to MPI_COMM_FREE. The object is ultimately deallocated when the count reaches zero.

Though collective, it is anticipated that this operation will normally be implemented to be local, though a debugging version of an MPI library might choose to synchronize. (*End of advice to implementors.*)

## 6.5 Motivating Examples

### 6.5.1 Current Practice #1

Example #1a:

```
int main(int argc, char **argv)
{
  int me, size;
  ...
  MPI_Init ( &argc, &argv );
  MPI_Comm_rank (MPI_COMM_WORLD, &me);
  MPI_Comm_size (MPI_COMM_WORLD, &size);

  (void)printf ("Process %d size %d\n", me, size);
  ...
  MPI_Finalize();
}
```

Example #1a is a do-nothing program that initializes itself legally, and refers to the "all" communicator, and prints a message. It terminates itself legally too. This example does not imply that MPI supports `printf`-like communication itself.
Example #1b (supposing that `size` is even):

```
int main(int argc, char **argv)
{
   int me, size;
   int SOME_TAG = 0;
   ...
   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &me);   /* local */
   MPI_Comm_size(MPI_COMM_WORLD, &size); /* local */

   if((me % 2) == 0)
   {
      /* send unless highest-numbered process */
      if((me + 1) < size)
         MPI_Send(..., me + 1, SOME_TAG, MPI_COMM_WORLD);
   }
   else
      MPI_Recv(..., me - 1, SOME_TAG, MPI_COMM_WORLD, &status);
```

```
       ...
       MPI_Finalize();
   }
```

Example #1b schematically illustrates message exchanges between "even" and "odd" processes in the "all" communicator.

### 6.5.2 Current Practice #2

```
int main(int argc, char **argv)
{
  int me, count;
  void *data;
  ...

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);

  if(me == 0)
  {
      /* get input, create buffer ``data'' */
      ...
  }

  MPI_Bcast(data, count, MPI_BYTE, 0, MPI_COMM_WORLD);

  ...

  MPI_Finalize();
}
```

This example illustrates the use of a collective communication.

### 6.5.3 (Approximate) Current Practice #3

```
int main(int argc, char **argv)
{
  int me, count, count2;
  void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
  MPI_Group MPI_GROUP_WORLD, grprem;
  MPI_Comm commslave;
  static int ranks[] = {0};
  ...
  MPI_Init(&argc, &argv);
  MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);
  MPI_Comm_rank(MPI_COMM_WORLD, &me);  /* local */

  MPI_Group_excl(MPI_GROUP_WORLD, 1, ranks, &grprem);  /* local */
  MPI_Comm_create(MPI_COMM_WORLD, grprem, &commslave);
```

```
    if(me != 0)
    {
      /* compute on slave */
      ...
      MPI_Reduce(send_buf,recv_buff,count, MPI_INT, MPI_SUM, 1, commslave);
      ...
      MPI_Comm_free(&commslave);
    }
    /* zero falls through immediately to this reduce, others do later... */
    MPI_Reduce(send_buf2, recv_buff2, count2,
               MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Group_free(&MPI_GROUP_WORLD);
    MPI_Group_free(&grprem);
    MPI_Finalize();
  }
```

This example illustrates how a group consisting of all but the zeroth process of the "all" group is created, and then how a communicator is formed (commslave) for that new group. The new communicator is used in a collective call, and all processes execute a collective call in the MPI_COMM_WORLD context. This example illustrates how the two communicators (that inherently possess distinct contexts) protect communication. That is, communication in MPI_COMM_WORLD is insulated from communication in commslave, and vice versa.

In summary, "group safety" is achieved via communicators because distinct contexts within communicators are enforced to be unique on any process.

### 6.5.4 Example #4

The following example is meant to illustrate "safety" between point-to-point and collective communication. MPI guarantees that a single communicator can do safe point-to-point and collective communication.

```
   #define TAG_ARBITRARY 12345
   #define SOME_COUNT       50

   int main(int argc, char **argv)
   {
     int me;
     MPI_Request request[2];
     MPI_Status status[2];
     MPI_Group MPI_GROUP_WORLD, subgroup;
     int ranks[] = {2, 4, 6, 8};
     MPI_Comm the_comm;
     ...
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

     MPI_Group_incl(MPI_GROUP_WORLD, 4, ranks, &subgroup); /* local */
```

```
  MPI_Group_rank(subgroup, &me);     /* local */

  MPI_Comm_create(MPI_COMM_WORLD, subgroup, &the_comm);

  if(me != MPI_UNDEFINED)
  {
      MPI_Irecv(buff1, count, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_ARBITRARY,
                        the_comm, request);
      MPI_Isend(buff2, count, MPI_DOUBLE, (me+1)%4, TAG_ARBITRARY,
                        the_comm, request+1);
      for(i = 0; i < SOME_COUNT, i++)
        MPI_Reduce(..., the_comm);
      MPI_Waitall(2, request, status);

      MPI_Comm_free(&the_comm);
  }

  MPI_Group_free(&MPI_GROUP_WORLD);
  MPI_Group_free(&subgroup);
  MPI_Finalize();
}
```

