Proposal for MPI 3.0 Error Handling

Limitations of current error handling specification

The MPI 2 spec includes mechanisms error handling and notification. While these mechanisms provide an important capability, especially for informing applications of MPI usage errors, these mechanisms have a number of limitations from the point of view of application fault tolerance.

- The specification makes no demands on MPI to survive failures or provide a way for MPI to operate in a degraded operation mode as a result of an failure. Although MPI implementers are encouraged to "circumscribe the impact of an error, so that normal processing can continue after an error handler was invoked", nothing more is specified in the standard. In particular, the defined MPI error classes are used only to clarify to the user the source of the error and do not describe the MPI functionality that is not available as a result of the error.
- All errors must be associated with some specific MPI call. As such, it is difficult for MPI to notify users of the failure of a given function that happen after the function has already returned. One example of this is `MPI_Send`, which may return after MPI has locally buffered the message. If there is a subsequent failure in sending the message it is not clear how the application would be notified of this event.
- There is no description of when error notification will happen relative to the occurrence of the error. In particular, the specification does not state whether an error that would cause MPI functions to return an error code under the `MPI_ERRORS_RETURN` error handler would cause a user-defined error handler to be called during the same MPI function or at some earlier or later point in time.
- Although MPI makes it possible for libraries to define their own error classes and invoke application error handlers, it is not possible for the application to define new error notification patterns either within or across processes. This means that it is not possible for one application process to ask to be informed of errors on other processes or for the application to be informed of specific classes of errors.

Proposed API

I. Error Occurrence

a. Execution Model

All errors will be defined in terms of an MPI execution model, which is the overall set of entities defined by the MPI specification (processes, communicators, files, one-sided windows, etc.) and their interactions. For each entity type we define a number of possible actions that are available to each entity, along with the degree to which the entity is capable of taking the possible action. For example, a given communication channel transfers data and does so at either full performance, degraded performance or fails to do so. The error handling API is focused on communicating to the application how the capabilities of different entities change over time, both improving and degrading.

The current preliminary model defines the following entities and actions:

- Processes

    Computation: full-performance, degraded-performance, failed

- Uni-directional point to point communication channels

    Data delivery: full-performance, degraded-performance, failed

- Communications (represents point-to-point messages as well as collectives and other data exchanges)

    Data Delivery: successful, partially corrupted (success on one subset of the processes involved in the communication, failure on another), failed

- MPI_Comms

    Data delivery: full-performance, degraded-performance, failed

- MPI_Windows

    Data delivery: full-performance, degraded-performance, failed

    Data storage: operational, failed

- MPI_Files

    Reading: full-performance, degraded-performance, failed

    Writing: full-performance, degraded-performance, failed

    Appending: full-performance, degraded-performance, failed

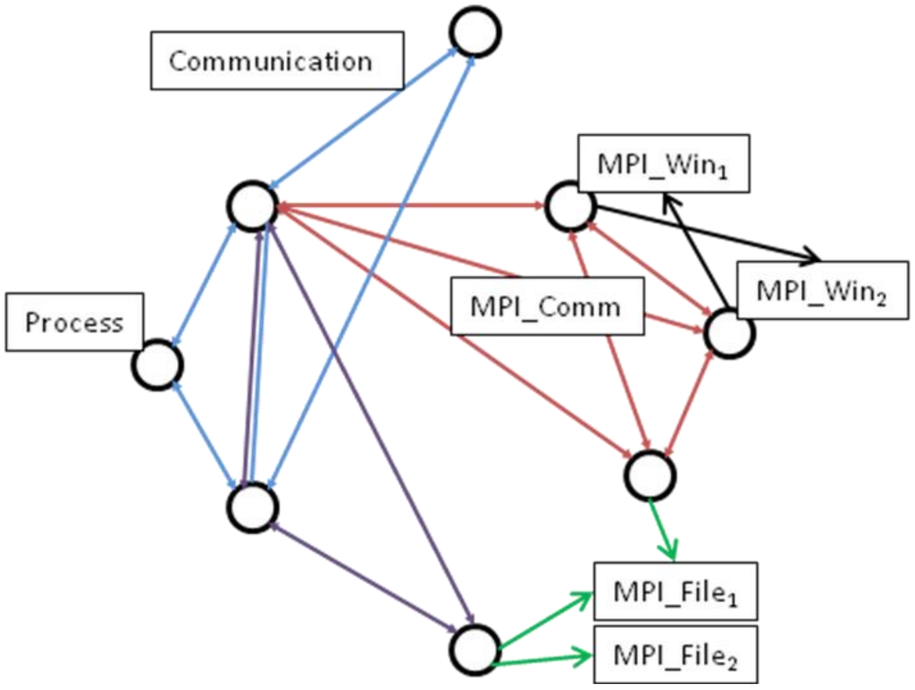
    File creation: full-performance, degraded-performance, failed

    File deletion: full-performance, degraded-performance, failed

    …

Additional entities that may be defined in the future include `MPI_Datatype`, `MPI_Op`, `MPI_Group`, `MPI_Topology`, `MPI_Request`, and `MPI_Status`.

