D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

September 13, 2011

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 17

Process Fault Tolerance

17.1 Introduction

MPI processes may fail at any time during execution. Long running and large scale applications are at increased risk of encountering process failure(s) during normal execution. This chapter introduces the MPI features that support the development of process fault tolerant applications and libraries.

Process fault tolerant applications must be sure to manage the error handlers associated with the communication handles. The default error handler is MPI_ERRORS_ARE_FATAL, as defined in Section 8.3.

> *Advice to implementors.* If the default error handler is not replaced by the application then many of the functions and semantics in this chapter may be avoided as they focus upon the continued use of the MPI interface after an error. Such a situation is not possible given the default error handler of MPI_ERRORS_ARE_FATAL on MPI_COMM_WORLD. If an implementation cannot provide the necessary functionality described in this chapter then it should return MPI_ERR_UNSUPPORTED_OPERATION for those operations defined in this chapter, and never return the error class MPI_ERR_RANK_FAIL_STOP from any MPI operation. (*End of advice to implementors.*)

17.2 MPI Terms and Conventions

When discussing fault tolerance procedures the following semantic terms are used.

error An error is the deviation of expected behavior from correct operation of the system (e.g., MPI library, MPI operation). Errors are caused by **faults** in one or more components of the system (e.g., memory corruption, physical defect) [1].

failure A failure occurs when the intended function of the system (e.g., MPI library, MPI operation) cannot be delivered because of one or more errors [1].

fail-stop process failure A process failure in which the MPI process permanently stops executing, and its internal state is lost [7].

alive process A process that is not failed and in the running state.

failed process A process that is not alive due to a fail-stop process failure.

recognized failed process A failed process that has been globally determined as failed by the use of a collective validate routine (e.g., MPI_COMM_VALIDATE).

collectively active A communicator or file handle that is able to successfully perform collective operations.

collectively inactive A communicator or file handle that is not able to perform collective operations possibly due to process failure.

17.3 Process Fault Detection

MPI will provide the ability to detect process failures and will guarantee that eventually all alive processes will know about the failure. The query operations defined in Section 17.4 allow the application to query for the failed set of processes in a communication group. Additional semantics regarding communication involving failed processes are defined later in this chapter.

It is possible that MPI mistakenly identifies a process as failed when it is not failed. In this situation the MPI library will exclude the mistakenly identified failed process from the MPI universe, and eventually all alive processes will see this process as failed. The MPI implementation is allowed to terminate the process that was mistakenly identified as failed.

Rationale. This means that MPI provides something like an **eventually perfect** failure detector for fail-stop process failures [2]. An eventually perfect failure detector is both **strongly complete** and **eventually strongly accurate**.

Strong completeness is defined as: "Eventually every process that crashes is permanently suspected by every correct process" [2]. In essence this means that eventually every failed process will be known to all alive processes. Without strong completeness communication operations with a failed process may not complete with an error, so it is possible that a process communicating with a failed process may wait indefinitely in, e.g., a blocking receive operation.

Eventual strong accuracy is defined as: "There is a time after which correct processes are not suspected by any correct process" [2]. Depending on the system architecture, it may be impossible to correctly determine if a process is failed or slow [4]. Eventual strong accuracy allows for unreliable failure detectors that may mistakenly suspect a process as failed when it is not failed [2].

If a process failure was reported to the application and the process is later found to be alive then MPI will exclude the process from the MPI universe. Resolving the mistake by excluding the process from the MPI universe is similar to the technique used by the group membership protocol in [6]. This additional constraint allows for consistent reporting of error states to the local process. Without this constraint the application would not be able to trust the MPI implementation when it reports process failure errors. Once an alive process receives notification of a failed peer process, then it may continue under the assumption that the process is failed. (*End of rationale.*)

Advice to users. The strong completeness condition of the failure detector allows the MPI implementation some flexibility in managing the performance costs involved with process failure detection and notification. As such, it is possible that for a period of time, some alive processes in the MPI universe know of process failures that other alive processes do not. Additionally, if a process was mistakenly reported as failed it is possible that for some period of time a subset of processes interact with the process normally, while others see it as failed. Eventually all processes in the MPI universe will become aware of the process failure. (*End of advice to users.*)

Advice to implementors. An MPI implementation may choose to provide a stronger failure detector (i.e., perfect failure detector), but is not required to do so. This may be possible for MPI implementations targeted at synchronous systems [3]. (*End of advice to implementors.*)

17.4 Querying for Failed Processes

At each process, the MPI implementation keeps track of failed processes. Query functions are provided to allow the user to determine which processes associated with a specific communicator, file or window have failed. These functions return a group comprising the failed processes.

17.4.1 Communicators

MPI_COMM_GROUP_FAILED(comm, failed)

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

```
int MPI_Comm_group_failed(MPI_Comm comm, MPI_Group *failed)
```

```
MPI_COMM_GROUP_FAILED(COMM, FAILED, IERROR)
    INTEGER COMM, FAILED, IERROR
```

MPI_COMM_GROUP_FAILED is a process local operation that creates a group comprising processes in the communicator comm that were known to be failed by the process at the time of the call. If comm is an intercommunicator, then the group contains the failed processes of the local group. Failed processes in the remote group of an intercommunicator can be queried using MPI_COMM_REMOTE_GROUP_FAILED, shown below.

MPI_COMM_REMOTE_GROUP_FAILED(comm, failed)

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

```
int MPI_Comm_remote_group_failed(MPI_Comm comm, MPI_Group *failed)
```

```
MPI_COMM_REMOTE_GROUP_FAILED(COMM, FAILED, IERROR)
    INTEGER COMM, FAILED, IERROR
```

17.4.2 Windows

The following function returns the group of failed processes associated with a window.

MPI_WIN_GET_GROUP_FAILED(win, failed)

IN	win	window object (handle)
OUT	failed	group of failed processes (handle)

```
int MPI_Win_get_group_failed(MPI_Win win, MPI_Group *failed)
```

```
MPI_WIN_GET_GROUP_FAILED(WIN, FAILED, IERROR)
    INTEGER WIN, FAILED, IERROR
```

17.4.3 Files

The following function returns the group of failed processes associated with a file.

