MPI for Java:
Position Document and Draft API Specification

Bryan Carpenter[1], Vladimir Getov[2], Glenn Judd[3],
Tony Skjellum[4] and Geoffrey Fox[1]

[1]NPAC, Syracuse University, Syracuse, USA
[2]School of Computer Science, University of Westminster, London, UK
[3]Computer Science Department, Brigham Young University, Provo, USA
[4]MPI Software Technology, Inc., Starkville, USA

Technical Report JGF-TR-03

Java Grande Forum

November, 1998

Contents

1 Introduction and background

A basic prerequisite for parallel programming is a good message passing API. Java comes with various ready-made packages for communication, notably an easy-to-use interface to BSD sockets, and the Remote Method Invocation (RMI) mechanism. Interesting as these interfaces are, it is questionable whether parallel programmers will find them especially convenient. Sockets and remote procedure calls have been around for about as long as parallel computing has been fashionable, and neither of them has been popular in that field. Both communication models are optimized for client-server programming, whereas the parallel computing world is mainly concerned with "symmetric" communication, occurring in groups of interacting peers.

This symmetric model of communication is captured in the successful Message Passing Interface (MPI) standard, established a few years ago [4]. MPI directly supports the Single Program Multiple Data (SPMD) model of parallel computing, wherein a group of processes cooperate by executing identical program images on local data values. Reliable point-to-point communication is provided through a shared, group-wide communicator, instead of socket pairs. MPI allows numerous blocking, non-blocking, buffered or synchronous communication modes. It also provides a library of true collective operations (broadcast is the most trivial example). An extended standard, MPI-2 [5], allows for dynamic process creation and access to memory in remote processes.

The MPI standard document thus far has provided language-independent specification as well as language-specific (C and Fortran) bindings [4]. While the MPI-2 release of the standard added a C++ binding [5], no Java binding has been offered or is planned by the MPI Forum. With the evident success of Java as a programming language, and its inevitable use in connection with parallel as well as distributed computing, the absence of a well-designed language-specific binding for message-passing with Java will lead to divergent, non-portable practices. Therefore, a standard specification is urgently needed to enable the development of portable Java Grande applications using MPI.

2 Current status

There are several known efforts towards the design of early MPI interfaces for Java with three fully functional but different Java-MPI implementations – mpiJava, JavaMPI, and MPIJ. The design of mpiJava is based on the use of native methods to build a wrapper to existing MPI library (MPICH). A comparable approach has been followed in the development of JavaMPI, but the JavaMPI wrappers were automatically generated by a special-purpose code generator. A large subset of MPI is implemented in pure Java within the DOGMA system for Java-based parallel programming. MPI Software Technology, Inc. have also announced that there is a commercial effort under way to develop a message-passing framework and parallel support environment for Java called JMPI [2]. Some of these "proof-of-concept" implementations have been available for more than a year with successful ports on clusters of workstations running Solaris or Windows NT, as well as IBM SP2, SGI Origin-2000, Fujitsu AP3000, and Hitachi SR2201 parallel platforms.

2.1 The mpiJava wrapper

The mpiJava software implements a Java binding for MPI proposed late in 1997 [1]. That proposal built on work on Java wrappers for MPI started at NPAC about a year earlier.

The mpiJava API is modelled as closely as practical on the C++ binding defined in the MPI 2.0 standard, specifically supporting the MPI 1.1 subset of that standard. In some cases the extra runtime information available in Java objects allows argument lists to be simplified relative to the C++ binding. In other cases restrictions of Java, especially the fact that all arguments are passed by value in Java, forces some changes to argument lists. But in general mpiJava adheres closely to earlier standards.

The implementation of mpiJava is through JNI wrappers to native MPI software. Interfacing Java to MPI is not always trivial. We often see low-level conflicts between the Java runtime and the interrupt mechanisms used in MPI implementations. The situation is improving as JDK matures, and the mpiJava software now works reliably on top of Solaris MPI implementations and various shared memory platforms. A port to Windows NT (based on WMPI) is available, and other ports are in progress.

Other work in progess includes development of demonstrator applications, and Java-specific extensions such as support for direct communication of serializable objects.

2.2 Automatic generation of MPI wrappers

In principle, the binding of existing MPI library to Java using JNI amounts to either dynamically linking the library to the Java virtual machine, or linking the library to the object code produced by a stand-alone Java compiler. Complications stem from the fact that Java data formats are in general different from those of C. Java implementations will have to use JNI which allows C functions to access Java data and perform format conversion if necessary. Such an interface is fairly convenient for writing *new* C code to be called from Java, but is not adequate for linking *existing* native code.

Clearly an additional interface layer must be written in order to bind a legacy library to Java. A large library like MPI has over a hundred exported functions, therefore it is preferable to automate the creation of the additional interface layer. The *Java-to-C interface generator* (JCI) [6] takes as input a header file containing the C function prototypes of the native library. It outputs a number of files comprising the additional interface: a file of C stub-functions; files of Java class and native method declarations; shell scripts for doing the compilation and linking. The JCI tool generates a C stub-function and a Java native method declaration for each exported function of the MPI library. Every C stub-function takes arguments whose types correspond directly to those of the Java native method, and converts the arguments into the form expected by the C library function.

As the JavaMPI bindings have been generated automatically from the C prototypes of MPI functions, they are very close to the C binding. However, there is nothing to prevent from parting with the C–style binding and adopting a Java-style object–oriented approach by grouping MPI functions into a hierarchy of classes.

2.3 Pure Java implementation of MPI

MPIJ is a completely Java-based implementation of MPI which runs as part of the Distributed Object Group Metacomputing Architecture (DOGMA) system. Being closely based on the C++ binding, MPIJ implements a large subset of MPI functionality including point-to-piont communication (all modes), intra-communicator operations, groups, user-defined reduction operations. Noteable capabilities that are not yet implemented include process topologies, caching, intercommunicators, and user-defined datatypes (these are arguably needed for legacy code only).