The following figure summarizes the MPI execution model. `Processes` are connected to each other and to `MPI_File` objects via `channels` and may host `MPI_Win` objects. `Processes` may exchange information via `communications`, which include `MPI_Send/MPI_Recv` pairs, `MPI_Sendrecvs`, collective operations, file operations and one-sided operations. Sets of `processes` may be grouped into MPI_Comm objects, which also include any channels between the processes in the MPI_Comm object.



Changes to the execution model are described via events. Each event identifies the MPI entity and the capability that has been affected, the old status of the capability and the new status of the capability. For example, if a node fails, this may be described by the MPI implementation as

- the computation capability of the processes executing on the node transitioning from full-performance to failed status,
- the data delivery capability of the communication channels that involve these processes transitioning from full-performance to failed status, and
- the delivery capability of all in-flight messages to/from the failed processes transitioning from successful to failed status.

It is not defined how MPI implementations map low-level events to changes in the MPI execution model. However, if a given capability is reported as being in failed status, the application's use of this capability should result in an error. On the other hand, if a given capability is reported as non-failed, it is acceptable for subsequent use of this capability to result in an error, as long as this error is also associated with an error event that reports that the capability shifts to a degraded status.

Processes, channels and MPI_Files are the only first-class entities. Communications correspond to individual MPI data exchanges, including MPI_Send/MPI_Recv pairs, MPI_Sendrecvs, collective operations, file operations or one-sided operations. Each communication is associated with a set of processes and channels and with a single MPI_Comm. File or one-sided communications are also associated with their respective `MPI_File` and `MPI_Window` objects. Each `processes` and `channel` is associated with every the `MPI_Comm` objects that contain it.

b. Notification ranks

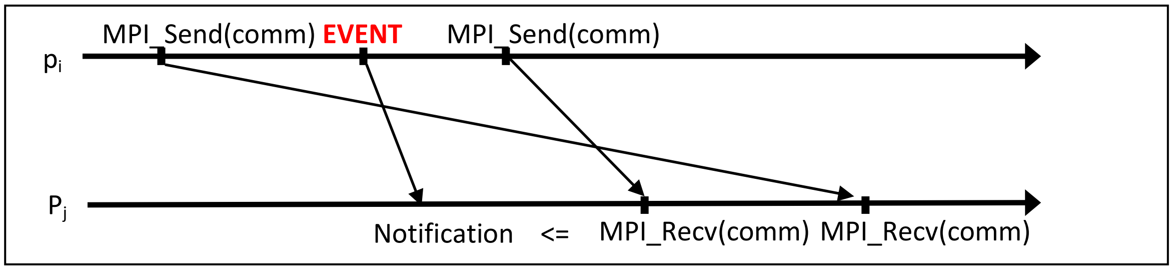
Each event that describes a change in the execution model must be communicated by MPI to one or more application ranks. For each event the standard will define the set of ranks, called the "notification set" that are by default subscribed to this event. It will be possible to change these default subscriptions using an explicit subscription/unsubscription API. For example, the notification set of a process failure event should be the set of ranks that communicate with the process. The notification set of a communication failure event (i.e. corruption of a single point-to-point message) is the set of processes participating in the communication. In the case of collective communication we will need to define whether processes that neither send nor receive the corrupted data should be informed of the corruption event.

c. Notification Sites

Each event will be associated with specific MPI_Comm, MPI_Window or MPI_File object. If a process executes an MPI operation on a given object that has undelivered events associated with the object, those events are delivered during that call. Delivery may be performed via any available API, including return codes, callbacks or explicit event queue retrieval operations (described in Section II). If two MPI function calls operate on a given MPI object with pending events we will need to decide whether the events are delivered to every such call or just one such call.

d. Notification Ordering

Events that are associated with a given MPI object must be delivered in FIFO order relative to any communication that uses this object. For example, consider the example in the following figure where there is a communication error event that is associated with communicator comm. Suppose that process p_j is subscribed to this event (either by default or via explicit subscription). It is required that process p_j is informed of the event no later than the next causal chain that connects the event to process p_j. In this



example, the event notification must happen at or before the first MPI_Recv call that uses comm.