MPI_FILE_GET_GROUP_FAILED(fh, failed)

IN	fh	file handle (handle)
OUT	failed	group of failed processes (handle)

```
int MPI_File_get_group_failed(MPI_File fh, MPI_Group *failed)
```

```
MPI_FILE_GET_GROUP_FAILED(FH, FAILED, IERROR)
    INTEGER FH, FAILED, IERROR
```

17.4.4 Examples

Example 17.1 Determine whether rank 5 has failed in the communicator.

```
/* Get MPI_COMM_WORLD's group */
MPI_Comm_group(MPI_COMM_WORLD, &comm_world_group);

/* Get the failed processes from MPI_COMM_WORLD */
MPI_Comm_group_failed(MPI_COMM_WORLD, &failed_group);

/* Translate MPI_COMM_WORLD rank of process 5 to rank in failed_group */
ranks1 = 5;
MPI_Group_translate_ranks(comm_world_group, 1, &ranks1, failed_group,
                          &ranks2);

if (ranks2 != MPI_UNDEFINED)
    printf("Rank 5 has failed\n");
```

Example 17.2 Determine whether any new processes have failed in the communicator.

```
MPI_Comm_group_failed(MPI_COMM_WORLD, &failed_group1);

/* Do some work... */

MPI_Comm_group_failed(MPI_COMM_WORLD, &failed_group2);

/* See if any processes failed during "work" */
MPI_Group_compare(failed_group1, failed_group2, &result);

if (result == MPI_IDENT)
    printf("No new failed processes have been detected\n");
else
    printf("New failed processes have been detected\n");
```

Example 17.3 Determine which new processes have failed in the communicator.

```
MPI_Comm_group_failed(MPI_COMM_WORLD, &failed_group1);

/* Do some work... */

MPI_Comm_group_failed(MPI_COMM_WORLD, &failed_group2);

/* Get group of processes that failed while we did "work" */
MPI_Group_difference(failed_group1, failed_group2, &newly_failed_group);
/* newly_failed_group contains processes that failed during "work" */
```

Example 17.4 Iterate over failed processes in the communicator.

```
/* Get group of failed processes */
MPI_Comm_group_failed(MPI_COMM_WORLD, &failed_group);

/* Allocate arrays for rank translation and initialize input array */
MPI_Group_size(failed_group, &num_failed);
failed_ranks     = malloc(num_failed * sizeof(int));
comm_world_ranks = malloc(num_failed * sizeof(int));
for (i = 0; i < num_procs; ++i)
    failed_ranks[i] = i;

/* Get MPI_COMM_WORLD's group */
MPI_Comm_group(MPI_COMM_WORLD, &comm_world_group);
/* Get MPI_COMM_WORLD ranks of processes in failed_group */
MPI_Group_translate_ranks(failed_group, num_failed, &failed_ranks,
                          comm_world_group, &comm_world_ranks);

for (i = 0; i < num_procs; ++i)
    printf("Process %d in MPI_COMM_WORLD has failed\n",
           comm_world_ranks[i]);
```

17.5 MPI Environmental Management

MPI errors are associated with the call site, and should be indicated as such in the return code, status, associated error handler, or an appropriate combination thereof. Errors that have been detected, but are not associated with the current call site should be postponed and delivered on a subsequent, related call.

> *Rationale.* These semantics allow a process to continue running without being interrupted by the failure of processes with which they may never or rarely communicate. (*End of rationale.*)

> *Advice to users.* A newly created communicator inherits the error handler that is associated with the parent communicator. Libraries should take care to set the error handler appropriately for their library directly after communicator creation. This allows the library to have its own error handler behavior separate from the calling process. (*End of advice to users.*)

17.5.1 Error Codes and Classes

The following error class is added:

MPI_ERR_RANK_FAIL_STOP	A process in the operation is failed (a fail-stop failure)

Table 17.1: Additional process fault tolerance error class

The MPI_ERR_RANK_FAIL_STOP error class indicates that a rank participating in the operation was detected as failed (fail-stop) either before or during the operation.

17.5.2 Startup

If a process failure or other error occurs before or during MPI_INIT then MPI_INIT should try to return an error code, and not abort by default. If the next MPI operation is not MPI_COMM_SET_ERRHANDLER (or MPI_COMM_CREATE_ERRHANLDER followed by MPI_COMM_SET_ERRHANDLER) then the MPI implementation will behave as if MPI_ERRORS_ARE_FATAL was set on MPI_COMM_WORLD.

> *Rationale.* For applications that assume MPI_ERRORS_ARE_FATAL semantics, then the failure that occurred during MPI_INIT is delayed until the next MPI function call. If the application intends to handle the failure then they are provided an opportunity to replace the default error handler before calling subsequent MPI operations, and then decide if and how to continue. (*End of rationale.*)

> *Advice to implementors.* A high quality implementation will, to the extent possible, return an appropriate error code and not abort if MPI_INIT is not able to complete successfully. A critical error may cause even a high quality MPI implementation to abort before or during MPI_INIT. (*End of advice to implementors.*)

MPI_FINALIZE will complete normally even in the presence of process failures, regardless of when the process failure occurs.

Advice to users. Considering the example in Chapter 8.7, the process with rank 0 in MPI_COMM_WORLD may have failed before, during or after the call to MPI_FINALIZE. MPI can only detect failure up to the point of MPI_FINALIZE and provides no support for fault tolerance after MPI_FINALIZE. So applications are encouraged to implement all rank specific code before the call to MPI_FINALIZE to handle the case where rank 0 in MPI_COMM_WORLD fails. (*End of advice to users.*)

Advice to implementors. Without process failure, `mpiexec` should return the exit code of rank 0. In the presence of process failure, `mpiexec` should return the exit code from the lowest ranked process that exits after calling MPI_FINALIZE. If no process returns from MPI_FINALIZE then `mpiexec` should return the exit code specified in the last call to MPI_ABORT. If multiple processes call MPI_ABORT with different errcode values then the last errcode should be used. The user should be aware of the unavoidable possibility for nondeterminism in this case. If no process calls either MPI_FINALIZE or MPI_ABORT, then `mpiexec` should return the exit code of rank 0. Given this advice, a fault tolerant application will eventually call either MPI_FINALIZE or MPI_ABORT in the remaining processes. After a process failure, a fault tolerant application may run to successful completion, and is allowed to properly set the exit code of their application. (*End of advice to implementors.*)

17.6 Point-to-Point Communication

Point-to-point communication with a failed process will not hang indefinitely but will eventually complete. An error code of the class MPI_ERR_RANK_FAIL_STOP will be returned for all point-to-point communication operations with a process that has been detected as failed. One exception is that an MPI implementation may complete, as normal, receive operations with messages sent by the failed process before it failed. The extent to which an MPI implementation can deliver such internally received messages is implementation dependent.

