Advice to users. The use of MPI_TESTSOME is likely to be more efficient than the use of MPI_TESTANY. The former returns information on all *completed* communications, with the latter, a new call is required for each communication that completes.

A server with multiple clients can use MPI_WAITSOME so as not to starve any client. Clients send messages to the server with service requests. The server calls MPI_WAITSOME with one receive request for each client, and then handles all receives that completed. If a call to MPI_WAITANY is used instead, then one client could starve while requests from another client always sneak in first. (*End of advice to users.*)

Advice to implementors. MPI_TESTSOME should *complete* as many pending communications as possible. (*End of advice to implementors.*)

Example 3.16 Client-server code (starvation can occur).

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .GT. 0) THEN         ! client code
   DO WHILE(.TRUE.)
      CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
      CALL MPI_WAIT(request, status, ierr)
   END DO
ELSE         ! rank=0 -- server code
   DO i=1,size-1
      CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
                     comm, request_list(i), ierr)
   END DO
   DO WHILE(.TRUE.)
      CALL MPI_WAITANY(size-1, request_list, index, status, ierr)
      CALL DO_SERVICE(a(1,index))  ! handle one message
      CALL MPI_IRECV(a(1, index), n, MPI_REAL, index, tag, &
                     comm, request_list(index), ierr)
   END DO
END IF
```

Example 3.17 Same code, using MPI_WAITSOME.

```
CALL MPI_COMM_SIZE(comm, size, ierr)
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank .GT. 0) THEN         ! client code
   DO WHILE(.TRUE.)
      CALL MPI_ISEND(a, n, MPI_REAL, 0, tag, comm, request, ierr)
      CALL MPI_WAIT(request, status, ierr)
   END DO
ELSE         ! rank=0 -- server code
   DO i=1,size-1
      CALL MPI_IRECV(a(1,i), n, MPI_REAL, i, tag, &
                     comm, request_list(i), ierr)
   END DO
   DO WHILE(.TRUE.)
```

```
      CALL MPI_WAITSOME(size, request_list, numdone, &
                        indices, statuses, ierr)
      DO i=1,numdone
         CALL DO_SERVICE(a(1, indices(i)))
         CALL MPI_IRECV(a(1, indices(i)), n, MPI_REAL, 0, tag, &
                        comm, request_list(indices(i)), ierr)
      END DO
   END DO
END IF
```

3.7.6 Non-Destructive Test of status

This call is useful for accessing the information associated with a request, without *freeing* the request (in case the user is expected to access it later). It allows one to layer libraries more conveniently, since multiple layers of software may access the same *completed* request and extract from it the status information.

MPI_REQUEST_GET_STATUS(request, flag, status)

IN	request	request (handle)
OUT	flag	boolean flag, same as from MPI_TEST (logical)
OUT	status	status object if flag is true (status)

C binding

```
int MPI_Request_get_status(MPI_Request request, int *flag,
            MPI_Status *status)
```

Fortran 2008 binding

```
MPI_Request_get_status(request, flag, status, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    LOGICAL, INTENT(OUT) :: flag
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_REQUEST_GET_STATUS(REQUEST, FLAG, STATUS, IERROR)
    INTEGER REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
    LOGICAL FLAG
```

Sets flag = `true` if the operation is *complete*, and, if so, returns in status the request status. However, unlike test or wait, it does not deallocate or *inactivate* the request; a subsequent call to test, wait or free should be executed with that request. It sets flag = `false` if the operation is not *complete*.

One is allowed to call MPI_REQUEST_GET_STATUS with a *null* or *inactive* request argument. In such a case the procedure returns with flag = `true` and *empty* status.