MPIJ communication uses native marshaling of primitive Java types. This technique allows MPIJ to achieve communication speeds comparable to, and frequently exceeding that, of native MPI implementations. (Java communication speed would be greatly increased if native marshaling were a core Java function.)

Current MPIJ work involves porting native marshalling to platforms other than Win32, investigation of standard libraries (e.g. BLAS) for improved performance, and porting to an improved version of DOGMA.

3 Proposed joint efforts

3.1 Java-MPI – API Specification

The MPI standard is explicitly object-based. The C and Fortran bindings rely on "opaque objects" that can be manipulated only by acquiring object handles from constructor functions, and passing the handles to suitable functions in the library. The C++ binding specified in the MPI-2 standard collects these objects into suitable class hierarchies and defines most of the library functions as class member functions. The Java-MPI API specification follows this model, lifting the structure of its class hierarchy directly from the C++ binding.

The immediate infrastructure to be provided builds literally on the MPI-1 infrastructure offered by the MPI Forum, together with language bindings motivated by the MPI-2 forum's C++ bindings. The purpose of that effort is to provide an immediate, ad hoc standardization for common message passing programs in Java, as well as to provide a basis for conversion between C, C++, Fortran77, and Java. Eventually, support for aspects of MPI-2 belong under this category as well, particularly dynamic process management, but not necessarily all of MPI-2, given its spartan implementation in the non-Java space.

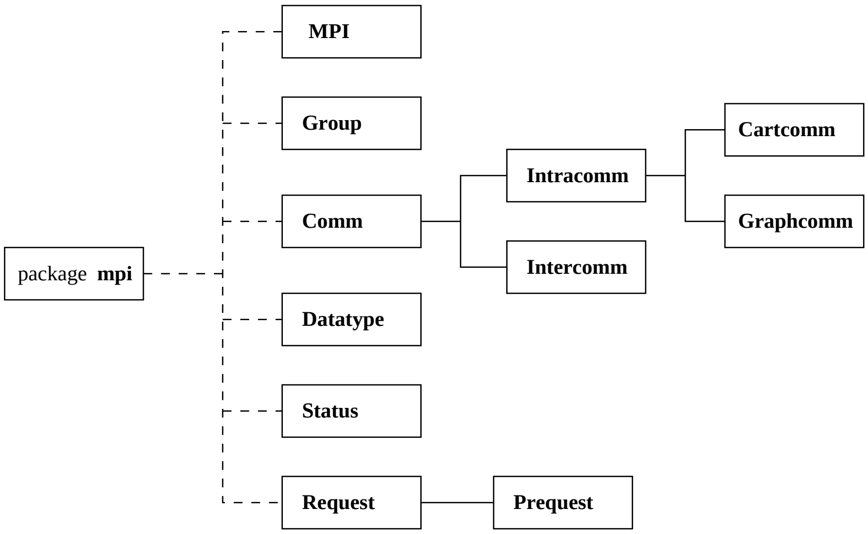

Figure 1: Principal classes of Java-MPI

The major classes of Java-MPI are illustrated in Figure 1. The class `MPI` only has static members. It acts as a module containing global services, such as initialization of MPI, and many global constants including the default communicator `COMM_WORLD`. The most important class in the package is the communicator class `Comm`. All communication functions in Java-MPI are members of `Comm` or its subclasses. As usual in MPI, a communicator stands for a "collective object" logically shared by a group of processors. The processes communicate, typically by addressing messages to their peers through the common communicator. Another class that is important for the Java-MPI specification is the `Datatype` class. This describes the type of the elements in the message buffers passed to send, receive, and all other communication functions.

The complete draft Java-MPI API specification is included as an appendix to this document.

3.2 Advanced Message Passing Infrastructure for Java

The present effort's purpose is to offer a first principles study of how to present MPI-like services to Java programs, in an upward compatible fashion. The purposes are twofold: performance and portability. For performance, we seek to take advantage of what has been learned since MPI-1 and MPI-2 were finalized, or which were ignored in MPI standardization for various reasons. The study will, for instance, draw on the body of knowledge just recently completed within the MPI/RT Forum, which strives to enhance both real-time and performance of message passing programs. From MPI/RT, we will at least glean design hints concerning channel abstractions, and the more direct use of object-oriented design for message passing than was done in MPI-1 or MPI-2 (despite existence of C++ bindings). Additionally, a fundamental look at data marshalling and unmarshalling in the Java context will be undertaken, and preference for Java-natural mechanisms and policies will be attempted. Along the lines of portability, a detachment from legacy implementations of Java over existing native methods will be emphasized, while also considering the possibility of layering the messaging middleware over standard transports and other Java-compliant middleware (such as CORBA). In a sense, the middleware developed at this level should offer a choice of a performance or generality emphasis, while always supporting portability. A policy/opportunity to support aspects of real-time and fault detection/fault-aware programs will be studied and standardized insofar as possible, again drawing on the concepts learned in the MPI/RT activity, and also drawing on experience from distributed computing real-time activities. The validity of this type of messaging middleware in the embedded and real-time Java application spaces will also be considered.

3.3 Further Actions

The ACG currently consideres to release a Request for Proposals for the Java-MPI efforts to cover:

- Standard Java-MPI API Specification
- Java-MPI wrapper publicly available on the Web
- Intelligent generator of wrappers to legacy MPI libraries
- Pure Java MPI implementation
- Test suite
- Java-MPI Benchmarks

A A Draft Java binding for MPI

This appendix outlines a Java language binding for MPI 1.1. It has been adapted from the proposal in reference [1], incorporating especially suggestions and comments from Vladimir Getov and Glen Judd. It is presented here as a basis for further discussion and comment.

