FT Working Group Ticket #276: Run-Through Stabilization Process Fault Tolerance

Joshua Hursey

Postdoctoral Research Associate
Oak Ridge National Laboratory

MPI Forum – Oct. 24, 2011

Fault Tolerance Working Group

Define a set of semantics and interfaces to enable fault tolerant applications and libraries to be portably constructed on top of MPI.

- **Application involved fault tolerance** (not transparent FT)
- **Starting with *fail-stop* process failure**
  - A process failure in which the process permanently stops executing, and its internal state is lost (e.g., due to a hardware component crash).
- **Two Complementary Proposals:**
  - **Run-Through Stabilization:** (*Target: MPI-3.0*)
    - Continue running and using MPI even if one or more MPI processes fail
  - **Process Recovery:** (*Target: MPI-3.1*)
    - Replace MPI processes in existing communicators, windows, file handles

Run-Through Stabilization Proposal

- **Error Handlers:**
  - **Application/Library must *opt-in* by:**
    - Replacing `MPI_ERRORS_ARE_FATAL` with at least `MPI_ERRORS_RETURN`
  - **MPI implementation may *opt-out* by:**
    - Returning `MPI_ERR_UNSUPPORTED_OPERATION` for new operations, and
    - Never returning the new error class `MPI_ERR_PROC_FAIL_STOP`
- **Error Class:** `MPI_ERR_PROC_FAIL_STOP`
  - A process in the operation is failed (fail-stop failure)
  - If this error class is returned then the MPI agrees to provide the specified semantics and interfaces defined by this proposal
- **The behavior of MPI after returning other error classes remains undefined by the standard.**

Failure detector exposed to the application

MPI will provide the ability to detect process failures. MPI will guarantee that eventually all alive processes will be able to know about the failure. The state management and query operations defined in this chapter 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.

- ***Eventually Perfect Detector***
  - Strong Completeness:
    Eventually every failed process will be known to all processes
  - Eventual Strong Accuracy:
    Processes reported as failed are eventually guaranteed to be failed

12 New MPI Operations

- **Communicator Object Permanence & Library Support**
  - Communication objects are not destroyed upon failure.
  - Management of failure is associated with the communication object.

```
Communicators:
   MPI_Comm_group_failed          (comm, group)        - Local
   MPI_Comm_group_remote_failed   (comm, group)        - Local
   MPI_Comm_reenable_any_source   (comm, group)        - Local
   MPI_Comm_any_source_enabled    (comm, result)       - Local
   MPI_Comm_validate              (comm, group)        - Collective
   MPI_Comm_ivalidate             (comm, group, req)   - Collective (nonblocking)

Windows:
   MPI_Win_group_failed           (win, group)         - Local
   MPI_Win_validate               (win, group)         - Collective
   MPI_Win_ivalidate              (win, group, req)    - Collective (nonblocking)

File Handles:
   MPI_File_group_failed          (fh, group)          - Local
   MPI_File_validate              (fh, group)          - Collective
   MPI_File_ivalidate             (fh, group, req)     - Collective (nonblocking)
```

Are there any failed processes (locally consistent)?

```
MPI_Group failed_grp;
int size;

MPI_Comm_group_failed( comm, &failed_grp );
/* if no failures failed_group = MPI_GROUP_EMPTY */

MPI_Group_size( failed_grp, &size );
if( size > 0 ) {
  /* There are failed processes */
}
```

Are there any failed processes (globally consistent)?

```
MPI_Group failed_grp;
int size;

MPI_Comm_validate ( comm, &failed_grp );
/* if no failures failed_group = MPI_GROUP_EMPTY */

MPI_Group_size( failed_grp, &size );
if( size > 0 ) {
  /* There are failed processes */
}
```

Are there any newly failed processes?

```
MPI_Group failed_grp[2], new_failed;
int result;

MPI_Comm_group_failed( comm, &failed_grp[0] );
...
MPI_Comm_group_failed( comm, &failed_grp[1] );

/* Where there any new failures? */
MPI_Group_compare( failed_grp[0], failed_grp[1], &result );
if( result != MPI_IDENT ) {
  MPI_Group_difference( failed_grp[0], failed_grp[1], &new_failed );
} else {
  /* No new failures */
}
```