Rationale. Messages sent from a process before it failed might have been internally received by the MPI implementation at receiving process but not yet delivered to the application. The MPI implementation can complete receive operations with such matching internally received messages, even if the receive operations are posted after the processes failure has been detected. (*End of rationale.*)

Advice to implementors. The implementation must ensure that ordering semantics are preserved when completing posted receives from internal buffers. For example, consider the case where two receives are posted to the same communicator with matching tags and sources, and that the intended source of the message is a failed process. If the first receive is completed with an error because no matching message was in an internal buffer, then the second receive must also be completed with an error, even if a matching message was internally received immediately before the second receive was posted. (*End of advice to implementors.*)

If a process failure affects a point-to-point operation with a buffer marked as OUT or INOUT then the contents of the buffer are **undefined**.

When a process detects a new process failure, the ability to perform wildcard receives (i.e., receives where MPI_ANY_SOURCE has been specified for the source parameter) will be disabled on all communicators that contain the failed process. When wildcard receives are disabled on a communicator, all pending wildcard receive operations on that communicator are completed and an error with class MPI_ERR_RANK_FAIL_STOP will be returned for those operations. Any new wildcard receive operations posted to a communicator with disabled wildcard receives will be immediately completed and return an error code of the class MPI_ERR_RANK_FAIL_STOP.

Wildcard receives can be re-enabled with the MPI_COMM_REENABLE_ANY_SOURCE function described below.

> *Rationale.* The fault semantics for a receive using the MPI_ANY_SOURCE wildcard were selected to be as described since the MPI implementation is unable to infer if the failed process was important to the completion of the receive operation. So such a decision should be left to the application. Additionally, regarding internal buffering, if the MPI implementation has access to an internal receive queue then it may decide to deliver the pending messages or discard them. If the MPI implementation does not have access to the receive queue (e.g., it is implemented in hardware) then it may not be able to determine if there are pending messages from the newly failed process or not, or whether or not the hardware automatically discarded the messages. (*End of rationale.*)

> *Advice to users.* There is a natural race condition when using the MPI_ANY_SOURCE wildcard in a scenario involving process failures. Consider the scenario in which one process in the communicator sends a message while a different process fails. The result from the receive operation will be determined by the order in which the message arrives at the receiving process and the receiving process becomes aware of the failed process. (*End of advice to users.*)

> *Advice to implementors.* Manager/worker style applications may issue a receive using the MPI_ANY_SOURCE wildcard in the manager process to progress the computation. It may be desired that when a process failure occurs the MPI implementation should deliver any messages pending from active processes before returning the MPI_ERR_RANK_FAIL_STOP error code. This allows the manager process to continue making progress until it must deal with the failed process(es). (*End of advice to implementors.*)

MPI_COMM_REENABLE_ANY_SOURCE(comm, failed)

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

```
int MPI_Comm_reenable_any_source(MPI_Comm comm, MPI_Group *failed)
```

```
MPI_COMM_REENABLE_ANY_SOURCE(COMM, FAILED, IERROR)
    INTEGER COMM, FAILED, IERROR
```

The MPI_COMM_REENABLE_ANY_SOURCE function re-enables wildcard receives on the communicator comm, and returns the group failed containing processes known as failed at the time wildcard receives were re-enabled. Wildcard receives will again be disabled, if any processes in the communicator are detected as failed after the most recent call to MPI_COMM_REENABLE_ANY_SOURCE.

Advice to users. Care must be taken when using MPI_COMM_REENABLE_ANY_SOURCE with multiple threads to avoid race conditions that may result in hung processes. For example, consider two threads running the following loop to receive and process messages from client processes.

```
while(!done) {
    MPI_Comm_reenable_any_source(comm, &failed_group);
    /* check that at least one client process is alive */
    if (ok_to_continue(failed_group) == FALSE)
        break;
    /* receive and process messages until something goes wrong */
    while(!done) {
        ret = MPI_Recv(..., MPI_ANY_SOURCE, ..., comm, ...);
        if (ret == MPI_ERR_RANK_FAIL_STOP)
            /* Something failed, go back and check if we can continue */
            break;
        /* process the received message */
    }
}
```

It is possible that just before one thread calls MPI_Recv all of the client processes fail and the other thread calls MPI_COMM_REENABLE_ANY_SOURCE. The first thread will be stuck in MPI_Recv waiting for a message that will never arrive. See Example 17.5 for a thread safe solution using reader-writer locks.

(*End of advice to users.*)

17.6.1 Nonblocking and Persistent Communication

If the referenced process in a nonblocking or persistent communication operation is locally known to be a failed process at the creation or start call then those operations will **not** return an error class indicative of this failure. Instead the error will be returned during the completion call.

The MPI_COMM_REENABLE_ANY_SOURCE operation may be used to reenable the wildcard on the associated communicator for created, inactive persistent requests using the MPI_ANY_SOURCE wildcard.

Advice to users. Section 3.7 provides the guiding semantic for return values from creation and start calls for nonblocking and persistent communication operations. (*End of advice to users.*)

17.6.2 Send-Receive

If one or both of the ranks fail during either MPI_SENDRECV or MPI_SENDRECV_REPLACE the function will return MPI_ERR_IN_STATUS. If one rank failed then this rank will be identified in the status. If both ranks fail then only one of the ranks will be identified in the status. The query functions defined in Section 17.4 can be used to determine the state of the other rank. If an error handler function is registered to the communicator then it will be called only once for the operation regardless of the number of failed ranks.

17.6.3 Examples

Example 17.5 Re-enabling wildcard receives in a thread-safe manner using reader-writer locks.

```
int recognize_cnt = 0;  /* global */
MPI_Group failed_group; /* global */

int my_cnt = recognize_cnt - 1; /* local to thread or block. */
                                /* - 1 to force a check in first loop */
writer_lock();
MPI_Comm_group_failed(comm, &failed_group);
writer_unlock();

while(!done) {
    reader_lock();
    if (my_cnt != recognize_cnt) {
        /* New failures were detected */
        /* check failed_group and decide if ok to continue */
        if (ok_to_continue(failed_group) == FALSE) {
            reader_unlock();
            break;
        }
        my_cnt == recognize_cnt;
    }
    err = MPI_Recv(..., MPI_ANY_SOURCE, ..., comm, ...);
    if (err == MPI_ERR_FAILSTOP) {
        /* Failure case */
        reader_unlock();
        writer_lock();
        if (my_cnt != recognize_cnt) {
            /* another thread has already re-enabled wildcards */
            writer_unlock();
            continue;
        }
        MPI_Comm_reenable_any_source(comm, &failed_group);
        ++recognize_cnt;
        writer_unlock();
        continue;
    }
    /* Process the received message */
}
```

17.7 Collective Communication

Collective operations will eventually complete and return either success or some error to each alive process and will not hang indefinitely in the presence of process failure. Depending on how the collective operation is implemented and when a failure occurs some processes may return an error while others return success from the same collective operation. For example, some collectives allow processes to leave early, those that leave early may return success while others may return an error if the failure is detected later in the collective operation. An application must be aware that this situation may arise and plan appropriately.