This is not a standalone specification of the behaviour of MPI—it is meant to be read in conjunction with the MPI standard document. Subsections are laid out in the same way as in the standard document, to allow cross-referencing. Where the Java binding makes no significant change to a particular section of the standard document, we will just note here that there are no special issues for the Java binding. This does not mean that the corresponding section of the standard is irrelevant to the Java binding—it may mean it is 100% relevant! Where practical the current proposals are modelled on the MPI C++ interface defined in the MPI standard version 2.0.

Changes to the mpiJava interface:

- Name conventions have been shifted to bring them into line with Sun's Java code conventions [7].
- The `MPI.OBJECT` basic type has been added.
- The public fields of `Status` have been replaced by inquiry methods.
- The MPI exception classes are no longer specified to be subclasses of `IOException`.
- The `Request` method `is_null` has been replaced by `isVoid`, and the constant `MPI.REQUEST_NULL` is removed in favour of providing a default constructor for `Request`.
- Derived type constructors `contiguous`, `vector`, `hvector`, `indexed`, `hindexed` become non-static methods on *oldtype*.
- A new variant of `Comm.pack` has been added.
- The bindings of *MPI_ATTR_PUT* and *MPI_ATTR_DELETE* have been removed.
- The `translateRanks` member of `group` becomes a non-static method on *group2*.
- The interface to `Cartcomm.dimsCreate` has been corrected.

Controversies and open questions:

- It is unclear whether the Java interface should support MPI derived data types. A proposal for a Java-compatible subset of derived types is included in this document, but deleting it could simplify the API significantly.

- It has been suggested that many of the communication operations should be overloaded to provide simplified variants that omit arguments such as `offset`, `count` (possibly `datatype`). This suggestion is not included in the current proposal, but it could be added later if there is general support. The obvious counter-argument is that it significantly increases the total number of methods in the API.

- It has been suggested that some specific support for multidimensional arrays may be desirable. In the current proposal, communicating multidimensional arrays depends on Java object serialization, which might be a performance bottleneck. A plausible position is to wait and see what happens in Java regarding multidimensional arrays and efficient object serialization before complicating this API.

- It has been suggested that the specification of `UserFunction` could be altered to improve type security. This suggestion isn't incorporated in the current draft, but perhaps it should be.

A.1 Introduction to MPI

Evidently, this document adds Java to the C and Fortran bindings defined in the MPI standard. Otherwise no special issues for the Java binding.

A.2 MPI Terms and Conventions

A.2.1 Document Notation

No special issues for Java binding.

A.2.2 Procedure Specification

In general we use *italicized* names to refer to entities in the MPI language independent procedure definitions, and `typewriter` font for concrete Java entities.

As a rule Java argument names will be the same as the corresponding language independent names. In non-static methods of `Comm`, `Status`, `Request`, `Datatype`, `Op` or `Group` (and subclasses), the class instance generally stands for the argument called *comm*, *status*, *request*, *datatype*, *op* or *group*, respectively in the language independent procedure definition[1]. Names may be altered to bring them into line with Sun's Java code conventions, as discussed in section A.2.5. For example, underscores in variable names are typically removed in favour of capitalizing internal words.

A.2.3 Semantic terms

No special issues for Java binding.

A.2.4 Data types

Opaque objects will be presented as Java objects. This introduces the option of simplifying the user's task in managing these objects. MPI destructors can be absorbed into Java object destructors, which are called automatically by the Java garbage collector. We adopt this strategy as the general rule. Explicit calls to MPI destructor functions are typically omitted from the Java user interface. An exception is made for the `Comm` classes. *MPI_COMM_FREE* is a collective operation, so the user must ensure that calls are made at consistent times by all processors involved—the call can't be left to the vagaries of the garbage collector.

> *Advice to implementors.* For JNI-based "wrapper" implementations based on the C binding of MPI, one should ensure that all MPI functions, including the `MPI_..._free` object destructors, are called *before* `MPI_Finalize` is called. Because invocation of Java destructors is under the control of the garbage collector there is no guarantee that all Java-level `finalize` methods will be called before the explicit call to the Java-level `MPI.finish`. A possible solution is to maintain a global count of the number of outstanding `MPI_..._free` calls. This is initialized to 1

[1] Exceptions to this rule include the derived datatype construction methods on `Datatype` and the `translateRank` method on `Group`.

by `MPI.init` and incremented by relevant constructors. All relevant Java `finalize` methods and `MPI.finish` should decrement this counter, and be prepared to call the physical `MPI_Finalize` function when and only when the count falls to zero. *(End of advice to implementors.)*

A.2.5 Language Binding

Naming Conventions All MPI classes belong to the package `mpi`. Conventions for capitalization, etc, in class and member names generally follow the recommendations of Sun's Java code conventions [7]. Class names are in mixed case with the first letter of each internal word capitalized. Method and ordinary variable names are in mixed case, with the first letter lowercase. Constant variables are all uppercase with words separated by underscores ("_").

Rationale. In general these conventions are consistent with the naming conventions of the MPI 2.0 C++ standard. Notable exceptions include the use of lower case for the first letters of method names, and avoidance of underscore in variable names. *(End of rationale.)*

Restrictions on *struct* derived type. Some options allowed for *derived data types* in the C and Fortran binding are deleted in the proposed Java binding. The Java VM does not incorporate a concept of a global linear address space. Passing physical addresses to data type definitions is not allowed. The use of the *MPI_TYPE_STRUCT* datatype constructor is also restricted in a way that makes it impossible to send mixed *basic datatypes* in a single message. Since, however, the set of basic datatypes recognised by MPI is extended to include serializable Java *objects*, this should not be a serious restriction in practice.