### 6.5.5 Library Example #1

The main program:

```
int main(int argc, char **argv)
{
  int done = 0;
  user_lib_t *libh_a, *libh_b;
  void *dataset1, *dataset2;
  ...
  MPI_Init(&argc, &argv);
  ...
  init_user_lib(MPI_COMM_WORLD, &libh_a);
  init_user_lib(MPI_COMM_WORLD, &libh_b);
  ...
  user_start_op(libh_a, dataset1);
  user_start_op(libh_b, dataset2);
  ...
  while(!done)
  {
     /* work */
     ...
     MPI_Reduce(..., MPI_COMM_WORLD);
     ...
     /* see if done */
     ...
```

```
    }
    user_end_op(libh_a);
    user_end_op(libh_b);

    uninit_user_lib(libh_a);
    uninit_user_lib(libh_b);
    MPI_Finalize();
  }
```

The user library initialization code:

```
  void init_user_lib(MPI_Comm comm, user_lib_t **handle)
  {
    user_lib_t *save;

    user_lib_initsave(&save); /* local */
    MPI_Comm_dup(comm, &(save -> comm));

    /* other inits */
    ...

    *handle = save;
  }
```

User start-up code:

```
  void user_start_op(user_lib_t *handle, void *data)
  {
    MPI_Irecv( ..., handle->comm, &(handle -> irecv_handle) );
    MPI_Isend( ..., handle->comm, &(handle -> isend_handle) );
  }
```

User communication clean-up code:

```
  void user_end_op(user_lib_t *handle)
  {
    MPI_Status status;
    MPI_Wait(handle -> isend_handle, &status);
    MPI_Wait(handle -> irecv_handle, &status);
  }
```

User object clean-up code:

```
  void uninit_user_lib(user_lib_t *handle)
  {
    MPI_Comm_free(&(handle -> comm));
    free(handle);
  }
```

### 6.5.6 Library Example #2

The main program:

```
   int main(int argc, char **argv)
   {
     int ma, mb;
     MPI_Group MPI_GROUP_WORLD, group_a, group_b;
     MPI_Comm comm_a, comm_b;

     static int list_a[] = {0, 1};
#if  defined(EXAMPLE_2B) | defined(EXAMPLE_2C)
     static int list_b[] = {0, 2 ,3};
#else/* EXAMPLE_2A */
     static int list_b[] = {0, 2};
#endif
     int size_list_a = sizeof(list_a)/sizeof(int);
     int size_list_b = sizeof(list_b)/sizeof(int);

     ...
     MPI_Init(&argc, &argv);
     MPI_Comm_group(MPI_COMM_WORLD, &MPI_GROUP_WORLD);

     MPI_Group_incl(MPI_GROUP_WORLD, size_list_a, list_a, &group_a);
     MPI_Group_incl(MPI_GROUP_WORLD, size_list_b, list_b, &group_b);

     MPI_Comm_create(MPI_COMM_WORLD, group_a, &comm_a);
     MPI_Comm_create(MPI_COMM_WORLD, group_b, &comm_b);

     if(comm_a != MPI_COMM_NULL)
        MPI_Comm_rank(comm_a, &ma);
     if(comm_b != MPI_COMM_NULL)
        MPI_Comm_rank(comm_b, &mb);

     if(comm_a != MPI_COMM_NULL)
        lib_call(comm_a);

     if(comm_b != MPI_COMM_NULL)
     {
       lib_call(comm_b);
       lib_call(comm_b);
     }

     if(comm_a != MPI_COMM_NULL)
       MPI_Comm_free(&comm_a);
     if(comm_b != MPI_COMM_NULL)
       MPI_Comm_free(&comm_b);
     MPI_Group_free(&group_a);
     MPI_Group_free(&group_b);
```

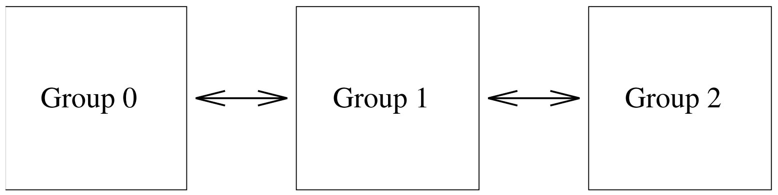

Figure 6.3: Three-group pipeline.

```
MPI::Intracomm MPI::Intercomm::Merge(bool high) const
```

This function creates an intra-communicator from the union of the two groups that are associated with intercomm. All processes should provide the same high value within each of the two groups. If processes in one group provided the value high = false and processes in the other group provided the value high = true then the union orders the "low" group before the "high" group. If all processes provided the same high argument then the order of the union is arbitrary. This call is blocking and collective within the union of the two groups.

The error handler on the new intercommunicator in each process is inherited from the communicator that contributes the local group. Note that this can result in different processes in the same communicator having different error handlers.