Rationale. One option considered was to change the MPI collective semantics to disallow leave early semantics and implement an agreement algorithm at the end of every collective operation. This would allow all ranks to receive consistent return values. Due to the considerable overhead implications of this option, it was decided to allow for the looser consistency model to minimize the performance impact of the fault tolerance code path, and to provide the agreement protocol as a separate operation (e.g., MPI_COMM_VALIDATE). (*End of rationale.*)

Collective operations require a collectively active communicator. As such, all failed processes must be collectively recognized using a collective validate operation (e.g., MPI_COMM_VALIDATE) described in Section 17.7.4. If a collectively inactive communicator is used in a collective operation (other than MPI_COMM_VALIDATE and MPI_ICOMM_VALIDATE) the operation will complete and return an error code of the class MPI_ERR_RANK_FAIL_STOP. The communicator becomes collectively inactive when a process in the communicator fails.

Rationale. Calling a function to collectively validate a communicator gives the MPI implementation an opportunity to restructure collective communication patterns before the communicator is used by the alive process. Without this requirement the MPI implementation may need to determine which processes in the communicator are alive and which are failed for every collective operation. This results in performance restrictive semantics for every collective call. The collective validate operation allows the MPI library to trust the agreed upon set of communication patterns for the collectives reducing the impact of the fault tolerance logic on failure-free collective performance. (*End of rationale.*)

Advice to implementors. Some implementations may choose to offer the option of uniformly returning collective operations. (*End of advice to implementors.*)

If a collective operation completes with an error, the contents of any OUT or INOUT buffers are **undefined**. In particular, if the MPI_IN_PLACE option is used then the state of the buffer is undefined under process failure conditions.

Collective communication operations performed over a collectively active communicator with failed processes exclude the failed processes from the operation. For gather-type operations, where a process (or processes) receives data from every other process, the contents of the segment of the receive buffer corresponding to a failed process is **undefined**.

Rationale. The buffer contents are permitted to be undefined to allow the potential for optimized collective hardware to be used more efficiently and directly when processes have failed. (*End of rationale.*)

17.7.1 User-Defined Reduction Operations

The query operations described in Section 17.4 are local and therefore allowed to be called from within the user-defined reduction operation to assist the operation in identifying and working around recognized failed processes.

> *Advice to users.* The participation of recognized failed processes in the communicator associated with a reduction or scan operation are skipped by the MPI implementation. User-defined reduction operations should take this into account when writing reduction operations that are sensitive to missing contributions. (*End of advice to users.*)

17.7.2 Inclusive and Exclusive Scan Operations

The participation of recognized failed processes is skipped, and their contribution is ignored in the communicator associated with the MPI_SCAN and MPI_EXSCAN operations. In the MPI_EXSCAN operation when there are recognized failed processes in the communicator then references to process 0 are replaced with the first alive process in the communicator.

17.7.3 Nonblocking Collective Operations

As with nonblocking point-to-point operations (see Section 17.6.1), if the communicator is collectively inactive at the start call of the nonblocking collective operation then the operation will not return an error class indicative of this failure. Instead the error will be returned during the completion call.

It is erroneous to overlap collective communication with collective validation operations (e.g., MPI_COMM_VALIDATE).

17.7.4 Validating Communicators

MPI_COMM_VALIDATE(comm, failed)

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

```
int MPI_Comm_validate(MPI_Comm comm, MPI_Group *failed)
```

```
MPI_COMM_VALIDATE(COMM, FAILED, IERROR)
    INTEGER COMM, FAILED, IERROR
```

The MPI_COMM_VALIDATE function re-activates collectives in the communicator comm and returns a group of known failed processes in failed. The function must be called collectively by all alive processes in the communicator comm. MPI_COMM_VALIDATE will either provide the same group of failed processes in failed to every process or will return an error at every process. All collective communication operations initiated before the call to MPI_COMM_VALIDATE must also complete before it is called, and no collective calls may be initiated until it has completed.

MPI_ICOMM_VALIDATE(comm, failed, req)

IN	comm	communicator (handle)
OUT	failed	group of failed processes (handle)
OUT	req	request (handle)

```
int MPI_Icomm_validate(MPI_Comm comm, MPI_Group *failed, MPI_Request *req)
```

```
MPI_ICOMM_VALIDATE(COMM, FAILED, REQ, IERROR)
    INTEGER COMM, FAILED, REQ, IERROR
```

The MPI_ICOMM_VALIDATE function has the same semantics as MPI_COMM_VALIDATE except that it is nonblocking.

17.7.5 Examples

Barrier

The example below illustrates what a user can infer from the return code of MPI_BARRIER when a process failure is possible.

Example 17.6 Process Failure during a Barrier operation.

```
idx = 1;
MPI_Comm_size(comm, &comm_size);

/* Get a starting set of failures */
MPI_Comm_validate(comm, &failed_grp[0]);

do {
  ret = MPI_Barrier(comm);
  if( ret == MPI_ERR_RANK_FAIL_STOP ) {
    printf("Some rank failed during barrier\ n");
    /* Do not know if everyone returned success or error.
     * Failure could have occurred:
     *  - Before the sync, all would return error.
     *  - After the sync during distribution,
     *    some will receive error and others success.
     */
  }

  /* All processes are guaranteed to return everywhere:
   * - Either success or failure, and
   * - Same values for 'newfailures' below
   */
  MPI_Comm_validate(comm, &failed_grp[idx]);
  MPI_Comm_compare( failed_grp[(idx+1)%2], failed_grp[idx], &ret);
  idx = (idx+1)%2;
  MPI_Group_free(&failed_grp[idx]);
  /* Handle any -new- failures */
```

```
  if( ret != MPI_IDENT ) {
    printf("Some process failed, trying again.\ n");
  }
  /* Otherwise, no new failures - Barrier successful */
  else {
    break;
  }
} while(1);
```

Bcast

In the presence of process failure, depending upon when the failure occurs and how the MPI_BCAST operation is implemented, different ranks will see different return values from various iterations of MPI_BCAST. Eventually all ranks will enter the collective validate operation (i.e., MPI_COMM_VALIDATE) either thinking that all of the MPI_BCAST operations succeeded or that at least one had failed. A recovery block [5] is defined around the inner loop, so that if a failure is detected only the inner loop set of MPI_BCAST operations need to be re-executed.