Multidimensional arrays and offsets. The C and Fortran languages define a straightforward mapping (or "sequence association") between their multidimensional arrays and equivalent one-dimensional arrays. So in C or Fortran a multidimensional array passed as a message buffer argument is first interpreted as a one-dimensional array with the same element type as the original multidimensional array. Offsets in the buffer (such as offsets occuring in derived data types) are then interpretted in terms of the effective one-dimensional array (or—equivalent up to a constant factor—in terms of physical memory). In Java the relationship between multidimensional arrays and one dimensional arrays is different. An "n-dimensional array" is equivalent to a one-dimensional array of $(n-1)$-dimensional arrays. In the proposed Java binding, message buffers are always one-dimensional arrays. The element type *may* be an object, which *may* have array type. Hence multidimensional arrays can appear as message buffers, but the interpretation is subtly different. In distinction to the C or Fortran case *offsets in multidimensional message buffers are always interpretted as offsets in the outermost one-dimensional array.*

Start of message buffer. C and Fortran both have devices for treating a section of an array, offset from the beginning of the array, as if it was an array in its own right. Java doesn't have any such mechanism. To provide the same flexibility, an `offset` parameter is associated with any buffer argument. This defines the position of the first actual buffer element in the Java array.

Error codes. Unlike the C and Fortran interfaces, the Java interfaces to MPI calls will not return explicit error codes. The Java exception mechanism will be used to report errors.

> *Rationale.* The exception mechanism is very widely used by Java libraries. It is inconvenient to use up the single return value of a Java function with an error code. (Java doesn't allow function arguments to be passed by reference, so returning multiple values tends to be more clumsy than in other languages.) *(End of rationale.)*

Multiple return values. A few functions in the MPI interface return multiple values, even after the error code is eliminated. This is dealt with in the proposed binding in various ways. Sometimes an MPI function initializes some elements in an array and also returns a count of the number of elements modified. In Java we typically return an array result, omitting the count. The count can be obtained subsequently from the `length` member of the array. Sometimes an MPI function initializes an object conditionally and returns a separate flag to say if the operation succeeded. In Java we typically return an object reference which is `null` if the operation fails. Occasionally extra internal state is added to an existing MPI class to hold extra results—for example the `Status` class has extra state initialized by functions like `Waitany` to hold the *index* value. Rarely none of these methods work and we resort to defining auxilliary classes to hold multiple results from a particular function.

Array count arguments. The proposed binding often omits array size arguments, because they can be picked up within the function by reading the `length` member of the array argument. A major exception is for message buffers, where an explicit count is always given.

> *Rationale.* In the proposed binding, message buffers have explicit `offset` and `count` arguments whereas other kinds of array argument typically do not. Message buffers aside, typical array arguments to MPI functions (eg, vectors of request structures) are small arrays. If subsections of these must be passed to an MPI function, the sections can be copied to smaller arrays at little cost. In contrast message buffers are typically large and copying them is expensive, so it is worthwhile to pass the extra arguments. Also, if derived data types are being used, the required value of the `count` argument is always different to the buffer length. *(End of rationale.)*

Concurrent access to arrays.

> *Advice to implementors.* In JNI-based wrapper implementations it may be necessary to impose some non-interference rules for concurrent read and write operations on arrays. When an array is passed to an MPI method such as a send or receive operation, the wrapper code will probably extract a pointer to the contents of the array using a JNI `Get...ArrayElements` routine. If the garbage collector *does not* support "pinning" (temporarily disabling run-time relocation of data for specific arrays—see [3] for more discussion), the pointer returned by this `Get` function may be to a temporary copy of the elements. The copy will be written back to the true Java array when a subsequent call to `Release...ArrayElements` is made. If two operations involving the same array are active concurrently, this copy-back may result in failure to save modifications made by one or more of the concurrent calls.
>
> Such an implementation may have to enforce a safety rule such as: *when several MPI send or receive (etc) operations are active concurrently, if any one of those operations writes to a particular array, none of the other operations must read or write* any portion *of that array.*
>
> If the garbage collector supports pinning, this problem does not arise. *(End of advice to implementors.)*

A.2.6 Processes

No special issues for Java binding.

A.2.7 Error Handling

As explained in section A.2.5, the Java methods do not return error codes. The Java exceptions thrown instead are defined in section A.7.3.

A.3 Point-to-Point Communication

A.3.1 Introduction

In general the Java binding of point-to-point communication operations will realize the MPI functions as methods of the Comm class. The basic point-to-point communication operations are *send* and *receive*. Their use is illustrated in the example below.

```
import mpi.* ;

class Hello {
  static public void main(String[] args) {
    MPI.init(args) ;

    int myrank = MPI.COMM_WORLD.rank() ;
    if(myrank == 0) {
      char [] message = "Hello, there".toCharArray() ;
      MPI.COMM_WORLD.send(message, 0, message.length, MPI.CHAR, 1, 99) ;
    }
    else {
      char [] message = new char [20] ;
      MPI.COMM_WORLD.recv(message, 0, 20, MPI.CHAR, 0, 99) ;
      System.out.println("received:" + new String(message) + ":") ;
    }

    MPI.finish();
  }
}
```