*Advice to implementors.* The implementation of MPI_INTERCOMM_MERGE, MPI_COMM_FREE and MPI_COMM_DUP are similar to the implementation of MPI_INTERCOMM_CREATE, except that contexts private to the input inter-communicator are used for communication between group leaders rather than contexts inside a bridge communicator. (*End of advice to implementors.*)

### 6.6.3 Inter-Communication Examples

#### Example 1: Three-Group "Pipeline"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Therefore, group 0 requires one inter-communicator, group 1 requires two inter-communicators, and group 2 requires 1 inter-communicator.

```
int main(int argc, char **argv)
{
  MPI_Comm   myComm;       /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm;  /* inter-communicator */
  MPI_Comm   mySecondComm; /* second inter-communicator (group 1 only) */
  int membershipKey;
  int rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  /* User code must generate membershipKey in the range [0, 1, 2] */
  membershipKey = rank % 3;
  /* Build intra-communicator for local sub-group */
  MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

  /* Build inter-communicators.  Tags are hard-coded. */
  if (membershipKey == 0)
  {                     /* Group 0 communicates with group 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          1, &myFirstComm);
  }
  else if (membershipKey == 1)
  {              /* Group 1 communicates with groups 0 and 2. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                          1, &myFirstComm);
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                          12, &mySecondComm);
  }
  else if (membershipKey == 2)
  {                     /* Group 2 communicates with group 1. */
    MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                          12, &myFirstComm);
  }

  /* Do work ... */

  switch(membershipKey)  /* free communicators appropriately */
  {
  case 1:
     MPI_Comm_free(&mySecondComm);
  case 0:
  case 2:
     MPI_Comm_free(&myFirstComm);
     break;
  }

  MPI_Finalize();
}
```

Example 2: Three-Group "Ring"

Groups 0 and 1 communicate. Groups 1 and 2 communicate. Groups 0 and 2 communicate. Therefore, each requires two inter-communicators.

```
int main(int argc, char **argv)
{
  MPI_Comm   myComm;      /* intra-communicator of local sub-group */
  MPI_Comm   myFirstComm; /* inter-communicators */
  MPI_Comm   mySecondComm;
  MPI_Status status;
```

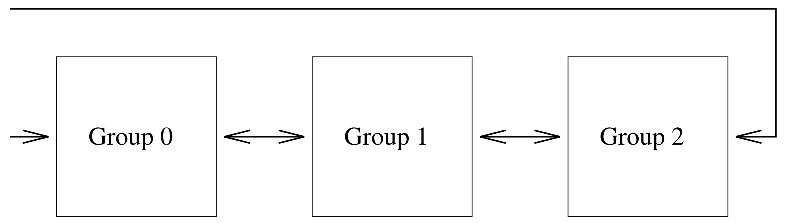

Figure 6.4: Three-group ring.

```
int membershipKey;
int rank;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...

/* User code must generate membershipKey in the range [0, 1, 2] */
membershipKey = rank % 3;

/* Build intra-communicator for local sub-group */
MPI_Comm_split(MPI_COMM_WORLD, membershipKey, rank, &myComm);

/* Build inter-communicators.  Tags are hard-coded. */
if (membershipKey == 0)
{              /* Group 0 communicates with groups 1 and 2. */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                       1, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                       2, &mySecondComm);
}
else if (membershipKey == 1)
{          /* Group 1 communicates with groups 0 and 2. */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                       1, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 2,
                       12, &mySecondComm);
}
else if (membershipKey == 2)
{         /* Group 2 communicates with groups 0 and 1. */
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 0,
                       2, &myFirstComm);
  MPI_Intercomm_create( myComm, 0, MPI_COMM_WORLD, 1,
                       12, &mySecondComm);
}

/* Do some work ... */
```

```
    /* Then free communicators before terminating... */
    MPI_Comm_free(&myFirstComm);
    MPI_Comm_free(&mySecondComm);
    MPI_Comm_free(&myComm);
    MPI_Finalize();
  }
```

Example 3: Building Name Service for Intercommunication

The following procedures exemplify the process by which a user could create name service for building intercommunicators via a rendezvous involving a server communicator, and a tag name selected by both groups.

After all MPI processes execute MPI_INIT, every process calls the example function, Init_server(), defined below. Then, if the new_world returned is NULL, the process getting NULL is required to implement a server function, in a reactive loop, Do_server(). Everyone else just does their prescribed computation, using new_world as the new effective "global" communicator. One designated process calls Undo_Server() to get rid of the server when it is not needed any longer.

Features of this approach include:

- Support for multiple name servers
- Ability to scope the name servers to specific processes
- Ability to make such servers come and go as desired.

```
#define INIT_SERVER_TAG_1   666
#define UNDO_SERVER_TAG_1   777

static int server_key_val;

/* for attribute management for server_comm,  copy callback: */
void handle_copy_fn(MPI_Comm *oldcomm, int *keyval, void *extra_state,
void *attribute_val_in, void **attribute_val_out, int *flag)
{
   /* copy the handle */
   *attribute_val_out = attribute_val_in;
   *flag = 1; /* indicate that copy to happen */
}

int Init_server(peer_comm, rank_of_server, server_comm, new_world)
MPI_Comm peer_comm;
int rank_of_server;
MPI_Comm *server_comm;
MPI_Comm *new_world;    /* new effective world, sans server */
{
    MPI_Comm temp_comm, lone_comm;
    MPI_Group peer_group, temp_group;
```