Is rank X alive in MPI_COMM_WORLD?

```
MPI_Group fail_grp, mcw_grp;
int grp_rank, mcw_rank = 5;

MPI_Comm_group( MPI_COMM_WORLD, &mcw_grp );
MPI_Comm_group_failed( MPI_COMM_WORLD, &fail_grp );

MPI_Group_translate_ranks( mcw_grp, 1, &mcw_rank, fail_grp, &grp_rank );
if( grp_rank != MPI_UNDEFINED ) {
  /* Rank 5 has failed */
}
```

Quick Overview of Semantics

- **Point-to-Point Communication**
  - **Isolation of process failure:**
    Communication between active processes is unaffected by the failure of other non-participating process.
    - Proc. A can communicate with Proc. B, even if Proc. C has failed.
  - `MPI_ANY_SOURCE` is exposed to all process failures…

Point-to-Point Communication:
MPI_Recv(…,MPI_ANY_SOURCE,…)

- **Once a process fails:**
  - Posted MPI_Recv(MPI_ANY_SOURCE):
    - Complete with **error** MPI_ERR_PROC_FAIL_STOP
  - Posted MPI_Irecv(MPI_ANY_SOURCE, &req):
    - Complete with **warning** MPI_WARN_PROC_FAIL_STOP
  - Subsequent MPI_ANY_SOURCE receives are disabled.
  - User can acknowledge the set of failures and re-enable via:

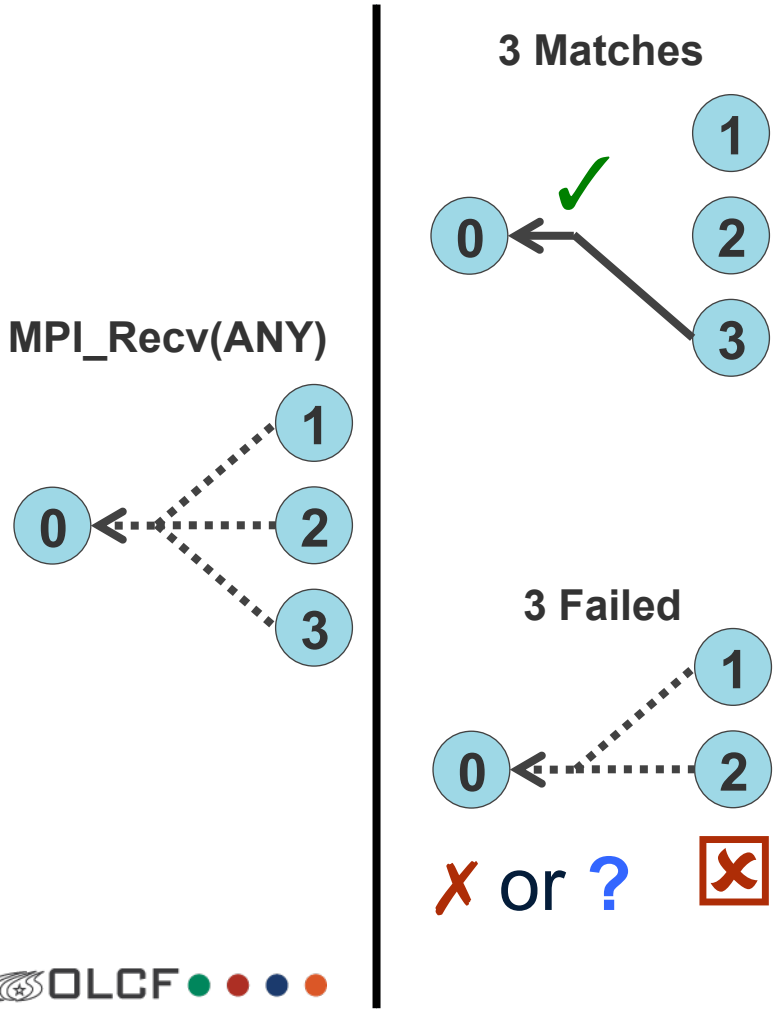
```
MPI_Comm_reenable_any_source(comm, group)
 IN  comm    communicator (handle)
 OUT group   group of failed processes (handle)

The 'group' returned is the set of locally known failures that are excluded from
matching in an operation using MPI_ANY_SOURCE.

MPI_Comm_any_source_enabled (comm, enabled)
 IN  comm     communicator (handle)
 OUT enabled  true iff the wildcard receives are enabled (logical)
```