A.3.2 Blocking Send and Receive operations

```
void Comm.send(Object buf, int offset, int count,
               Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Blocking send operation. Java binding of the MPI operation *MPI_SEND*. The data part of the message consists of a sequence of `count` values, each of the type indicated by `datatype`. The actual argument associated with `buf` must be an array. The value `offset` is a subscript in this array, defining the position of the first item of the message.

The elements of `buf` may have primitive type or class type. If the elements are objects, they must be serializable objects. If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`: the basic MPI datatypes supported, and their correspondence to Java types, are as follows

MPI datatype	Java datatype
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.OBJECT	`Object`

If the `datatype` argument represents an MPI derived type, its *base type* must agree with the element type of `buf` (see section A.3.12).

> *Rationale.* The `datatype` argument is not redundant in Java, because this proposal includes support for MPI derived types. If it was decided to remove derived types from the API, `datatype` arguments could be removed from various functions, and Java runtime inquiries could be used internally to extract the element type of the buffer, or methods like `send` could be overloaded to accept buffers with elements of the 9 basic types. (The disadvantage of the latter approach is that it leads to a large proliferation in the number of methods. Historically MPI has eschewed this kind of overloading. Java at least provides runtime mechanisms for checking type correctness, so the extra security provided by overloading is probably not an essential requirement.) *(End of rationale.)*

If a data type has `MPI.OBJECT` as its base type, the objects in the buffer will be transparently serialized and unserialized inside the communication operations.

> *Advice to implementors.* While straightforward in principle, this extension involves several complications for JNI-based wrapper implementations. The size of the physical receive buffer cannot be computed in advance, so it may be necessary to split the physical message into a header message containing buffer size, followed by the data. This notably complicates wrappers for communication methods based on `Request` classes. Also, the wrapper has to be prepared to interpret derived types with `MPI.OBJECT` as the base type, because of course the underlying C MPI cannot do this. *(End of advice to implementors.)*

```
Status Comm.recv(Object buf, int offset, int count,
                 Datatype datatype, int source, int tag)
```

`buf`	receive buffer array
`offset`	initial offset in receive buffer
`count`	number of items in receive buffer
`datatype`	datatype of each item in receive buffer
`source`	rank of source
`tag`	message tag
returns:	status object

Blocking receive operation. Java binding of the MPI operation *MPI_RECV*. The actual argument associated with `buf` must be an array. The value `offset` is a subscript in this array, defining the position into which the first item of the incoming message will be copied.

The elements of `buf` may have primitive type or class type. If the `datatype` argument represents an MPI basic type, its value must agree with the element type of `buf`; if `datatype` represents an MPI derived type, its *base type* must agree with the element type of `buf` (see section A.3.12).

The MPI constants *MPI_ANY_SOURCE* and *MPI_ANY_TAG* are available as `MPI.ANY_SOURCE` and `MPI.ANY_TAG`.

The following methods can be used to interrogate the return status of a receive operation.

```
int Status.getSource()
```

returns: source of the received message.

```
int Status.getTag()
```

returns: tag of the received message.

> *Rationale.* Following common object-oriented practices, source and tag values are returned by inquiry functions `getSource` and `getTag`, rather than through publicly accessible fields in the `Status` class. *(End of rationale.)*

```
int Status.getCount(Datatype datatype)
```

`datatype`	datatype of each item in receive buffer
returns:	number of received entries

Java binding of the MPI operation *MPI_GET_COUNT*.

A.3.3 Data type matching and data conversion

The Java language definition places quite detailed constraints on the representation of its primitive types—for example it requires conformance with IEEE 754 for `float` and `double`. There may still be a requirement for representation conversion in heterogenous systems. For example, source and destination computers (or virtual machines) may have different endianess.

A.3.4 Communication Modes

```
void Comm.bsend(Object buf, int offset, int count,
                Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Send in buffered mode. Java binding of the MPI operation *MPI_BSEND*. Further comments as for `send`.

```
void Comm.ssend(Object buf, int offset, int count,
                Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Send in synchronous mode. Java binding of the MPI operation *MPI_SSEND*. Further comments as for `send`.

```
void Comm.rsend(Object buf, int offset, int count,
                Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag

Send in ready mode. Java binding of the MPI operation *MPI_RSEND*. Further comments as for `send`.

A.3.5 Semantics of point-to-point communication

No special issues for Java binding.

A.3.6 Buffer allocation and usage

```
void MPI.bufferAttach(byte [] buffer)
```

`buffer`	buffer array

Provides to MPI a buffer in user's memory to be used for buffering outgoing messages. Java binding of the MPI operation *MPI_BUFFER_ATTACH*.

```
byte [] MPI.bufferDetach()
```

returns:	buffer array

Detach the buffer currently associated with MPI. Java binding of the MPI operation *MPI_BUFFER_DETACH*. The MPI constant *MPI_BSEND_OVERHEAD* is available as `MPI.BSEND_OVERHEAD`.

[It is probably impossible to implement this function properly in a JNI interface to native Java, because there is no way to return the system-defined buffer as a Java array.]

A.3.7 Nonblocking communication

Nonblocking communications use methods of the `Request` class to identify communication operations and match the operation that initiates the communication with the operation that terminates it.

```
Request Comm.isend(Object buf, int offset, int count,
                   Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	communication request

Start a standard mode, nonblocking send. Java binding of the MPI operation *MPI_ISEND*. Further comments as for `send`.

```
Request Comm.ibsend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	communication request

Start a buffered mode, nonblocking send. Java binding of the MPI operation *MPI_IBSEND*. Further comments as for `send`.

```
Request Comm.issend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	communication request

Start a synchronous mode, nonblocking send. Java binding of the MPI operation *MPI_ISSEND*. Further comments as for `send`.

```
Request Comm.irsend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	communication request

Start a ready mode, nonblocking send. Java binding of the MPI operation *MPI_IRSEND*. Further comments as for `send`.

```
Request Comm.irecv(Object buf, int offset, int count,
                   Datatype datatype, int source, int tag)
```

`buf`	receive buffer array
`offset`	initial offset in receive buffer
`count`	number of items in receive buffer
`datatype`	datatype of each item in receive buffer
`source`	rank of source
`tag`	message tag
returns:	communication request

Start a nonblocking receive. Java binding of the MPI operation *MPI_IRECV*. Further comments as for `recv`.