Example 17.7 Process Failure during a Bcast operation (using recovery blocks).

```
idx = 1;
MPI_Comm_size(comm, &comm_size);

/* Get a starting set of failures */
MPI_Comm_validate(comm, &failed_grp[0]);

for(offset = 0; offset < 10; ++offset) {
  for(i = 0; i < comm_size; ++i ) {
    buffer = offset + i;
    ret = MPI_Bcast(&buffer, 1, MPI_INT, 0, comm);
    if( ret == MPI_ERR_RANK_FAIL_STOP ) {
      printf("Some rank failed during broadcast.\ n");
      break;
    }
  }

  MPI_Comm_validate(comm, &failed_grp[idx]);
  MPI_Comm_compare( failed_grp[(idx+1)%2], failed_grp[idx], &ret);
  idx = (idx+1)%2;
  MPI_Group_free(&failed_grp[idx]);
  /* Handle any -new- failures and continue */
  if( ret != MPI_IDENT ) {
    printf("Some process failed, trying again.\ n");
    offset--; /* Redo the last set of broadcasts */
  }
}
```

Exscan

The below example illustrates MPI_EXSCAN ignoring the contribution of recognized failed processes in the communicator. The logic to check for first non-failed rank is omitted for brevity, but the output from this rank should not be printed since the receive buffer is undefined.

Example 17.8 Process Failure during a Exscan operation.

```
idx = 1;
MPI_Comm_rank(comm, &comm_rank);
MPI_Comm_size(comm, &comm_size); /* Assume size = 5 */

send_buffer = comm_rank + 1;

/* Get a starting set of failures */
MPI_Comm_validate(comm, &failed_grp[0]);

do {
  ret = MPI_Exscan(send_buffer, recv_buffer, 1, MPI_INT, MPI_SUM, comm);
  if( ret == MPI_ERR_RANK_FAIL_STOP ) {
      printf("Some rank failed during scan\ n");
  }

  MPI_Comm_validate(comm, &failed_grp[idx]);
  MPI_Comm_compare( failed_grp[(idx+1)%2], failed_grp[idx], &ret);
  idx = (idx+1)%2;
  MPI_Group_free(&failed_grp[idx]);
  /* Handle any -new- failures and continue */
  if( ret != MPI_IDENT ) {
    continue;
  } else {
    break;
  }
} while(1);

printf("Rank %d) Received %2d\ n", comm_rank, recv_buffer);
/* Rank 0 has undefined receive buffer
 * Displays:
 * Rank 1) Received  1
 * Rank 2) Received  3
 * Rank 3) Received  6
 * Rank 4) Received 10
 */

/************* Rank 2 fails **************/

do {
  ret = MPI_Exscan(send_buffer, recv_buffer, 1, MPI_INT, MPI_SUM, comm);
```

```
    if( ret == MPI_ERR_RANK_FAIL_STOP ) {
        printf("Some rank failed during scan\ n");
    }

    MPI_Comm_validate(comm, &failed_grp[idx]);
    MPI_Comm_compare( failed_grp[(idx+1)%2], failed_grp[idx], &ret);
    idx = (idx+1)%2;
    MPI_Group_free(&failed_grp[idx]);
    /* Handle any -new- failures and continue */
    if( ret != MPI_IDENT ) {
      continue;
    } else {
      break;
    }
  } while(1);

  printf("Rank %d) Received %2d\ n", comm_rank, recv_buffer);
  /* Rank 0 has undefined receive buffer
   * Displays:
   * Rank 1) Received  1
   * Rank 3) Received  3
   * Rank 4) Received  7
   */

  /************* Rank 0 fails *************/

  do {
    ret = MPI_Exscan(send_buffer, recv_buffer, 1, MPI_INT, MPI_SUM, comm);
    if( ret == MPI_ERR_RANK_FAIL_STOP ) {
        printf("Some rank failed during scan\ n");
    }

    MPI_Comm_validate(comm, &failed_grp[idx]);
    MPI_Comm_compare( failed_grp[(idx+1)%2], failed_grp[idx], &ret);
    idx = (idx+1)%2;
    MPI_Group_free(&failed_grp[idx]);
    /* Handle any -new- failures and continue */
    if( ret != MPI_IDENT ) {
      continue;
    } else {
      break;
    }
  } while(1);

  printf("Rank %d) Received %2d\ n", comm_rank, recv_buffer);
  /* Rank 1 has undefined receive buffer
   * Displays:
   * Rank 3) Received  2
```

```
 * Rank 4) Received  6
 */
```

17.8 Group, Contexts, Communicators, and Caching

This section describes additional semantic clarifications for Chapter 6 regarding the effect of process failure on groups, contexts, communicators, and caching.

17.8.1 Group Management

MPI_GROUP_SIZE will return the number of processes, regardless of state, in the group.

If a failed process is represented in a group passed to a group constructor (e.g., MPI_GROUP_UNION) then the failed process is represented in the new group.

Groups including failed processes are allowed to be passed to the group destructor operation, MPI_GROUP_FREE. The group destructor operation will complete even in the presence of additional process failures not inclusive of the calling process.

17.8.2 Communicator Management

MPI_COMM_SIZE will return the number of processes in the local group, regardless of state, in the communicator.

All participating communicator(s) must be collectively active before calling any communicator creation operation. Otherwise, the communicator creation operation will uniformly return an error code of the class MPI_ERR_RANK_FAIL_STOP.

In the presence of process failures, the communicator construction operations must ensure that the communicator is either created successfully at all participating processes; or not created, and all participating processes return some error.

