D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

January 24, 2022
This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 16

Process Fault Tolerance

16.1 Introduction

In distributed systems with numerous or complex components, a serious risk is that a component fault manifests as a process failure that disrupts the normal execution of a long running application. A process failure is a common outcome for many hardware, network, or software faults that cause a process to crash; it can be more formally defined as a fail-stop failure: the affected MPI process unexpectedly and permanently stops communicating. This chapter introduces MPI features that support the development of applications, libraries, and programming languages that can tolerate MPI process failures. The primary goal is to specify error classes and interfaces that permit users to continue simple MPI communication (e.g., some point-to-point patterns) after failures have impacted the execution and rebuild MPI objects (communicators, files, etc.) as needed to restore the full capability of MPI to carry out application elaborate communication operations (like collective communications), or dynamic process operations (allowing for spawning replacement processes). This specification does not include mechanisms to restore the application data lost due to process failures. The literature is rich with diverse fault tolerance techniques that the users may employ at their discretion, including checkpoint-restart, algorithmic dataset recovery, and continuation ignoring failed MPI processes. All these fault tolerance approaches benefit from, and often require, the definitions and interfaces specified in this chapter in order to resume communicating after a failure.

The expected behavior of MPI in the case of an MPI process failure is defined by the following statements: any MPI operation that involves a failed process must not block indefinitely but either succeed or raise an MPI error (see Section 16.2); an MPI operation that does not involve a failed process will complete normally, unless interrupted by the user through provided functionality. By default, errors indicate only the local impact of the failure on an operation, and make no guarantee that other processes have also been notified of the same failure; asynchronous failure propagation is not guaranteed or required, and users must exercise caution when determining the set of processes where a failure has been detected and raised an error. If an application needs global knowledge of failures, it can use the interfaces defined in Section 16.3 to explicitly propagate the notification of locally detected failures, or set communicators in specific modes that enforce such propagation.

Some usage patterns on reliable machines do not require fault tolerance. An MPI implementation that does not tolerate process failures must never raise a *fault tolerance error* (as listed in Section 16.4). Applications using the interfaces defined in this chapter must be portable across MPI implementations (including those which do not provide fault tolerance, but in this case the interfaces may exhibit undefined behavior after a process failure at any MPI process.) Fault tolerant applications may determine if the implementation supports fault tolerance by querying the predefined attribute MPI_FT on MPI_COMM_WORLD (see 9.1.2.)

> *Advice to users.* The MPI standard does not specify transparent process recovery upon MPI process failure. In particular, restoring the lost dataset, spawning spare processes or taking other recovery actions are the responsibility of the user.
>
> Many of the operations and semantics described in this chapter are applicable only when the MPI application has replaced the default error handler MPI_ERRORS_ARE_FATAL on the communicators and windows it uses. (*End of advice to users.*)

16.2 Failure Notification

When an operation raises a fault tolerance error it may not satisfy its specification (like any other error, see 9.4). Note that the reminder of this chapter defines operations that maintain full specification semantic after raising a fault tolerance error; such exceptions will be explicitely stated. A list of fault tolerance errors is provided in Section 16.4.

Nonblocking operations do not raise fault tolerance errors during creation or initiation. The corresponding completion call raises a fault tolerance error when appropriate.

An operation involving a failed MPI process must always complete in a finite amount of time (possibly by raising one of the process failure error classes listed in Section 16.4).

An MPI process is considered involved in a communication (for the purpose of this chapter) if its failure may prevent the successful update of user-visible state (e.g., output buffers, synchronizations, etc.) by its operations. More formally, an MPI process is involved in a communication if any of the following is true:

- The process is in the group over which the operation is collective.
- The process is a destination or a specified or matched source in a point-to-point communication.
- The operation is an MPI_ANY_SOURCE receive operation and the process belongs to the source group.
- The process is a specified target in a remote memory operation.

By default, process failure errors are raised only during communication operations in which a failed process is involved, but the range of processes whose failure may cause operations to raise an error is user controllable.

16.2.1 Error Reporting Range

Users can control the range of processes whose failure cause MPI operations to raise errors on their communicators by setting the following values to the info key "mpi_error_range" on their communicators:

"operation" If an operation on the communicator does not involve a failed MPI process (such as a point-to-point message between two non-failed MPI processes), it must not raise a fault tolerance error. This is the default if the info key is not set.

"group" The failure of any MPI process in the group of the communicator cause the communication context to become revoked. This causes communication operations to raise a fault tolerance error of class MPI_ERR_REVOKED, even in operations that involve only non-failed processes.

"global" The failure of any MPI process in the MPI universe cause communication context to become revoked. This causes communication operations to raise a fault tolerance error of class MPI_ERR_REVOKED, even in operations that involve only non-failed processes.

Advice to implementors. As long as an implementation can complete operations, it may choose to delay raising an error. Another valid implementation might choose to raise an error as quickly as possible. (*End of advice to implementors.*)

16.2.2 Fault Tolerance Errors in Point-to-Point Communication

An MPI implementation raises errors of the following classes in order to notify users that a point-to-point communication operation could not complete successfully because of the failure of at least one involved MPI process:

- MPI_ERR_PROC_FAILED_PENDING indicates, for a nonblocking communication, that the communication is a receive operation from MPI_ANY_SOURCE and no send operation has matched, yet a potential sending MPI process has failed. Neither the operation nor the request identifying the operation is completed.
- In all other cases, the operation raises an error of class MPI_ERR_PROC_FAILED to indicate that the failure prevents the operation from following its failure-free specification. If there is a request identifying a point-to-point communication, it is completed. Communication involving the failed MPI process, initiated on this communicator after the error raised, must also raise an error of class MPI_ERR_PROC_FAILED.

16.2.3 Fault Tolerance Errors in Collective Communication

When a collective operation cannot be completed because of the failure of an involved MPI process, the collective operation raises an error of class MPI_ERR_PROC_FAILED.