II. Error Notification API

a. Notification subscription

The error notification API is based on the publish-subscribe model, where processes must subscribe to errors on various MPI entities to notified of these errors. The following methods are available for this purpose:

```
MPI_Event_subscribe(MPI_entity entity, int rank, MPI_Comm rankComm,
     MPI_Event_capability capability_type, MPI_Event_change
     capability_change_type, MPI_Event_context context,
     MPI_Event_Subscription* subscription)
```

```
MPI_Event_subscribe(MPI_entity entity,
     MPI_Event_capability capability_type, MPI_Event_change
     capability_change_type, MPI_Event_context context,
     MPI_Event_Subscription* subscription)
```

Subscribes the calling process to events that modify the given capability of the given MPI entity in the given fashion. For objects that may reside on another process, the process' rank must be provided (rank is relative to the given communicator). Any of these arguments may be specified using a wildcard to indicate that the application wishes to subscribe to events that have any value for the given argument. The call returns an opaque object in the `subscription` argument that describes the set of events being subscribed to. It also takes a `context` argument, which is described in section II.b.ii. Associated with the `context` will be an event handler callback function, also described in II.b.ii.

```
int MPI_Event_unsubscribe(MPI_Event_Subscription subscription)
```

The inverse of `MPI_Event_subscribe`.
We need to decide what happens when multiple MPI_Subscribe calls cover the same event and then the application unsubscribes from one of these subscriptions. Does the process remain registered to the events in the intersection of the two original subscriptions?

```
int MPI_Subscribe_free(MPI_Event_Subscription subscription)
```

Frees the event descriptor associated with this handle. This does not unsubscribe the process from the events described by this handle but rather frees the space taken up by the handle itself. It may not be possible to unsubscribe the process from the events that match the given subscription. (will need to work out the details of this)

b. Event delivery

i. Return Codes

The "return codes API" will be used by default when the application doesn't do any specific event subscription. By default for an event E, associated with MPI object O and let P be E's notification set. If process $p_l \in P$ operates on O, this process will return an error code if E represents a transition in some capability of some MPI entity from non-failed to failed state. Other events are not delivered.

ii. Callbacks and contexts

If the application explicitly subscribes to receive events it must provide a callback function for each event pattern it wants to listen for. Given the event E described above, if p_i calls a function that operates on O and the application has subscribed itself to E, MPI may call (subject to the ordering requirements of section I.d) the provided callback. The callback may then return the same or different MPI_Event object to identify the state of the application after the callback has performed its corrective measures. If the event corresponds to an error (e.g. some entity transitions from non-failed to failed state), this causes function call to return an error. Otherwise, the function call returns MPI_SUCCESS.

When the application registers multiple event handler callbacks that match the same event, all of them will be executed by MPI and each of them will be able to return an event. If any of these events correspond to errors, p_i's function call will return an error. Otherwise, it will return MPI_SUCCESS.

Because the above behavior makes it difficult to manage event handling in libraries, this specification uses a concept of event contexts, ensuring that all event subscription and handling is performed within some event context. The intention is to allow each code module (library, thread, etc.) to create its own event context object and do all of its event subscriptions using this object (only one callback allowed per context). Furthermore, given a pair of context objects, the application will be able to establish a partial order between the contexts. Intuitively, if contexA < contextB, the event handler in contextA will be called before the event handler in contextB and contextB's event handler will be given the event returned by contextA's handler, rather than the original event. If given context is ordered after multiple other contexts, all of the events returned by these contexts will be provided as arguments (the argument to the callback is a reference to an array of contexts and the size of the array). p_i's function call returns an error if some event handler that belongs to a context that is followed by no other context returns an error event. Otherwise, it returns MPI_SUCCESS.

Looking from a high level, event contexts allow application modules to establish hierarchies of event notification, allowing lower-level modules to process and respond to events before higher-level modules. Furthermore, since applications will be allowed to define their own events, lower-level modules will be able to define their own execution model and use context chaining to raise the level of abstraction that their users must deal with. It may be useful to enable applications to apply this generic event notification functionality to non-MPI function calls.

```
MPI_Context_create(MPI_Context* context,
     MPI_Event (callback*)(MPI_Event* evt, int num_events))
```

Creates a new context with the given event handler callback.

```
MPI_Context_free(MPI_Context context)
```

Deallocates the given context object and unsubscribes any subscriptions associated with this context.

```
MPI_Context_order(MPI_Context front, MPI_Context back)
```

Given two contexts, orders the `front` context such that its event handler is called before `back’s` event handler.

c. Event management

```
int MPI_Create_event(MPI_Entity entity,
     MPI_Event_capability capability_type,
     MPI_Event_change capability_change_type, MPI_Event* evt)
```

Creates an event from the given MPI entity that describes the given change to the given capability of the entity.

```
int MPI_Event_free(MPI_Event evt)
```

Deallocates the given event.