Point-to-Point Communication:
MPI_Recv(…,MPI_ANY_SOURCE,…)

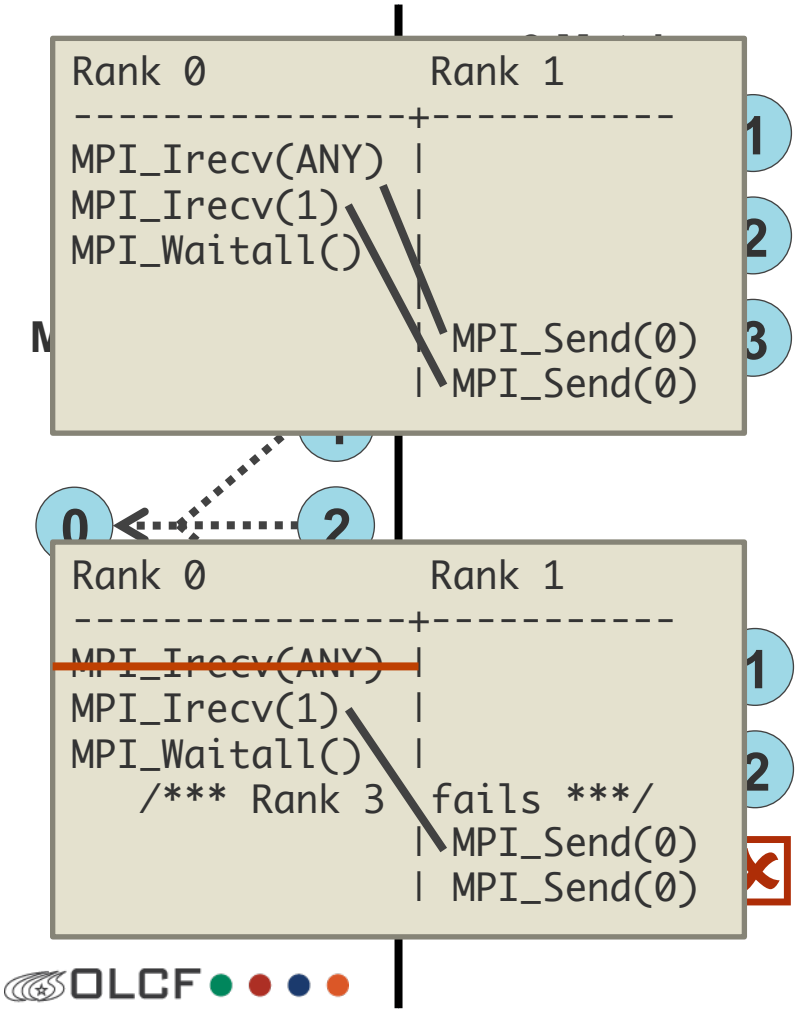
- **Is a new failure important to the completion of the MPI_Recv(MPI_ANY_SOURCE)?**



- **Option 1:**
  Do nothing, continue blocking
  - Largest user burden
- **Option 2:**
  Complete all such receives with error, option to re-enable
  - Ordering of matching subsequent receives is challenging to manage.
- **Option 3:**
  Provide the user a warning, but do not complete the message.
  - User is in control of the cancel/continue
  - Blocking receive must be canceled due to lack of a request handle.