Advice to users.

Depending on how the collective operation is implemented and when an MPI process failure occurs, some participating MPI processes may raise an error while other MPI processes return successfully from the same collective operation. For example, in MPI_BCAST, the root process may succeed before a failed MPI process disrupts the operation, resulting in some other processes raising an error.

(*End of advice to users.*)

Advice to users.

Note that communicator creation functions (e.g., MPI_COMM_DUP or MPI_COMM_SPLIT) are collective operations. As such, if a failure happened during the call, an error might be raised at some MPI processes while others succeed and obtain a new communicator handle. Although it is valid to communicate between MPI processes that succeeded in creating the new communicator handle, the user is responsible for ensuring a consistent view of the communicator creation, if needed. A conservative solution is to check the global outcome of the communicator creation function with MPI_COMM_AGREE (defined in Section 16.3.1), as illustrated in Example 16.1. (*End of advice to users.*)

After an MPI process failure, MPI_COMM_FREE (as with all other collective operations) may not complete successfully at all processes. For any MPI process that receives the return code MPI_SUCCESS, the behavior is defined in Section 7.4.3. If an MPI process raises a process failure error (classes MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the communicator handle comm is set to MPI_COMM_NULL; however, the implementation makes no guarantee about the success or failure of the MPI_COMM_FREE operation, locally or remotely.

Advice to users. Users are encouraged to call MPI_COMM_FREE on communicators they do not wish to use anymore, even when they contain failed MPI processes. Although the operation may raise a fault tolerance error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. (*End of advice to users.*)

Error Uniformity

As noted above, by default, collective communication do not enforce uniformity in error raising accross processes. Despite the performance advantages that non-uniformity offer, a common usage pattern in applications is to transform non-uniform error raising into a uniform behavior accross all processes of the group (as illustrated in Example 16.1).

Users can set collective operations on a communicator to enforce uniform error raising by setting the following values in the info key "mpi_error_uniform" on the communicator:

"local" Process fault tolerance errors are raised to indicate that an MPI process failure prevents from garanteeing the specified behavior at the local process for the collective communication operation (e.g., the output buffer contains invalid data). Other processes may have locally satisfied their specification (e.g., the output buffer is valid at that process) and may have returned MPI_SUCCESS.

"coll" Process fault tolerance errors are raised to indicate that an MPI process failure prevents from garanteeing the specified behavior at any process for the collective communication operation. Non-synchronizing collective communication become synchronizing.

"create" Process fault tolerance errors are raised to indicate that an MPI process failure prevents from garanteeing the specified behavior at any process for the MPI communication context creation/destruction collective operation (e.g., MPI_COMM_DUP could not create a new communicator at any process in the group of the communicator). Non-synchronizing context creation/destruction collective operations become synchronizing. Collective communication that do not create or free a communication context are not impacted.

16.2.4 Dynamic Process Management

Rationale. As with communicator creation functions, if a failure happens during a dynamic process management operation, an error might be raised at some MPI processes while others succeed and obtain a new valid communicator. For most communicator creation functions, users can validate the success of the operation by communicating on a pre-existing communicator spanning over the same group of processes (in the worst case, from MPI_COMM_WORLD). This is however not always possible for dynamic process management operations, since these operations can create a new intercommunicator between previously disconnected MPI processes. The following additional failure case semantics allow for users to validate, on the created intercommunicator itself, the success of the dynamic process management operation. (*End of rationale.*)

If the MPI implementation raises a fault tolerance error at the root process in MPI_COMM_ACCEPT or MPI_COMM_CONNECT, the corresponding operation must also raise a fault tolerance error at its root process.

Advice to users. The root process of an operation can succeed when a fault tolerance error is raised at some other non-root process. (*End of advice to users.*)

When using the intercommunicator returned from MPI_COMM_SPAWN, MPI_COMM_SPAWN_MULTIPLE, or MPI_COMM_GET_PARENT, a communication for which the root process of the spawn operation is the source or the destination must not deadlock. When the root process raises a fault tolerance error from a spawn operation, no MPI processes are spawned.

Advice to implementors. An implementation is allowed to abort a spawned MPI process during MPI_INIT when it cannot setup an intercommunicator with the root process of the spawn operation because of a process failure.

An implementation may report it spawned all the requested MPI processes even when a process created from MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE failed, and instead delay raising a fault tolerance error to a later communication involving this process. (*End of advice to implementors.*)

Advice to users. To determine how many new MPI processes have effectively been spawned, the normal semantics for hard and soft spawn applies: if the requested number of processes is unavailable for a hard spawn, an error of class MPI_ERR_SPAWN is raised (possibly leaving MPI in an undefined state), and an appropriate error code is set in the array_of_errcodes parameter. Note however that an implementation may report that it has spawned the requested number of MPI processes even when some MPI processes have failed before exiting MPI_INIT. This condition can be detected by communicating over the created intercommunicator with these processes.(*End of advice to users.*)

Advice to implementors. MPI_COMM_JOIN does not require any supplementary semantics. When the remote MPI process on the fd socket has failed, the operation succeeds and sets intercomm to MPI_COMM_NULL. (*End of advice to implementors.*)

After an MPI process failure, MPI_COMM_DISCONNECT (as with all other collective operations) may not complete successfully at all MPI processes. For any process that receives the return code MPI_SUCCESS, the behavior is defined in 11.10.4. If an MPI process raises a fault tolerance error (classes MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the communicator handle comm is set to MPI_COMM_NULL; however, the implementation makes no guarantee about the success or failure of the MPI_COMM_DISCONNECT operation, locally or remotely.