Advice to implementors. The uniform creation of the communicator handle semantic constraint is similar to the constraint on MPI_COMM_VALIDATE. In fact, an implementation can wrap existing communicator creation functions in a recovery block loop bound by MPI_COMM_VALIDATE operations to achieve the necessary semantic constraint. However, high quality implementations should be able to combine these operations to improve communicator creation performance in the presence of process failure. (*End of advice to implementors.*)

If a recognized failed process is represented in a communicator passed to the communicator constructor operation then it is represented in the new communicator as a recognized failure, except in the case of MPI_COMM_SPLIT. In the MPI_COMM_SPLIT operation, recognized failed processes in the associated communicator effectively supply the color MPI_UNDEFINED. If all other participating processes specify the same valid color then the newcomm will be a communicator that contains all active processes at the time of the communicator creation.

Rationale. This semantic of MPI_COMM_SPLIT allows libraries to easily create a new communicator of alive ranks. (*End of rationale.*)

Collectively inactive communicators are allowed to be passed to the communicator destructor operation, MPI_COMM_FREE. The communicator destructor operation will complete even in the presence of additional process failures not inclusive of the calling process.

MPI_COMM_COLLECTIVES_ENABLED(comm, active)

IN	comm	comm (handle)
OUT	active	true if the communicator is collectively active (logical)

```
int MPI_Comm_collectives_enabled(MPI_Comm comm, int *active)
```

```
MPI_COMM_COLLECTIVES_ENABLED(COMM, ACTIVE, IERROR)
    LOGICAL ACTIVE
    INTEGER COMM, IERROR
```

MPI_COMM_COLLECTIVES_ENABLED is a local operation that returns a logical value (active) indicating if the communicator is currently collectively active or not.

17.8.3 Inter-Communication

MPI_COMM_REMOTE_SIZE will return the number of processes in the remote group, regardless of state, in the communicator.

All participating communicator(s) must be collectively active before calling any inter-communicator construction operation. Otherwise, the inter-communicator creation operation will return an error code of the class MPI_ERR_RANK_FAIL_STOP. If a recognized failed process is represented in a communicator passed to the inter-communicator constructor operation then it is represented in the new inter-communicator as a recognized failure.

In the presence of process failures, the inter-communicator construction operations must ensure that the inter-communicator is either created successfully at all participating processes; or not created, and all participating processes return some error.

MPI_COMM_VALIDATE and MPI_ICOMM_VALIDATE can be used with both intra-communicators and inter-communicators. Using MPI_COMM_VALIDATE and MPI_ICOMM_VALIDATE over an inter-communicator will collectively re-enable collectives on the inter-communicator.

17.9 Process Topologies

All participating communicator(s) must be collectively active before calling any topology creation operation (i.e., MPI_GRAPH_CREATE, MPI_CART_CREATE, MPI_DIST_GRAPH_CREATE_ADJACENT, MPI_DIST_GRAPH_CREATE, and MPI_CART_SUB). Otherwise, the topology creation operation will return an error code of the class MPI_ERR_RANK_FAIL_STOP. If a recognized failed process is represented in a communicator passed to the topology constructor operation then it is represented in the new communicator as a recognized failure.

In the presence of process failures, the topology creation operations must ensure that the communicator is either created successfully at all participating processes; or not created, and all participating processes return some error.

For MPI_GRAPH_CREATE and MPI_CART_CREATE recognized failed processes are assumed to contribute identical values to their peers in the group defined by comm_old.

For MPI_DIST_GRAPH_CREATE_ADJACENT and MPI_DIST_GRAPH_CREATE recognized failed processes do not contribute to the construction of the input communication graph, and are assumed to contribute identical values for reorder and the info argument as their peers in the group of the associated communicator.

For MPI_CART_SUB recognized failed processes do not contribute to the subgrid topology construction operation.

17.10 Process Creation and Management

All participating communicator(s) must be collectively active before calling any spawn operation. Otherwise, the spawn operation will return an error code of the class MPI_ERR_RANK_FAIL_STOP. If a recognized failed process is represented in the parent communicator passed to the spawn operation then it is represented in the parent communicator returned to the children from MPI_COMM_GET_PARENT as a recognized failure.

> *Advice to users.* Note that the MPI_ERR_RANK_FAIL_STOP error case mentioned above is a different scenario than MPI_ERR_SPAWN error case which is raised when a child process fails to start. The raising of an error of the class MPI_ERR_RANK_FAIL_STOP indicates that some parent process in the communicator is a globally unrecognized failed process. (*End of advice to users.*)

In the presence of process failures, the spawn operations must ensure that either the children are started, and the associated inter-communicator is created successfully everywhere; or no children are connected to the parents, the inter-communicator is not created, and all participating parent processes return some error.

Recognized failed processes in the parent communicator comm do not participate in the spawn collective operation. Setting the root argument in a spawn operation to the rank of a failed process will raise an error of the class MPI_ERR_RANK.

17.10.1 Establishing Communication

All participating communicator(s) must be collectively active before calling MPI_COMM_ACCEPT or MPI_COMM_CONNECT. Otherwise, the accept and connect operations will return an error code of the class MPI_ERR_RANK_FAIL_STOP. If a recognized failed process is represented in the server or client communicator comm then it is represented in the resulting inter-communicator as a recognized failure.

In the presence of process failures, the accept and connect operations must ensure that the resulting inter-communicator is created successfully everywhere; or the inter-communicator is not created, and all participating processes return some error.

Setting the root argument in the accept and connect operations to the rank of a failed process will raise an error of the class MPI_ERR_RANK.

MPI_COMM_DISCONNECT will complete normally even in the presence of process failures, regardless of when the process failure occurs or if the process failure is recognized.

In the case of an error returned from MPI_COMM_JOIN, the state of the associated socket file descriptor (fd) is undefined.

17.11 One-Sided Communication

All participating communicator(s) must be collectively active before calling the MPI_WIN_CREATE operation. Otherwise, the MPI_WIN_CREATE operation will return an error code of the class MPI_ERR_RANK_FAIL_STOP. If a recognized failed process is represented in a communicator passed to the MPI_WIN_CREATE operation then it is represented in the group associated with the created window.

In the presence of process failures, the MPI_WIN_CREATE operation must ensure that the window is either created successfully at all participating processes; or not created, and all participating processes return some error.

MPI_WIN_FREE will complete normally even in the presence of process failures, regardless of when the process failure occurs.

One-sided communication (e.g., MPI_PUT, MPI_GET) with failed processes will return MPI_ERR_RANK_FAIL_STOP. If the process failure is unknown at the time of the call then the error may be delayed until a subsequent operation with this target, or the next synchronization call in the same epoch on this window. If an error is returned from MPI_GET and MPI_ACCUMULATE then the state of the buffer at the origin_addr is undefined.