The following functions are used to complete nonblocking communication operations (and also communications started using the persistent communication requests—subclass `Prequest`—introduced later). We use the following terminology. A request is "active" if it is associated with an ongoing communication. Otherwise it is inactive. An inactive instance of the base class `Request` is called a "void request". Note, however, that an inactive instance of the `Prequest` subclass is not said to be "void", because it retains detailed information about a communication pattern even when no corresponding communication is ongoing.

> *Rationale.* A "void request" corresponds to what is called a "null handle" in the C and Fortran MPI bindings. It seems impractical to have completion operations like `wait` set request object references to null references in the Java sense (because Java methods cannot directly modify references passed to them as arguments). To avoid a confusing semantic distinction between null MPI handles and null Java references we introduce the terminology of a "void request object". The `Request` default constructor can be used to explicitly create a void request, replacing assignment of the value *MPI_REQUEST_NULL*. The inquiry `Request.isVoid` can be used to determine if a particular request is void, replacing a test for equality to *MPI_REQUEST_NULL*. *(End of rationale.)*

```
Status Request.wait()
```

returns: status object

Blocks until the operation identified by the request is complete. Java binding of the MPI operation *MPI_WAIT*. After the call returns, the request object becomes inactive.

`Status Request.test()`

returns: status object or null reference

Returns a status object if the operation identified by the request is complete, or a null reference otherwise. Java binding of the MPI operation *MPI_TEST*. After the call, if the operation is complete (ie, if the return value of `test` is non-null), the request object becomes an inactive request.

`Request.Request()`

Default constructor for `Request`. Constructs a void request.

`boolean Request.isVoid()`

returns: true if the request object is void, false otherwise

Note that `isVoid` is always false on instances of the subclass `Prequest`.

`void Request.finalize()`

> *Advice to implementors.* In a JNI-based wrapper implementation, the finalize method for `Request` should call `MPI_Request_free` only if `isVoid()` is false. If the request is void, for example because a `wait` operation on the object recently succeeded, the `MPI_Request_free` call is redundant. *(End of advice to implementors.)*

`static Status Request.waitAny(Request [] arrayOfRequests)`

`arrayOfRequests` array of requests

returns: status object

Blocks until one of the operations associated with the active requests in the array has completed. Java binding of the MPI operation *MPI_WAITANY*. The index in `arrayOfRequests` for the request that completed can be obtained from the status object through the `Status.getIndex` inquiry. The corresponding element of `arrayOfRequests` becomes inactive.

The `arrayOfRequests` may contain inactive requests. If the list containts no active requests, the method immediately returns a status for which the `getIndex` inquiry will return `MPI.UNDEFINED`.

`static Status Request.testAny(Request [] arrayOfRequests)`

`arrayOfRequests`	array of requests
returns:	status object or null reference

Tests for completion of either one or none of the operations associated with active requests. Java binding of the MPI operation *MPI_TESTANY*. If some request completed, the index in `arrayOfRequests` of that request can be obtained from the status object through the `Status.getIndex()` inquiry. The corresponding element of `arrayOfRequests` becomes inactive. If no request completed, `testAny` returns a null reference.

The `arrayOfRequests` may contain inactive requests. If the list containts no active requests, the method immediately returns a status for which the `getIndex` inquiry will return `MPI.UNDEFINED`.

`int Status.getIndex()`

returns:	index of the the operation that completed.

This method can be applied to any status object returned by `Request.waitAny`, `Request.testAny`, `Request.waitSome`, or `Request.testSome`.

`static Status [] Request.waitAll(Request [] arrayOfRequests)`

`arrayOfRequests`	array of requests
returns:	array of status objects

Blocks until all of the operations associated with the active requests in the array have completed. Java binding of the MPI operation *MPI_WAITALL*. The result array will be the same size as `arrayOfRequests`. On exit, requests become inactive. If the *input* value of `arrayOfRequests` contains any inactive requests, corresponding elements of the result array will contain null status references.

`static Status [] Request.testAll(Request [] arrayOfRequests)`

`arrayOfRequests`	array of requests
returns:	array of status objects, or a null reference

Tests for completion of *all* of the operations associated with active requests. Java binding of the MPI operation *MPI_TESTALL*. If all operations have completed, the exit values of the argument array and the result array are as for `waitAll`. If any operation has not completed, the result value is null and no element of the argument array is modified.

```
static Status [] Request.waitSome(Request [] arrayOfRequests)
```

`arrayOfRequests`	array of requests
returns:	array of status objects

Blocks until at least one of the operations associated with the active requests in the array has completed. Java binding of the MPI operation *MPI_WAITSOME*. The size of the result array will be the number of operations that completed. The index in `arrayOfRequests` for each request that completed can be obtained from the returned status objects using the `Status.getIndex` inquiry. The corresponding elements in `arrayOfRequests` become inactive.

If `arrayOfRequests` list contains no active requests, `testAll` immediately returns a null reference.

```
static Status [] Request.testSome(Request [] arrayOfRequests)
```

`arrayOfRequests`	array of requests
returns:	array of status objects

Behaves like `waitSome`, except that it returns immediately. Java binding of the MPI operation *MPI_TESTSOME*. If no operation has completed, `testSome` returns an array of length zero and elements of `arrayOfRequests` are unchanged. Otherwise, arguments and return value are as for `waitSome`.

A.3.8 Probe and Cancel

```
Status Comm.iprobe(int source, int tag)
```

`source`	source rank
`tag`	tag value
returns:	status object or null reference

Check if there is an incoming message matching the pattern specified. Java binding of the MPI operation *MPI_IPROBE*. If such a message is currently available, a status object similar to the return value of a matching `recv` operation is returned. Otherwise a null reference is returned.

```
Status Comm.probe(int source, int tag)
```

`source`	source rank
`tag`	tag value
returns:	status object or null reference

Wait until there is an incoming message matching the pattern specified. Java binding of the MPI operation *MPI_PROBE*. Returns a status object similar to the return value of a matching `recv` operation.

```
void Request.cancel()
```

Mark a pending nonblocking communication for cancellation. Java binding of the MPI operation *MPI_CANCEL*.

```
boolean Status.testCancelled()
```

returns:	true if the operation was succesfully cancelled, false otherwise

Test if communication was cancelled. Java binding of the MPI operation *MPI_-TEST_CANCELLED*.