> *Advice to users.* Users are encouraged to call MPI_COMM_DISCONNECT on communicators they do not wish to use anymore, even when they contain failed MPI processes. Although the operation may raise a fault tolerance error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. (*End of advice to users.*)

16.2.5 One-Sided Communication

When an operation on a window raises a fault tolerance error, the state of all data held in memory exposed by that window becomes undefined at all MPI processes for which a one-sided communication operation could have modified local data in that window (a target in a remote write, or accumulate operation, or an origin in a remote read operation), and the operation completion has not been semantically guaranteed (as an example by a successful synchronization between the origin and the target, after the origin had issued an MPI_WIN_FLUSH).

> *Advice to users.* Assessing if a particular portion of the exposed memory remains correct is the responsibility of the user. Note that in passive target mode, when an error is raised at the origin, the target memory data may become undefined before a synchronization raises an error at the target.
>
> The exposed memory data becomes undefined for all uses, not only the window in which the error was raised. Any overlapping windows or uses involving shared memory also read undefined data (even if they do not involve MPI calls). (*End of advice to users.*)

> *Advice to implementors.* A high quality implementation should limit the scope of the exposed memory that becomes undefined (for example, only the memory addresses and ranges that have been targeted by a remote write, or accumulate, or have been an origin in a remote read). In that case, we encourage implementations to document the provided behavior, and to expose the availability of this feature at runtime, as an example by caching an implementation specific attribute on the window. (*End of advice to implementors.*)

Non-synchronizing one-sided communication operations (as an example MPI_GET, MPI_PUT) whose outputs are undefined, due to an MPI process failure, are not required to raise a fault tolerance error. However, if a communication cannot complete correctly due to process failures, the synchronization operation must raise a fault tolerance error at least at the origin.

> *Advice to implementors.* Non-synchronizing operations (MPI_WIN_FLUSH_LOCAL, MPI_WIN_FLUSH_LOCAL_ALL) are not required to raise a fault tolerance error. (*End of advice to implementors.*)

Advice to users. As with collective operations over MPI communicators, active target one-sided synchronization operations may raise a fault tolerance error at some MPI process while the corresponding operation returned MPI_SUCCESS at some other MPI process. (*End of advice to users.*)

Passive target synchronization operations may raise a process failure error when any MPI process in the window has failed (even when the target specified in the argument of the passive target synchronization has not failed).

Rationale. An implementation of passive target synchronization may need to communicate with non-target MPI processes in the window, as an example, a previous owner of an access epoch on the target window. (*End of rationale.*)

After an MPI process failure, MPI_WIN_FREE (as with all other collective operations) may not complete successfully at all MPI processes. For any process that receives the return code MPI_SUCCESS, the behavior is defined in Section 12.2.5. If a process raises a process failure error (classes MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the window handle win is set to MPI_WIN_NULL; however, the implementation makes no guarantee about the success or failure of the MPI_WIN_FREE operation, locally or remotely.

Advice to users. Users are encouraged to call MPI_WIN_FREE on windows they do not wish to use anymore, even when they contain failed MPI processes. Although the operation may raise a fault tolerance error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. Before calling MPI_WIN_FREE, it may be required to call MPI_WIN_REVOKE to close an epoch that couldn't be completed as a consequence of a process failure (see Section 16.3.2). (*End of advice to users.*)

16.2.6 I/O

This section defines the behavior of I/O operations when MPI process failures prevent their successful completion. I/O backend failure error classes and their consequences are defined in Section 14.7.

If am MPI process failure prevents a file operation from completing, an MPI error of class MPI_ERR_PROC_FAILED is raised. Once an MPI implementation has raised an error of class MPI_ERR_PROC_FAILED, the state of the file pointers involved in the operation that raised the error is *undefined*.

Advice to users. Since collective I/O operations may not synchronize with other MPI processes, process failures may not be reported during a collective I/O operation. Users are encouraged to use MPI_COMM_AGREE on a communicator containing the same group as the file handle when they need to deduce the completion status of collective operations on file handles and maintain a consistent view of file pointers. The file pointer can be reset by using MPI_FILE_SEEK with the MPI_SEEK_SET update mode. (*End of advice to users.*)

After an MPI process failure, MPI_FILE_CLOSE (as with all other collective operations) may not complete successfully at all MPI processes. For any MPI process that receives the return code MPI_SUCCESS, the behavior is defined in Section 14.2.2. If an MPI process raises a process failure error (classes MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the file handle fh is set to MPI_FILE_NULL; however, the implementation makes no guarantee about the success or failure of the MPI_FILE_CLOSE operation, locally or remotely.

> *Advice to users.* Users are encouraged to call MPI_FILE_CLOSE on files they do not wish to use anymore, even when they contain failed MPI processes. Although the operation may raise a fault tolerance error and not synchronize properly, this gives a high quality implementation an opportunity to release local resources and memory consumed by the object. (*End of advice to users.*)

16.3 Failure Mitigation Functions

16.3.1 Communicator Functions

Process failure notification is not global in MPI. MPI processes that do not call operations involving a failed MPI process are possibly never notified of its failure (see Section 16.2). If a notification must be propagated, MPI provides a function to revoke a communicator at all members.

MPI_COMM_REVOKE(comm)

IN	comm	communicator (handle)

C binding

```
int MPI_Comm_revoke(MPI_Comm comm)
```

Fortran 2008 binding

```
MPI_Comm_revoke(comm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_REVOKE(COMM, IERROR)
    INTEGER COMM, IERROR
```

This function notifies all MPI processes in the groups (local and remote) associated with the communicator comm that this communicator is revoked. The revocation of a communicator by any MPI process completes non-local MPI operations on comm at all MPI processes by raising an error of class MPI_ERR_REVOKED (with the exception of MPI_COMM_SHRINK, MPI_COMM_AGREE, and MPI_COMM_IAGREE). This function is not collective and therefore does not have a matching call on remote MPI processes. All non-failed MPI processes belonging to comm will be notified of the revocation despite failures.