Point-to-Point Communication:
MPI_Recv(…,MPI_ANY_SOURCE,…)

- **Is a new failure important to the completion of the MPI_Recv(MPI_ANY_SOURCE)?**



- **Option 1:**
  Do nothing, continue blocking
  - Largest user burden
- **Option 2:**
  Complete all such receives with error, option to re-enable
  - Ordering of matching subsequent receives is challenging to manage.
- **Option 3:**
  Provide the user a warning, but do not complete the message.
  - User is in control of the cancel/continue
  - Blocking receive must be canceled due to lack of a request handle.

MPI_COMM_REENABLE_ANY_SOURCE Example:

```
MPI_Group failed_group;
bool done = false;

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 ) {
    printf("Error: Cannot continue. Too many failures!\n");

    break;
  }

  /* receive and process messages until something goes wrong */
  while( !done ) {

    ret = MPI_Recv(&buf,1,MPI_INT, MPI_ANY_SOURCE, 42,comm,MPI_STATUS_IGNORE);


    if( ret == MPI_ERR_PROC_FAIL_STOP ) {
      /* Something failed, go back and check if we can continue */
      printf("Warning: Some rank failed. Attempting to recover.\n");
      break;
    }
    /* process the received message */
    check_if_done(&done);
  }
}
```

MPI_COMM_REENABLE_ANY_SOURCE Example:

```
MPI_Group failed_group; MPI_Request req;
bool done = false;

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 ) {
    printf("Error: Cannot continue. Too many failures!\n");
    MPI_Cancel(&req);
    break;
  }

  /* receive and process messages until something goes wrong */
  while( !done ) {
    if( req == MPI_REQUEST_NULL ) {
      MPI_Irecv(&buf,1,MPI_INT, MPI_ANY_SOURCE, 42,comm, &req);
    }
    ret = MPI_Wait(&req, MPI_STATUS_IGNORE);
    if( ret == MPI_WARN_PROC_FAIL_STOP ) {
      /* Something failed, go back and check if we can continue */
      printf("Warning: Some rank failed. Attempting to recover.\n");
      break;
    }
    /* process the received message */
    check_if_done(&done);
  }
}
```

Quick Overview of Semantics

- **Communication Object Creation:**
  - Uniformly consistent creation of the communicator.
  - Collectives need not be enabled in all participating communicators.
  - Failed processes may be propagated to new communicators.
- **Collective Communication**
  - Fault-aware: Will eventually terminate in the presence of failure.
  - Not required to provide uniform return codes in the presence of failure.
  - Collectives disabled once *any* process in the communicator fails.
    - Collectives re-enabled by calling **`MPI Comm validate()`**

```
MPI_Comm_validate(comm, group)
 IN  comm    communicator (handle)
 OUT group   group of failed processes (handle)

This collective operation provides a fault tolerant agreement protocol. This
operation agrees upon the 'group' of globally known process failures in the
specified communicator. Upon successful completion all processes are provided a
uniform return code, the same value for 'group', and collectives are re-enabled.
Subsequent collective operations exclude the group of agreed upon failures.
```

MPI_COMM_SPLIT Example:

```
MPI_Comm new_comm;
MPI_Group failed_grp;
int color, key = comm_rank;

if( comm_rank < 8 ) {
  color = 0;
} else {
  color = 1;
}

do {
  ret = MPI_Comm_split( comm, color, key, &new_comm );
  if( ret == MPI_ERR_PROC_FAIL_STOP ) {
    /* Re-enable collectives over this communicator */
    MPI_Comm_validate( comm, &failed_grp );
    MPI_Group_free( &failed_grp );
  }
  /* If there was a process failure, then try again */
} while( ret == MPI_ERR_PROC_FAIL_STOP );

// If comm_size=16, and Ranks 2,4,10 are failed then
// Ranks 0-7  see a communicator of size 6 since ranks 2 and 4 are excluded:
//   [ 0, 1, 3, 5, 6, 7]
// Ranks 8-15 see a communicator of size 7 since rank 10 is excluded:
//   [ 8, 9,11,12,13,14,15]
```