Communication with active processes will proceed as normal even if there are failures in the group associated with the epoch on the window.

17.11.1 Validating Windows

Rationale. Since the communicator associated with the window cannot be accessed after window creation and since groups cannot be used for communication it is necessary to defined a validation operation specific to windows in addition to communicators (see Section 17.7.4). (*End of rationale.*)

MPI_WIN_VALIDATE(win, failed)

IN	win	window object (handle)
OUT	failed	group of failed processes (handle)

```
int MPI_Win_validate(MPI_Win win, MPI_Group *failed)
```

```
MPI_WIN_VALIDATE(WIN, FAILED, IERROR)
    INTEGER WIN, FAILED, IERROR
```

The MPI_WIN_VALIDATE function returns a group, failed, of globally known failed processes in the group associated with the window. The function must be called collectively by all alive processes in the window win. MPI_WIN_VALIDATE will either provide the same group of failed processes in failed to every process or will return an error at every process.

MPI_IWIN_VALIDATE(win, failed, req)

IN	win	window object (handle)
OUT	failed	group of failed processes (handle)
OUT	req	request (handle)

```
int MPI_Iwin_validate(MPI_Win win, MPI_Group *failed, MPI_Request *req)
```

```
MPI_IWIN_VALIDATE(WIN, FAILED, REQ, IERROR)
    INTEGER WIN, FAILED, REQ, IERROR
```

The MPI_IWIN_VALIDATE function has the same semantics as MPI_WIN_VALIDATE except that it is nonblocking.

17.11.2 Synchronization Calls

Advice to users. There are no requests or status objects used in the one-sided communication operations. As such, it is difficult to identify which operations failed from the synchronization operation. In the case of process failure, upon completion of the epoch an error will be returned to indicate that a process failed during the epoch. Other synchronization and query functions defined in Section 17.4 can be used to determine which process(es) are failed. (*End of advice to users.*)

The MPI_WIN_FENCE operation will proceed normally (completing or starting epochs and synchronizing RMA operations on the window, as defined in Section 11.4) in the presence of failed processes in the group associated with the window. If an unrecognized failed processes exists in the group associated with the window at the time of the call to MPI_WIN_FENCE then the operation will return an error in the class of MPI_ERR_RANK_FAIL_STOP.

Advice to implementors. This means that MPI_WIN_FENCE must be able to work around process failures that emerge during the synchronization operation to complete the epoch. It does not require that all alive calling processes are returned the same error code. But it does require that the one-sided operations are completed between all alive, communicating peer sets, and that the epoch is started or completed as normal. (*End of advice to implementors.*)

Recognized failed processes are excluded from the synchronization in the MPI_WIN_FENCE, MPI_WIN_START, MPI_WIN_COMPLETE, MPI_WIN_POST, and MPI_WIN_WAIT operations.

A call to MPI_WIN_COMPLETE will return an error in the class of MPI_ERR_RANK_FAIL_STOP if any process fails between the MPI_WIN_START and the subsequent call to MPI_WIN_COMPLETE. The epoch is completed as normal on the window. If the origin process did not communicate with the failed processes during the epoch, then MPI_WIN_COMPLETE may return success.

Advice to users. It is possible that some participating processes in the synchronization will see an error while others see success. The user should be aware of this situation, and use other synchronization operations, such as a collective validate operation (e.g., MPI_WIN_VALIDATE described in Section 17.11.1), to ensure that all processes in the group completed the operation. (*End of advice to users.*)

A call to MPI_WIN_WAIT will return an error in the class of MPI_ERR_RANK_FAIL_STOP if any process fails between the MPI_WIN_POST and the subsequent call to MPI_WIN_WAIT. The epoch is completed as normal on the window. If an error is returned by MPI_WIN_WAIT then the state of the target window memory, if accessed by any of the failed processes, is undefined.

Advice to users. Since it is possible for an implementation of MPI_WIN_COMPLETE to finish before MPI_WIN_POST, some processes may leave the epoch synchronization successfully while others return an error. The epoch is guaranteed to be finished, but the state of the target window memory is undefined. Other synchronization operations, such as a collective validate operation (e.g., MPI_WIN_VALIDATE described in Section 17.11.1), can be used to ensure that all processes in the group completed the operation. (*End of advice to users.*)

A call to MPI_WIN_LOCK will return an error in the class MPI_ERR_RANK_FAIL_STOP if the target rank is locally known to be failed. A call to MPI_WIN_UNLOCK will return an error in the class MPI_ERR_RANK_FAIL_STOP if the target rank is locally known to be failed. The associated epoch will be marked as completed.

17.12 I/O

Advice to users. The state of the external file must be determined by the application (e.g., Did a failed process finish writing/reading/syncing before failing?). The application may be able to use the MPI_FILE_READ_AT operation to determine the state of the file. The collective validate operations (e.g., MPI_FILE_VALIDATE described in Section 17.12.1) help to ensure buffers are fully flushed to disk. (*End of advice to users.*)

17.12.1 Validating File Handles

Rationale. Since the communicator associated with the file handle cannot be accessed after creation and since groups cannot be used for communication it is necessary to defined a validation operation specific to file handles in addition to communicators (see Section 17.7.4). (*End of rationale.*)

MPI_FILE_VALIDATE(fh, failed)

IN	fh	file handle (handle)
OUT	failed	group of failed processes (handle)

```
int MPI_File_validate(MPI_File fh, MPI_Group *failed)
```

```
MPI_FILE_VALIDATE(FH, FAILED, IERROR)
    INTEGER FH, FAILED, IERROR
```

The MPI_FILE_VALIDATE function re-activates collectives in the file handle fh and returns a group of globally known failed processes failed in the group associated with the file handle. The function must be called collectively by all alive processes in the file handle fh. MPI_FILE_VALIDATE will either provide the same group of failed processes in failed to every process or will return an error at every process. All collective communication operations initiated before the call to MPI_FILE_VALIDATE must also complete before it is called, and no collective calls may be initiated until it has completed.

MPI_IFILE_VALIDATE(fh, failed, req)

IN	fh	file handle (handle)
OUT	failed	group of failed processes (handle)
OUT	req	request (handle)

```
int MPI_Ifile_validate(MPI_File fh, MPI_Group *failed, MPI_Request *req)
```

```
MPI_IFILE_VALIDATE(FH, FAILED, REQ, IERROR)
    INTEGER FH, FAILED, REQ, IERROR
```

The MPI_IFILE_VALIDATE function has the same semantics as MPI_FILE_VALIDATE except that it is nonblocking.