A.3.9 Persistent communication requests

```
Prequest Comm.sendInit(Object buf, int offset, int count,
                       Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a standard mode send. Java binding of the MPI operation *MPI_SEND_INIT*. Further comments as for `send`.

```
Prequest Comm.bsendInit(Object buf, int offset, int count,
                        Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a buffered mode send. Java binding of the MPI operation *MPI_BSEND_INIT*. Further comments as for `send`.

```
Prequest Comm.ssendInit(Object buf, int offset, int count,
                        Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a synchronous mode send. Java binding of the MPI operation *MPI_SSEND_INIT*. Further comments as for `send`.

```
Prequest Comm.rsendInit(Object buf, int offset, int count,
                        Datatype datatype, int dest, int tag)
```

`buf`	send buffer array
`offset`	initial offset in send buffer
`count`	number of items to send
`datatype`	datatype of each item in send buffer
`dest`	rank of destination
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a ready mode send. Java binding of the MPI operation *MPI_RSEND_INIT*. Further comments as for `send`.

```
Prequest Comm.recvInit(Object buf, int offset, int count,
                       Datatype datatype, int source, int tag)
```

`buf`	receive buffer array
`offset`	initial offset in receive buffer
`count`	number of items in receive buffer
`datatype`	datatype of each item in receive buffer
`source`	rank of source
`tag`	message tag
returns:	persistent communication request

Creates a persistent communication request for a receive operation. Java binding of the MPI operation *MPI_RECV_INIT*. Further comments as for `recv`.

```
void Prequest.start()
```

Activate a persistent communication request. Java binding of the MPI operation *MPI_START*. The communication is completed by using the request in one of the operations `Request.wait`, `Request.test`, `Request.waitAny`, `Request.testAny`, `Request.waitAll`, `Request.testAll`, `Request.waitSome`, or `Request.testSome`. On successful completion the request becomes inactive again. It can be reactivated by a further call to `start`.

```
static void Prequest.startAll(Prequest [] arrayOfRequests)
```

`arrayOfRequests`	array of persistent communication requests

Activate a list of communication requests. Java binding of the MPI operation *MPI_STARTALL*.

A.3.10 Send-receive

```
Status Comm.sendrecv(Object sendbuf, int sendoffset,
                     int sendcount, Datatype sendtype,
                     int dest, int sendtag,
                     Object recvbuf, int recvoffset,
                     int recvcount, Datatype recvtype,
                     int source, int recvtag)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`dest`	rank of destination
`sendtag`	send tag
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of each item in receive buffer
`source`	rank of source
`recvtag`	receive tag
returns:	status object

Execute a blocking send and receive operation. Java binding of the MPI operation *MPI_SENDRECV*. Further comments as for `send` and `recv`.

```
Status Comm.sendrecvReplace(Object buf, int offset,
                            int count, Datatype datatype,
                            int dest, int sendtag,
                            int source, int recvtag)
```

`buf`	buffer array
`offset`	initial offset in buffer
`count`	number of items to send
`datatype`	datatype of each item in buffer
`dest`	rank of destination
`sendtag`	send tag
`source`	rank of source
`recvtag`	receive tag
returns:	status object

Execute a blocking send and receive operation, receiving message into send buffer. Java binding of the MPI operation *MPI_SENDRECV_REPLACE*. Further comments as for `send` and `recv`.

A.3.11 Null processes

The constant *MPI_PROC_NULL* is available as `MPI.PROC_NULL`.

A.3.12 Derived datatypes

In C or Fortran bindings of MPI, derived datatypes have two roles. One is to allow messages to contain mixed types (for example they allow an integer count followed by a sequence of real numbers to be passed in a single message). The other is to allow noncontiguous data to be transmitted. In the Java binding presented here the first role is abandoned. Any derived type can only include elements of a single basic type.

> *Rationale.* In the C binding of MPI, for example, the *MPI_TYPE_-STRUCT* constructor for derived types might be used to describe the physical layout of a *struct* containing mixed types. This will not work in Java, because Java does not expose the low-level layout of its objects. In C and Fortran another use of *MPI_TYPE_STRUCT* involves incorporating offsets computed as differences between absolute addresses, so that parts of a message can come from separately declared entities. It might be possible to contrive something analogous in a Java binding, somehow encoding object references instead of physical addresses. Such a device is likely to be cumbersome—even in the C and Fortran the mechanism is not particularly elegant. Meanwhile, the effect of either of these applications of *MPI_TYPE_STRUCT* can be achieved by using `MPI.OBJECT` as the buffer type, and relying on Java object serialization. *(End of rationale.)*

This leaves description of noncontiguous buffers as the essential role for derived data types in the Java binding.

Every derived data type constructable in the Java binding has a uniquely defined *base type*. This is one of the 9 basic types enumerated in section A.3.2. Derived types inherit their base types from their precursors in a straightforward way.

In the proposed binding a **general datatype** is an object that specifies two things

- A base type
- A sequence of integer displacements

In contrast to the C and Fortran bindings the displacements are in terms of subscripts in the buffer array argument, *not* byte displacements.

The base types for the predefined MPI datatypes are

MPI datatype	base type
MPI.BYTE	`byte`
MPI.CHAR	`char`
MPI.SHORT	`short`
MPI.BOOLEAN	`boolean`
MPI.INT	`int`
MPI.LONG	`long`
MPI.FLOAT	`float`
MPI.DOUBLE	`double`
MPI.OBJECT	`Object`
MPI.LB	$\perp$
MPI.UB	$\perp$
MPI.PACKED	`byte`

The symbol $\perp$ is a special undefined value. The displacement sequence for the predefined types consists of a single zero.