MPI_GATHER Example:

```
MPI_Group failed_grp[2];
int s_buff = comm_rank;
int r_buff[comm_size] = {0}
int idx = 1;

MPI_Comm_validate( comm, &failed_grp[0] ); /* Initial set */
do {
  ret = MPI_Gather( &s_buff, 1, MPI_INT, &r_buff, 1, MPI_INT, 0, comm );
  if( ret == MPI_ERR_PROC_FAIL_STOP ) {
    /* I know that the error is local, so the contents of r_buff are undefined.
     * I do not know if every other process in the communicator has been
     * returned an error. So do not take divergent action here. Instead check
     * using MPI_Comm_validate below.
     */
  }

  /* Where there any new failures across the call to MPI_Gather? */
  MPI_Comm_validate( comm, &failed_grp[idx] );
  MPI_Group_compare( failed_grp[(idx+1)%2], failed_grp[idx], &r );
  idx = (idx+1)%2;
  MPI_Group_free(&failed_grp[idx]);

} while( r != MPI_IDENT ); /* Try again upon failure */

// If size=5, and rank 3 fails:
//   r_buff @ Rank 0 is [0,1,2,?,4]
```

MPI_BCAST Example:

```
MPI_Group failed_grp[2];
int r, idx = 1;

MPI_Comm_validate( comm, &failed_grp[0] ); /* Initial set */

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_PROC_FAIL_STOP ) {
      /* Some rank failed during the broadcast.
       * Wait for all other processes at the MPI_Comm_validate check below.
       */
      break;
    }
  }

  /* Where there any new failures across the MPI_Bcast inner loop? */
  MPI_Comm_validate( comm, &failed_grp[idx] );
  MPI_Group_compare( failed_grp[(idx+1)%2], failed_grp[idx], &r );
  idx = (idx+1)%2;
  MPI_Group_free(&failed_grp[idx]);
  if( r != MPI_IDENT ) {
    /* Redo the last set of broadcasts */
    offset--;
  }
}
```

Collective Communication:
MPI_Comm_validate_multiple()

```
MPI_Comm_validate(parent_comm, group[num_comms])
for(i = 0; i < num_comms; ++i){
  MPI_Comm_validate(sub_comm[i], group[i]);
}
```

- **Validate multiple communicators in a single collective**
  - Performance optimization to combine multiple collective operations
  - Child communicators must be subset of parent communicator
    - parent_comm must be the same at all processes
    - sub_comms may be different at all processes
  - Group of failed processes per-communicator, globally consistent

```
MPI_Comm_validate_multiple(parent_comm, num_comms, sub_comms[], groups[])
 IN  parent_comm  parent communicator (handle)
 IN  num_comms    number of communicators passed in (integer)
 IN  sub_comms    communicators derived from parent_comm (array of handles)
 OUT groups       array of groups of failed processes (array of handles)

This collective operation validates multiple communicators in a single
collective operation.
```

Process Failure Handler (FailHandler)

- **Uniform notification of process failure in a communication object not associated with the current MPI call**
  - Called once per set of process failures
    - Additional process failures may trigger another callback after the call completes.
  - Only allowed to use local operations from within the callback
    - FailHandler callback cannot be triggered from within any FailHandler callback
- **Created with:** `MPI_XXX_CREATE_ERRHANDLER()`
  - Uses the same object: `MPI_Errhandler`
  - New functions for accessing and attaching a FailHandler
  - NULL handle: `MPI_FAILHANDLER_NULL`

```
MPI_Comm_set_failhandler( comm, errhandler)
MPI_Comm_get_failhandler( comm, errhandler)
MPI_Win_set_failhandler(   win, errhandler)
MPI_Win_get_failhandler(   win, errhandler)
MPI_File_set_failhandler(   fh, errhandler)
MPI_File_get_failhandler(   fh, errhandler)
```

Continue discussion from the draft proposal

- **Continue discussion from the draft proposal attached to Ticket #276**
  - https://svn.mpi-forum.org/trac/mpi-forum-web/ticket/276
- **Specifically:**
  - One-sided
  - I/O
  - Process creation and management