A communicator is revoked at a given MPI process either when MPI_COMM_REVOKE is locally called on it, or when any MPI operation on comm raises an error of class MPI_ERR_REVOKED at that process. Once a communicator has been revoked at an MPI process, all subsequent non-local operations on that communicator (with the same exceptions as above), are considered local and must complete by raising an error of class MPI_ERR_REVOKED at that process.

MPI_COMM_IS_REVOKED(comm, flag)

IN	comm	communicator (handle)
OUT	flag	true if the communicator is revoked (logical)

C binding

```
int MPI_Comm_is_revoked(MPI_Comm comm, int *flag)
```

Fortran 2008 binding

```
MPI_Comm_is_revoked(comm, flag, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_IS_REVOKED(COMM, FLAG, IERROR)
    INTEGER COMM, IERROR
    LOGICAL FLAG
```

Returns flag = true if the communicator associated with the handle comm is revoked at the calling process. It returns flag = false otherwise. The operation is local.

> *Advice to users.* In a multithreaded application, a thread calling MPI_COMM_IS_REVOKED may return flag = true before the operation that raises the first exception of class MPI_ERR_REVOKED has completed in a concurrent thread. (*End of advice to users.*)

MPI_COMM_SHRINK(comm, newcomm)

IN	comm	communicator (handle)
OUT	newcomm	communicator (handle)

C binding

```
int MPI_Comm_shrink(MPI_Comm comm, MPI_Comm *newcomm)
```

Fortran 2008 binding

```
MPI_Comm_shrink(comm, newcomm, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT) :: newcomm
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_SHRINK(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

This collective operation creates a new intra- or intercommunicator newcomm from the intra- or intercommunicator comm, respectively, by excluding the group of failed MPI processes as agreed upon during the operation. The groups of newcomm must include every MPI process that returns from MPI_COMM_SHRINK, and it must exclude every MPI process whose failure caused an operation on comm to raise an MPI error of class MPI_ERR_PROC_FAILED or MPI_ERR_PROC_FAILED_PENDING at a member of the groups of newcomm, before that member initiated MPI_COMM_SHRINK. This call is semantically equivalent to an MPI_COMM_SPLIT operation that would succeed despite failures, where members of the groups of newcomm participate with the same color and a key equal to their rank in comm.

This function never raises an error of class MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED. The defined semantics of MPI_COMM_SHRINK are maintained when comm is revoked, or when the group of comm contains failed MPI processes.

Advice to users. MPI_COMM_SHRINK is a collective operation, even when comm is revoked.

The group of newcomm may still contain failed MPI processes, whose failure will be detected in subsequent MPI operations. (*End of advice to users.*)

MPI_COMM_ISHRINK(comm, newcomm, request)

IN	comm	communicator (handle)
OUT	newcomm	communicator (handle)
OUT	request	communication request (handle)

C binding

```
int MPI_Comm_ishrink(MPI_Comm comm, MPI_Comm *newcomm,
             MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Comm_ishrink(comm, newcomm, request, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Comm), INTENT(OUT), ASYNCHRONOUS :: newcomm
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_ISHRINK(COMM, NEWCOMM, REQUEST, IERROR)
    INTEGER COMM, NEWCOMM, REQUEST, IERROR
```

MPI_COMM_ISHRINK is a nonblocking variant of MPI_COMM_SHRINK. With the exception of its nonblocking behavior, the semantics of MPI_COMM_ISHRINK are as if MPI_COMM_SHRINK was executed at the time MPI_COMM_ISHRINK is called. All restrictions and assumptions for nonblocking collective operations (see Section 6.12) apply to MPI_COMM_ISHRINK and the returned request.

Note that, as with MPI_COMM_IDUP (see Section 7.4.2), it is erroneous to use newcomm before request has completed.

MPI_COMM_GET_FAILED(comm, failedgrp)

IN	comm	communicator (handle)
OUT	failedgrp	group of failed processes (handle)

C binding

```
int MPI_Comm_get_failed(MPI_Comm comm, MPI_Group *failedgrp)
```

Fortran 2008 binding

```
MPI_Comm_get_failed(comm, failedgrp, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    TYPE(MPI_Group), INTENT(OUT) :: failedgrp
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_GET_FAILED(COMM, FAILEDGRP, IERROR)
    INTEGER COMM, FAILEDGRP, IERROR
```

This local operation returns the group failedgrp of processes from the communicator comm that are locally known to have failed. The failedgrp can be empty, that is, equal to MPI_GROUP_EMPTY.

For any two groups obtained from calls to that routine at the same MPI process with the same comm, the smallest group is a prefix of the largest group, that is, the same processes have the same ranks in the two groups up to the size of the smallest group.

Advice to users. MPI makes no assumption about asynchronous progress of the failure detection. A valid MPI implementation may choose to update the group of locally known failed MPI processes only when it enters a function that must raise a fault tolerance error.

It is possible that only the calling MPI process has detected the reported failure. If global knowledge is necessary, MPI processes detecting failures should use the call MPI_COMM_REVOKE. (*End of advice to users.*)

MPI_COMM_ACK_FAILED(comm, nack, nacked)

IN	comm	communicator (handle)
IN	nack	Maximum number of process failures to acknowledge (integer)
OUT	nacked	Number of process failures acknowledged (integer)

C binding

```
int MPI_Comm_ack_failed(MPI_Comm comm, int nack, int *nacked)
```

Fortran 2008 binding

```
MPI_Comm_ack_failed(comm, nack, nacked, ierror)
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(IN) :: nack
    INTEGER, INTENT(OUT) :: nacked
```

