detects message corruption</i>		
	<i>Continue masking corruption while interfaces are inspected</i>		
2	MPI*	Catch	MPI Message Corruption (node x: job z)
	<i>Translate node x to iface p-q (values stored internally)</i>		
3	MPI*	Throw	Check Interface (iface p-q: node x)
	<i>Ask script to check interfaces for suspected failure</i>		
4(a)	Autonomic Script	Catch	Check Interface (iface p-q: node x)
	<i>Checks interfaces</i>		
4(b)	IB Fault Monitor	Catch	Check Interface (iface p-q: node x)
	<i>Checks interfaces</i>		
5	Autonomic Script	Throw	Failed Physical Interface (iface p: node x)
	<i>Notify of confirmed failed interface</i>		
6	MPI*	Catch	Failed Physical Interface (iface p: node x)
	<i>Translate node x to job z (values stored internally)</i>		
7	MPI*	Throw	Failed MPI Physical Interface (iface p: node x: job z)
	<i>Notify MPI of failed interface</i>		
8	MPI	Catch	Failed MPI Physical Interface (iface p: node x: job z)
	<i>Mark interface p as down</i>		
	<i>If possible, use an alternative interface</i>		
	<i>If not, suspend communication until interface restored</i>		
	<i>Interface p returned to service on node x</i>		
9	Autonomic Script	Throw	Restored Physical Interface (iface p: node x)
	<i>Interface p has been restored to service on node x</i>		
10(a)	IB Fault Monitor	Catch	Restored Physical Interface (iface p: node x)
	<i>Confirm interface is restored</i>		
10(b)	MPI*	Catch	Restored Physical Interface (iface p: node x)
	<i>Translate node x to job z (values stored internally)</i>		
11	MPI*	Throw	Restored MPI Physical Interface (iface p: node x: job z)
	<i>Notify MPI of restored/new interface p</i>		
12	MPI	Catch	Restored MPI Physical Interface (iface p: node x: job z)
	<i>Add interface p back to the possible interfaces for communication</i>		

## C MPICH2 FTB Events

This page describes the FTB events thrown and caught by MPICH2 in version 1.2.1

### Event Space

MPICH2 throws events in the following event space:

Region name:	<b>ftb</b>
Component category:	<b>mpi</b>
Component name:	<b>mpich2</b>

### Events Thrown

#### RESOURCES

Some resource has been exhausted. The payload describes the type of resource e.g.:

Type	Description
mem	memory allocation
request	request object
communicator	communicator object
datatype	datatype object
context_id	context id for communicator
sock	no socket buffers available

Severity: **error**

#### COMMUNICATION

A communication operation failed. The payload contains the address of the process with which the operation failed, or "unknown".

Severity: **error**

#### UNREACHABLE

An attempt to establish a connection with a process failed. The payload contains the address of the process to which the connection attempt failed. or "unknown". Severity: **error**

#### ABORT

The process aborted. This is due to the application calling MPI\_Abort() or an internal error which caused the application to abort. Severity: **fatal**

#### OTHER

Unspecified error. Severity: **error**

### Events Caught

MPICH2 does not catch any events at this time.

## D MVAPICH2 FTB Events

In this document we present the list of FTB events used by MVAPICH2 to proactive migrate MPI processes when a node is failing.

### CR\_FTB\_NODE\_FAIL

This event is published by the Migration-Trigger component and consumed by *mpirun\_rsh* (the job launcher). It contains the name of the failing node.

**Severity:** INFO

**Payload:** node:<hostname>

The payload includes the hostname which is expected to fail.

### CR\_FTB\_USER\_TRIGGER

This event is published when a user requests a migration (for example in a maintenance operation). Published by the Migration-Trigger component and consumed by the job launcher. It contains the name of the source node of migration.

**Severity:**INFO

**Payload:** node:<hostname>

The payload includes the hostname which is expected to become unavailable.

### CR\_FTB\_MIGRATE

This event is published by *mpirun\_rsh* and consumed by the *mpispawns* and CR (Checkpoint Restart) module. It contains the name of the source and target nodes of the migration.

**Severity:** INFO

**Payload:** node:<src\_hostname, dest\_hostname>

The payload contains the hostname of the source and target nodes of the migration.

### CR\_FTB\_CKPT\_DONE

This event is published by the MPI processes that were able to checkpoint and indicates that the checkpoint is completed successfully. It is consumed by *mpirun\_rsh*.

**Severity:** INFO

### CR\_FTB\_CKPT\_FAIL

This event is published by the MPI processes that failed to take the checkpoint and indicates that the checkpoint is failed. It is consumed by *mpirun\_rsh*.

**Severity:** ERROR

### CR\_FTB\_MIGRATE\_IMG\_C

This event is published by the *mpispawn* on the migration source and indicates that the migration of the checkpoint image is complete. It is consumed by *mpirun\_rsh*.

**Severity:** INFO

**Payload:** node:<src\_hostname, tgt\_hostname>

The payload includes the source and target nodes of the migration.

### **CR\_FTБ\_MIGRATE\_RESTART**

This event is published by *mpirun\_rsh* and consumed by *mpispawns*. It indicates that the migrated processes can be restarted.

**Severity:** INFO

**Payload:** node:<tgt\_hostname>, rank:<mpiranks>

The payload includes the target node of the migration and the list of processes ranks migrated.

### **CR\_FTБ\_RSRT\_DONE**

This event is published by MPI processes and consumed by *mpirun\_rsh*. It indicates that the restart completed successfully.

**Severity:** INFO

### **CR\_FTБ\_RSRT\_FAIL**

This event is published by MPI processes and consumed by *mpirun\_rsh*. It indicates that the restart failed.

**Severity:** ERROR

### **CR\_FTБ\_CKPT\_FINALIZE**

This event indicates that the CR module has been shutdown. Published by all MPI processes doing MPI\_Finalize().

**Severity:** INFO

## E MVAPICH FTB Workflows

FTB-IB is a FTB component that uses the FTB infrastructure to notify other FTB enabled components about failures in the Infiniband Network. Since FTB-IB is a low-level component, it is not really dependent on other components and so does not have to subscribe to events. However, the following two workflow show where FTB-IB could be used.

### E.1 Workflow: Process migration due to network failure

1. An MPI Job is launched on a given set of nodes.
2. Depending on the design of the MPI Library, it could use one or more Ports / HCAs. Assume for the sake of this example that there is only one active port per node.
3. The MPI library subscribes to the FTB\_IB\_ADAPTER\_INFO and FTB\_IB\_PORT\_INFO events. As the name suggests, these FTB Events indicate the status of the InfiniBand Adapters / Ports.
4. Assume that the port that was used by the MPI library goes down. The MPI Library will see a specific failure, maybe `ibv_poll_cq()` failing with `IBV_WC_RETRY_EXC_ERR`. However, this information is not sufficient to determine if the failure was due to a port failure on the sender's node or on the receiver's node.
5. The FTB\_IB\_PORT\_INFO event thrown by FTB-IB would indicate to the MPI library (either on the sender or the receiver as the case may be) that a port went down. Armed with this information, the MPI Library can then trigger a process migration from the faulty node to a spare node.

### E.2 Workflow: Port Failover

1. IB Adapters are usually equipped with 2 ports for "High-Availability". A well designed IB network would use both ports of each adapter to ensure that from a given node, all other nodes are reachable through either of the two ports and survive one or more link failures. The same concept could be extended by using multiple IB Adapters per node.
2. In the event of a port failure, the MPI Library can fail-over to using alternate ports to maintain connectivity.
3. IU Workflow - Interconnect Failure, Section B.6 talks about this in greater detail.