MPI_FILE_COLLECTIVES_ENABLED(fh, active)

IN	fh	file handle (handle)
OUT	active	true if the file handle is collectively active (logical)

```
int MPI_File_collectives_enabled(MPI_File fh, int *active)
```

```
MPI_FILE_COLLECTIVES_ENABLED(FH, ACTIVE, IERROR)
    LOGICAL ACTIVE
    INTEGER FH, IERROR
```

MPI_FILE_COLLECTIVES_ENABLED is a local operation that returns a logical value (active) indicating if the file handle is currently collectively active or not.

17.12.2 File Manipulation

All participating communicator(s) must be collectively active before calling the MPI_FILE_OPEN operation. All failed processes must be collectively recognized using the collective validate operation (i.e., MPI_FILE_VALIDATE) on the associated file handle before calling MPI_FILE_CLOSE operation. Otherwise, the MPI_FILE_OPEN and MPI_FILE_CLOSE operations will return an error code of the class MPI_ERR_RANK_FAIL_STOP. If a recognized failed process is represented in a communicator passed to the MPI_FILE_OPEN operation then it is represented in the group associated with the created file handle as a recognized failure. If MPI_FILE_CLOSE returns an error code of the class MPI_ERR_RANK_FAIL_STOP the file will **not** be closed.

Rationale. If a new process failure emerges before the file is closed, the application may want to adjust what each process wrote to the file before attempting to close it again. If the close operation is made to work around process failures (as with similar operations like MPI_COMM_FREE and MPI_WIN_FREE), then it is difficult for the application to determine if the close operation was successful at any newly failed process (e.g., did the process fail before or after returning from MPI_FILE_CLOSE?). (*End of rationale.*)

In the presence of process failures, the MPI_FILE_OPEN operation must ensure that the file handle is either created successfully at all participating processes; or not created, and all participating processes return some error.

Advice to users. Opening a file with recognized failed processes may be useful for an application to dump state before terminating the application. (*End of advice to users.*)

Advice to implementors. The info argument to the MPI_FILE_OPEN operation may be used to modify the fault tolerance semantics of the operation. For example, an implementation may provide an info key to only create the file handle on all alive processes in the communicator, and reduce the associated group by the number of failures. (*End of advice to implementors.*)

The file handle must be collectively active before calling the MPI_FILE_SET_SIZE, MPI_FILE_PREALLOCATE, and MPI_FILE_SET_INFO operations. Otherwise, these operations will return an error code of the class MPI_ERR_RANK_FAIL_STOP. The one exception to this is when the info argument to MPI_FILE_SET_INFO does not require global uniformity. In that case, it is valid for the implementation to return success at all alive processes, even if there are unrecognized failed processes.

Depending on how the MPI_FILE_SET_SIZE, MPI_FILE_PREALLOCATE, and MPI_FILE_SET_INFO collective operations are implemented and when a process failure occurs some alive processes may return an error while others return success.

Reserved File Hints

Advice to implementors. Some info keys must become fault tolerant to consistently provide the specified functionality. For example, collective_buffering may require redundant buffering to handle the loss of one or more target nodes. At the point the implementation cannot provide the required behavior subsequent operations on the file handle should return an appropriate error code. (*End of advice to implementors.*)

17.12.3 File Views

The file handle must be collectively active before calling the MPI_FILE_SET_VIEW operation. Otherwise, the operation will return an error code of the class MPI_ERR_RANK_FAIL_STOP. Depending on how this collective is implemented and when a failure occurs some processes may return an error, while others return success.

Recognized failed processes participate in a passive manner in the MPI_FILE_SET_VIEW operation. Such processes effectively pass identical parameters for those that need to be identical on all processes, and provide values for disp, filetype and info that do not perturb active processes in the operation.

17.12.4 Data Access

In collective operations on file handles, recognized failed processes do not read or write in the file operation. Recognized failed processes **do** affect the shared file pointer as defined by the MPI_FILE_SET_VIEW operation, and should be processed in order during any synchronization needed by the collective operation.

The file handle must be collectively active before calling the MPI_FILE_READ_AT_ALL, MPI_FILE_WRITE_AT_ALL, MPI_FILE_READ_ALL, MPI_FILE_WRITE_ALL, MPI_FILE_READ_ORDERED, MPI_FILE_WRITE_ORDERED, and MPI_FILE_SEEK_SHARED operations. Otherwise, these operations will return an error code of the class MPI_ERR_RANK_FAIL_STOP. Depending on how these collectives are implemented and when a failure occurs some processes may return an error, while others return success.

> *Advice to implementors.* Some implementations may choose to offer the option of a uniformly returning version of the MPI_FILE_SEEK_SHARED operation that is able to work around emerging process failures to provide a consistent view of the shared file pointer. However, the implementation is not required to do so. (*End of advice to implementors.*)

As with nonblocking point-to-point (see Section 17.6.1) and collective (see Section 17.7.3) operations, if the file handle is collectively inactive at the start call of the split collective operation then the start operation will **not** return an error class indicative of this failure. Instead the error will be returned during the end call.

The end call will return an error code of the class MPI_ERR_RANK_FAIL_STOP if there are unrecognized failed processes in the group associated with the file handle. Depending on how these split collectives are implemented and when the failure occurs some processes may return an error, while other return success. Recognized failed processes participate as they do in the blocking collective variations of these operations.

17.12.5 Consistency and Semantics

The file handle must be collectively active before calling the MPI_FILE_SET_ATOMICITY, and MPI_FILE_SYNC operations. Otherwise, these operations will return an error code of the class MPI_ERR_RANK_FAIL_STOP. Depending on how these collectives are implemented and when a failure occurs some processes may return an error, while others return success. Recognized failed processes do not contribute to these operations.

Bibliography

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11 – 33, January-March 2004. 17.2

[2] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, March 1996. 17.3

[3] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35:288–323, April 1988. 17.3

[4] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32:374–382, April 1985. 17.3

[5] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on reliable software*, pages 437–449. ACM Press, 1975. 17.7.5

[6] Aleta M. Ricciardi and Kenneth P. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the tenth annual ACM symposium on Principles of Distributed Computing*, PODC '91, pages 341–353, New York, NY, USA, 1991. ACM. 17.3

[7] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1:222–238, August 1983. 17.2

Index