```
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_ACK_FAILED(COMM, NACK, NACKED, IERROR)
    INTEGER COMM, NACK, NACKED, IERROR
```

This local operation gives the users a way to acknowledge locally notified failures on comm. The operation acknowledges the first nack process failures on comm, that is, it acknowledges the failure of members with a rank lower than nack in the group that would be produced by a concurrent call to MPI_COMM_GET_FAILED on the same comm.

The operation also sets the value of nacked to the current number of acknowledged process failures in comm, that is, a process failure has been acknowledged on comm if and only if the rank of the process is lower than nacked in the group that would be produced by a subsequent call to MPI_COMM_GET_FAILED on the same comm.

nacked can be larger than nack when process failures have been acknowledged in a prior call to MPI_COMM_ACK_FAILED.

After an MPI process failure is acknowledged on comm, unmatched MPI_ANY_SOURCE receive operations on the same comm that would have raised an error of class MPI_ERR_PROC_FAILED_PENDING (see Section **??**) proceed without further raising errors due to this acknowledged failure. Also, MPI_COMM_AGREE on the same comm will not raise an error of class MPI_ERR_PROC_FAILED due to this acknowledged failure (according to the specification found later in this section).

Advice to users. One may query, without side effect, for the number of currently aknowledged process failures in comm by supplying 0 in nack. Conversely, one may unconditionally acknowledge all currently known process failures in comm by supplying the size of the group of comm in nack. Note that the number of acknowledged processes, as returned in nacked, can be smaller or larger than the value supplied in nack; It is however never larger than the size of the group returned by a subsequent call to MPI_COMM_GET_FAILED.

Calling MPI_COMM_ACK_FAILED on a communicator with failed MPI processes has no effect on collective operations (except for MPI_COMM_AGREE). If a collective operation would raise an error due to the communicator containing a failed process (as defined in Section **??**), it can continue to raise an error even after the failure has been acknowledged. In order to use collective operations between MPI processes of a communicator that contains failed MPI processes, users should create a new communicator by calling MPI_COMM_SHRINK. (*End of advice to users.*)

MPI_COMM_AGREE(comm, flag)

IN	comm	communicator (handle)
INOUT	flag	bitwise 'AND' of contributed values (integer)

C binding

```
int MPI_Comm_agree(MPI_Comm comm, int *flag)
```

Fortran 2008 binding

```
MPI_Comm_agree(comm, flag, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(INOUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_AGREE(COMM, FLAG, IERROR)
    INTEGER COMM, FLAG, IERROR
```

The purpose of this collective communication is to agree on the integer value flag and on the group of failed processes in comm.

On completion, all non-failed MPI processes have agreed to set the output integer value of flag to the result of a bitwise *'AND'* operation over the contributed input values of flag. If comm is an intercommunicator, the value of flag is a bitwise *'AND'* operation over the values contributed by the remote group.

When an MPI process fails before contributing to the operation, the flag is computed ignoring its contribution, and MPI_COMM_AGREE raises an error of class MPI_ERR_PROC_FAILED. However, if all MPI processes have acknowledged this failure prior to the call to MPI_COMM_AGREE, using MPI_COMM_ACK_FAILED, the error related to this failure is not raised. When an error of class MPI_ERR_PROC_FAILED is raised, it is consistently raised at all MPI processes, in both the local and remote groups (if applicable).

After MPI_COMM_AGREE raised an error of class MPI_ERR_PROC_FAILED, the group produced by a subsequent call to MPI_COMM_GET_FAILED on comm contains every MPI process that didn't contribute to the computation of flag.

> *Advice to users.* Using a combination of MPI_COMM_ACK_FAILED and MPI_COMM_AGREE as illustrated in Example 16.3, users can propagate and synchronize the knowledge of failures across all MPI processes in comm. When MPI_SUCCESS is returned locally from MPI_COMM_AGREE, the operation has not raised an error of class MPI_ERR_PROC_FAILED at any MPI process and thereby returned MPI_SUCCESS at all other MPI processes. (*End of advice to users.*)

This function never raises an error of class MPI_ERR_REVOKED. The defined semantics of MPI_COMM_AGREE are maintained when comm is revoked, or when the group of comm contains failed MPIprocesses.

> *Advice to users.* MPI_COMM_AGREE is a collective operation, even when comm is revoked. (*End of advice to users.*)

MPI_COMM_IAGREE(comm, flag, request)

IN	comm	communicator (handle)
INOUT	flag	bitwise 'AND' of contributed values (integer)
OUT	request	communication request (handle)

C binding

```
int MPI_Comm_iagree(MPI_Comm comm, int *flag, MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Comm_iagree(comm, flag, request, ierror)
```

```
    TYPE(MPI_Comm), INTENT(IN) :: comm
    INTEGER, INTENT(INOUT), ASYNCHRONOUS :: flag
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_COMM_IAGREE(COMM, FLAG, REQUEST, IERROR)
    INTEGER COMM, FLAG, REQUEST, IERROR
```

This function has the same semantics as MPI_COMM_AGREE except that it is non-blocking.

16.3.2 One-Sided Functions

MPI_WIN_REVOKE(win)

IN	win	window object (handle)

C binding

```
int MPI_Win_revoke(MPI_Win win)
```

Fortran 2008 binding

```
MPI_Win_revoke(win, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_REVOKE(WIN, IERROR)
    INTEGER WIN, IERROR
```

This function notifies all MPI processes in the group associated with the window win that this window is revoked. The revocation of a window by any MPI process completes RMA operations on win at all MPI processes and RMA synchronizations on win raise an error of class MPI_ERR_REVOKED. This function is not collective and therefore does not have a matching call on remote MPI processes. All non-failed MPI processes belonging to win will be notified of the revocation despite failures.

A window is revoked at a given MPI process either when MPI_WIN_REVOKE is locally called on it, or when any MPI operation on win raises an error of class MPI_ERR_REVOKED at that process. Once a window has been revoked at an MPI process, all subsequent RMA operations on that window are considered local and RMA synchronizations must complete by raising an error of class MPI_ERR_REVOKED at that process. In addition, the current epoch is closed and RMA operations originating from this MPI process are interrupted and completed with undefined outputs.

MPI_WIN_IS_REVOKED(win, flag)

IN	win	window object (handle)
OUT	flag	true if the window is revoked (logical)

C binding

```
int MPI_Win_is_revoked(MPI_Win win, int *flag)
```

Fortran 2008 binding

```
MPI_Win_is_revoked(win, flag, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_IS_REVOKED(WIN, FLAG, IERROR)
    INTEGER WIN, IERROR
    LOGICAL FLAG
```

Returns flag = true if the window associated with the handle win is revoked at the calling process. It returns flag = false otherwise. The operation is local.

Advice to users. In a multithreaded application, a thread calling MPI_WIN_IS_REVOKED may return flag = true before the operation that raises the first exception of class MPI_ERR_REVOKED has completed in a concurrent thread. (*End of advice to users.*)

MPI_WIN_GET_FAILED(win, failedgrp)

IN	win	window object (handle)
OUT	failedgrp	(handle)

C binding

```
int MPI_Win_get_failed(MPI_Win win, MPI_Group *failedgrp)
```

Fortran 2008 binding

```
MPI_Win_get_failed(win, failedgrp, ierror)
    TYPE(MPI_Win), INTENT(IN) :: win
    TYPE(MPI_Group), INTENT(OUT) :: failedgrp
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_WIN_GET_FAILED(WIN, FAILEDGRP, IERROR)
    INTEGER WIN, FAILEDGRP, IERROR
```

This local operation returns the group failedgrp of MPI processes from the window win that are locally known to have failed. The failedgrp can be empty, that is, equal to MPI_GROUP_EMPTY.

Advice to users. MPI makes no assumption about asynchronous progress of the failure detection. A valid MPI implementation may choose to update the group of locally known failed MPI processes only when it enters a synchronization function and must raise a fault tolerance error. (*End of advice to users.*)

Advice to users. It is possible that only the calling MPI process has detected the reported failure. If global knowledge is necessary, MPI processes detecting failures should use the call MPI_WIN_REVOKE. (*End of advice to users.*)

16.3.3 I/O Functions

MPI_FILE_REVOKE(fh)

IN	fh	file (handle)

C binding

```
int MPI_File_revoke(MPI_File fh)
```

Fortran 2008 binding

```
MPI_File_revoke(fh, ierror)
    TYPE(MPI_File), INTENT(IN) :: fh
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_REVOKE(FH, IERROR)
    INTEGER FH, IERROR
```

This function notifies all MPI processes in the group associated with the file handle fh that this file handle is revoked. The revocation of a file handle by any MPI process completes non-local MPI operations on fh at all MPI processes by raising an error of class MPI_ERR_REVOKED. This function is not collective and therefore does not have a matching call on remote MPI processes. All non-failed MPI processes belonging to fh will be notified of the revocation despite failures.

A file handle is revoked at a given MPI process either when MPI_FILE_REVOKE is locally called on it, or when any MPI operation on fh raises an error of class MPI_ERR_REVOKED at that process. Once a file handle has been revoked at an MPI process, all subsequent non-local operations on that file handle are considered local and must complete by raising an error of class MPI_ERR_REVOKED at that process.

MPI_FILE_IS_REVOKED(fh, flag)

IN	fh	file (handle)
OUT	flag	true if the file handle is revoked (logical)

C binding

```
int MPI_File_is_revoked(MPI_File fh, int *flag)
```

Fortran 2008 binding

```
MPI_File_is_revoked(fh, flag, ierror)
```

```
    TYPE(MPI_File), INTENT(IN) :: fh
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_FILE_IS_REVOKED(FH, FLAG, IERROR)
    INTEGER FH, IERROR
    LOGICAL FLAG
```

Returns flag = true if the file handle associated with fh is revoked at the calling process. It returns flag = false otherwise. The operation is local.

> *Advice to users.* In a multithreaded application, a thread calling MPI_FILE_IS_REVOKED may return flag = true before the operation that raises the first exception of class MPI_ERR_REVOKED has completed in a concurrent thread. (*End of advice to users.*)

16.4 Fault Tolerance Error Codes and Classes

Among the error classes defined in Section 9.4, the following are **fault tolerance error** classes:

MPI_ERR_PROC_FAILED	The operation could not complete because of an MPI process failure (a fail-stop failure).
MPI_ERR_PROC_FAILED_PENDING	The operation was interrupted by an MPI process failure (a fail-stop failure). The request is still pending and the operation may be completed later.
MPI_ERR_REVOKED	The communication object used in the operation has been revoked.

Table 16.1: Fault tolerance error classes

16.5 Examples

16.5.1 Safe Communicator Creation

The example below illustrates how a new communicator can be safely created despite disruption by MPI process failures. A child communicator is created with MPI_COMM_SPLIT, then the global success of the operation is verified with MPI_COMM_AGREE. If any MPI g failed to create the child communicator handle, all MPI processes are notified by the value of the integer agreed on. MPI processes that had successfully created the child communicator handle destroy it, as it cannot be used consistently.

Example 16.1 Fault Tolerant Communicator Split Example

```
int Comm_split_consistent(MPI_Comm parent, int color, int key, MPI_Comm* child)
{
```

```
    rc = MPI_Comm_split(parent, color, key, child);
    split_ok = (MPI_SUCCESS == rc);
    MPI_Comm_agree(parent, &split_ok);
    if(split_ok) {
        /* All surviving processes have created the "child" comm
         * It may contain supplementary failures and the first
         * operation on it may raise an error, but it is a
         * workable object that will yield well specified outcomes */
        return MPI_SUCCESS;
    }
    else {
        /* At least one process did not create the child comm properly
         * if the local process did succeed in creating it, it disposes
         * of it, as it is a broken, inconsistent object */
        if(MPI_SUCCESS == rc) {
            MPI_Comm_free(child);
        }
        return MPI_ERR_PROC_FAILED;
    }
}
```

16.5.2 Obtaining the consistent group of failed processes

Users can invoke MPI_COMM_GET_FAILED, MPI_WIN_GET_FAILED, to obtain the group of failed MPI processes, as detected at the local MPI process. However, these operations are local, thereby the invocation of the same function at another MPI process can result in a different group of failed processes being returned.

In the following examples, we illustrate two different approaches that permit obtaining the consistent group of failed MPI processes across all MPI processes of a communicator. The first one employs MPI_COMM_SHRINK to create a temporary communicator excluding failed MPI processes. The second one employs MPI_COMM_AGREE to synchronize the set of acknowledged failures.

Example 16.2 Fault-Tolerant Consistent Group of Failures Example (Shrink variant)

```
Comm_failure_allget(MPI_Comm c, MPI_Group * g) {
    MPI_Comm s; MPI_Group c_grp, s_grp;

    /* Using shrink to create a new communicator, the underlying
     * group is necessarily consistent across all processes, and excludes
     * all processes detected to have failed before the call */
    MPI_Comm_shrink(c, &s);
    /* Extracting the groups from the communicators */
    MPI_Comm_group(c, &c_grp);
    MPI_Comm_group(s, &s_grp);
    /* s_grp is the group of still alive processes, we want to
     * return the group of failed processes. */
    MPI_Group_difference(c_grp, s_grp, g);
```

```
    MPI_Group_free(&c_grp); MPI_Group_free(&s_grp);
    MPI_Comm_free(&s);
}
```

Example 16.3 Fault-Tolerant Consistent Group of Failures Example (Agree variant)

```
Comm_failure_allget2(MPI_Comm c, MPI_Group * g) {
    int rc; int T=1;
    int size; int nacked;
    MPI_Group gf;
    int ranges[3] = {0, 0, 1};

    MPI_Comm_size(c, &size);

    do {
        /* this routine is not pure: calling MPI_Comm_ack_failed
         * affects the state of the communicator c */
        MPI_Comm_ack_failed(c, size, &nacked);
        /* we simply ignore the T value in this example */
        rc = MPI_Comm_agree(c, &T);
    } while( rc != MPI_SUCCESS );
    /* after this loop, MPI_Comm_agree has returned MPI_SUCCESS at
     * all processes, so all processes have Acknowledged the same set of
     * failures. Let's get that set of failures in the g group. */
    if( 0 == nacked ) {
        *g = MPI_GROUP_EMPTY;
    }
    else {
        MPI_Comm_get_failed(c, &gf);
        ranges[1] = nacked - 1;
        MPI_Group_range_incl(gf, 1, ranges, g);
        MPI_Group_free(&gf);
    }
}
```

16.5.3 Fault-Tolerant Master/Worker

The example below presents a master code that handles worker failures by discarding failed worker MPI processes and resubmitting the work to the remaining workers. It demonstrates the different failure cases that may occur when posting receptions from MPI_ANY_SOURCE as discussed in the advice to users in Section ??.

Example 16.4 Fault-Tolerant Master Example

```
int master(void)
{
    MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
    MPI_Comm_size(comm, &size);
```

```
    MPI_Comm_group(comm, &gcomm);

    /* ... submit the initial work requests ... */

    /* Progress engine: Get answers, send new requests,
       and handle process failures */
    MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, &req );
    while( (active_workers > 0) && work_available ) {
        rc = MPI_Wait( &req, &status );
        if( MPI_SUCCESS == rc ) {
            /* ... process the answer and update work_available ... */
        }
        else {
            MPI_Error_class(rc, &ec);
            if( (MPI_ERR_PROC_FAILED == ec) ||
                (MPI_ERR_PROC_FAILED_PENDING == ec) ) {
                /* We ack the full size of comm, so we will ack
                 * unconditionally. Variable gsize will contain all
                 * currently known failures. */
                MPI_Comm_ack_failed(comm, size, &gsize);

                /* ... find the lost work and requeue it ... */
                MPI_Comm_get_failed(comm, &g);
                granks = (int*) calloc(active_workers-gsize-1, sizeof(int));
                cranks = (int*) calloc(active_workers-gsize-1, sizeof(int));
                for(i = active_workers; i < gsize; i++)
                    granks[i-active_workers] = i;
                MPI_Group_translate_ranks(g, gsize, granks, gcomm, cranks);
                /* iterate over newly failed procs */
                for(i = active_workers; i < gsize; i++) {
                    /* resubmit the work */
                }
                free(cranks); free(granks);
                MPI_Group_free(&g);

                active_workers = size - gsize - 1;

                /* no need to repost when the request is still pending */
                if( ec == MPI_ERR_PROC_FAILED_PENDING )
                    continue;
            }
        }
        /* get ready to receive more notifications from workers */
        MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, &req );
    }
    /* ... cancel request and cleanup ... */
}
```

16.5.4 Fault-Tolerant Iterative Refinement

The example below demonstrates a method of fault tolerance for detecting and handling failures. At each iteration, the algorithm checks the return code of the MPI_ALLREDUCE. If the return code indicates a process failure for at least one MPI process, the algorithm revokes the communicator, agrees on the presence of failures, and shrinks it to create a new communicator. By calling MPI_COMM_REVOKE, the algorithm ensures that all MPI processes will be notified of process failure and enter the MPI_COMM_AGREE. If an MPI process fails, the algorithm must complete at least one more iteration to ensure a correct answer.

Example 16.5 Fault-tolerant iterative refinement with shrink and agreement

```
while( gnorm > epsilon ) {
    /* Add a computation iteration to converge and
       compute local norm in lnorm */
    rc = MPI_Allreduce(&lnorm, &gnorm, 1, MPI_DOUBLE, MPI_MAX, comm);
    MPI_Error_class(rc, &ec);

    if( (MPI_ERR_PROC_FAILED == ec) ||
        (MPI_ERR_REVOKED == ec) ||
        (gnorm <= epsilon) ) {

        /* This process detected a failure, but other processes may have
         * proceeded into the next MPI_Allreduce. Since this process
         * will not match that following MPI_Allreduce, these other
         * processes would be at risk of deadlocking. This process thus
         * calls MPI_Comm_revoke to interrupt other processes and notify
         * them that it has detected a failure and is leaving the
         * failure free execution path to go into recovery. */
        if( MPI_ERR_PROC_FAILED == ec )
            MPI_Comm_revoke(comm);

        /* About to leave: let's be sure that everybody
           received the same information */
        allsucceeded = (rc == MPI_SUCCESS);
        rc = MPI_Comm_agree(comm, &allsucceeded);
        MPI_Error_class(rc, &ec);
        if( ec == MPI_ERR_PROC_FAILED || !allsucceeded ) {
            MPI_Comm_shrink(comm, &comm2);
            MPI_Comm_free(comm); /* Release the revoked communicator */
            comm = comm2;
            gnorm = epsilon + 1.0; /* Force one more iteration */
        }
    }
}
```

Bibliography

Index

MPI_ANY_SOURCE, 2, 3, 12, 19
MPI_Comm, 8–13
MPI_COMM_NULL, 4–6
MPI_COMM_WORLD, 2, 5
MPI_ERR_PROC_FAILED, 3, 4, 6–8, 10, 12, 13, 17
MPI_ERR_PROC_FAILED_PENDING, 3, 10, 12, 17
MPI_ERR_REVOKED, 3, 4, 6–10, 13–17
MPI_ERR_SPAWN, 5
MPI_ERRORS_ARE_FATAL, 2
MPI_File, 16
MPI_FILE_NULL, 8
MPI_FT, 2
MPI_Group, 10, 15
MPI_GROUP_EMPTY, 11, 15
MPI_Request, 13
MPI_SEEK_SET, 7
MPI_SUCCESS, 4, 6, 7, 13
MPI_Win, 14, 15
MPI_WIN_NULL, 7

EXAMPLES:Comm_failure_allget example, 18
EXAMPLES:Comm_failure_allget2 example, 19
EXAMPLES:Fault-tolerant iterative refinement with shrink and agreement, 21
EXAMPLES:Master example, 19
EXAMPLES:MPI_COMM_ACK_FAILED, 19
EXAMPLES:MPI_COMM_AGREE, 17, 19, 21
EXAMPLES:MPI_COMM_FREE, 17, 18, 21
EXAMPLES:MPI_COMM_GET_FAILED, 19
EXAMPLES:MPI_COMM_GROUP, 18
EXAMPLES:MPI_COMM_REVOKE, 21
EXAMPLES:MPI_COMM_SHRINK, 18, 21
EXAMPLES:MPI_COMM_SPLIT, 17
EXAMPLES:MPI_GROUP_DIFFERENCE, 18
EXAMPLES:MPI_GROUP_FREE, 18

"mpi_error_range", 2
"mpi_error_uniform", 4
"coll", 4
"create", 4
"global", 3
"group", 3
"local", 4
"operation", 3

MPI_ALLREDUCE, 21
MPI_BCAST, 3
MPI_COMM_ACCEPT, 5
MPI_COMM_ACK_FAILED, 12, 13
MPI_COMM_ACK_FAILED(comm, nack, nacked), 11
MPI_COMM_AGREE, 4, 7, 8, 12–14, 17, 18, 21
MPI_COMM_AGREE(comm, flag), 12
MPI_COMM_CONNECT, 5
MPI_COMM_DISCONNECT, 6
MPI_COMM_DUP, 3, 4
MPI_COMM_FREE, 4
MPI_COMM_GET_FAILED, 12, 13, 18
MPI_COMM_GET_FAILED(comm, failedgrp), 11
MPI_COMM_GET_PARENT, 5
MPI_COMM_IAGREE, 8
MPI_COMM_IAGREE(comm, flag, request), 13
MPI_COMM_IDUP, 10
MPI_COMM_IS_REVOKED, 9
MPI_COMM_IS_REVOKED(comm, flag), 9
MPI_COMM_ISHRINK, 10
MPI_COMM_ISHRINK(comm, newcomm, request), 10
MPI_COMM_JOIN, 5
MPI_COMM_REVOKE, 8, 11, 21
MPI_COMM_REVOKE(comm), 8

MPI_COMM_SHRINK, 8–10, 12, 18
MPI_COMM_SHRINK(comm, newcomm), 9
MPI_COMM_SPAWN, 5
MPI_COMM_SPAWN_MULTIPLE, 5
MPI_COMM_SPLIT, 3, 10, 17
MPI_FILE_CLOSE, 7, 8
MPI_FILE_IS_REVOKED, 17
MPI_FILE_IS_REVOKED(fh, flag), 16
MPI_FILE_REVOKE, 16
MPI_FILE_REVOKE(fh), 16
MPI_FILE_SEEK, 7
MPI_GET, 6
MPI_INIT, 5
MPI_PUT, 6
MPI_WIN_FLUSH, 6
MPI_WIN_FLUSH_LOCAL, 6
MPI_WIN_FLUSH_LOCAL_ALL, 6
MPI_WIN_FREE, 7
MPI_WIN_GET_FAILED, 18
MPI_WIN_GET_FAILED(win, failedgrp), 15
MPI_WIN_IS_REVOKED, 15
MPI_WIN_IS_REVOKED(win, flag), 15
MPI_WIN_REVOKE, 7, 14, 16
MPI_WIN_REVOKE(win), 14

TERM:error handling
- fault tolerance, **1**
  - ack, 12
  - agree, 13, 14
  - communicator, **3**, **8**
  - dynamic process, **5**
  - fault tolerance error, 1, **17**
  - I/O, **7**, **16**
  - inquiry, 2
  - mitigation, **8**
  - notification, **2**, 17
  - one-sided, **6**, **14**
  - revoke, 8, 9, 14–17
  - shrink, 9, 10
- process failure, **1**