If the displacement sequence of a datatype is

$$DispSeq = \{disp_0, \ldots, disp_{n-1}\}$$

then

$$\begin{aligned} lb(DispSeq) &= \min_j disp_j, \\ ub(DispSeq) &= \max_j(disp_j + 1), \quad \text{and} \\ extent(DispSeq) &= ub(DispSeq) - lb(DispSeq) \end{aligned}$$

> *Rationale.* This definition of the extent differs from the definition in the C or Fortran. It is in units of the buffer array index, *not* in units of bytes. *(End of rationale.)*

> *Advice to implementors.* In a JNI-based wrapper implementation, the return values of the `extent`, `size`, `lb` and `ub` inquiries will have to be suitably scaled from the result obtained from the corresponding C inquiries. *(End of advice to implementors.)*

These definitions have to be modified if the type construction involves `MPI.LB`, `MPI.UB`.

`Datatype Datatype.contiguous(int count)`

`count` replication count

returns: new datatype

Construct new datatype representing replication of this datatype into contiguous locations. Java binding of the MPI operation *MPI_TYPE_CONTIGUOUS*. The base type of the new datatype is the same as the base type of this type. Assume the displacement sequence of this type is

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent ex. Then the new datatype has a displacement sequence with `count` $\cdot$ `n` entries defined by:

$$\begin{aligned}\{ \quad & disp_0, \ldots, disp_{n-1}, \\ & disp_0 + ex, \ldots, disp_{n-1} + ex, \\ & \ldots, \\ & disp_0 + ex \cdot (\texttt{count} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{count} - 1) \quad \}\end{aligned}$$

```
Datatype Datatype.vector(int count, int blocklength, int stride)
```

`count`	number of blocks
`blocklength`	number of elements in each block
`stride`	number of elements between start of each block
returns:	new datatype

Construct new datatype representing replication of this datatype into locations that consist of equally spaced blocks. Java binding of the MPI operation *MPI_TYPE_VECTOR*. The base type of the new datatype is the same as the base type of this type. Assume the displacement sequence of this type is

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent ex. Let `bl` be `blocklength`. Then the new datatype has a displacement sequence with `count` $\cdot$ `bl` $\cdot$ `n` entries defined by:

$$\begin{aligned}\{ \quad & disp_0, \ldots, disp_{n-1}, \\ & disp_0 + ex, \ldots, disp_{n-1} + ex, \\ & \ldots, \\ & disp_0 + ex \cdot (\texttt{bl} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{bl} - 1), \\ \\ & disp_0 + ex \cdot \texttt{stride}, \ldots, disp_{n-1} + ex \cdot \texttt{stride}, \\ & disp_0 + ex \cdot (\texttt{stride} + 1), \ldots, disp_{n-1} + ex \cdot (\texttt{stride} + 1), \\ & \ldots, \\ & disp_0 + ex \cdot (\texttt{stride} + \texttt{bl} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{stride} + \texttt{bl} - 1),\end{aligned}$$

$\ldots,$

$$
\begin{aligned}
& disp_0 + ex \cdot \texttt{stride} \cdot (\texttt{count} - 1), \ldots, disp_{n-1} + ex \cdot \texttt{stride} \cdot (\texttt{count} - 1), \\
& disp_0 + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + 1), \ldots, \\
& \qquad disp_{n-1} + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + 1), \\
& \ldots, \\
& disp_0 + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + \texttt{bl} - 1), \ldots, \\
& \qquad disp_{n-1} + ex \cdot (\texttt{stride} \cdot (\texttt{count} - 1) + \texttt{bl} - 1) \quad \}
\end{aligned}
$$

```
Datatype Datatype.hvector(int count, int blocklength, int stride)
```

`count`	number of blocks
`blocklength`	number of elements in each block
`stride`	number of elements between start of each block
returns:	new datatype

Identical to `vector` except that the stride is expressed directly in terms of the buffer index, rather than the units of this type. Java binding of the MPI operation *MPI_TYPE_HVECTOR. Unlike other language bindings*, the value of `stride` is *not* measured in bytes. The displacement sequence of the new type is:

$$
\begin{aligned}
\{ \quad & disp_0, \ldots, disp_{n-1}, \\
& disp_0 + ex, \ldots, disp_{n-1} + ex, \\
& \ldots, \\
& disp_0 + ex \cdot (\texttt{bl} - 1), \ldots, disp_{n-1} + ex \cdot (\texttt{bl} - 1), \\
& \\
& disp_0 + ex \cdot \texttt{stride}, \ldots, disp_{n-1} + \texttt{stride}, \\
& disp_0 + \texttt{stride} + ex, \ldots, disp_{n-1} + \texttt{stride} + ex, \\
& \ldots, \\
& disp_0 + \texttt{stride} + ex \cdot (\texttt{bl} - 1), \ldots, disp_{n-1} + \texttt{stride} + ex \cdot (\texttt{bl} - 1), \\
& \\
& \ldots, \\
& \\
& disp_0 + \texttt{stride} \cdot (\texttt{count} - 1), \ldots, disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1), \\
& disp_0 + \texttt{stride} \cdot (\texttt{count} - 1) + ex, \ldots,
\end{aligned}
$$

$$disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1) + ex,$$
$$\ldots,$$
$$disp_0 + \texttt{stride} \cdot (\texttt{count} - 1) + ex \cdot (\texttt{bl} - 1), \ldots,$$
$$disp_{n-1} + \texttt{stride} \cdot (\texttt{count} - 1) + ex \cdot (\texttt{bl} - 1) \quad \}$$

```
Datatype Datatype.indexed(int [] arrayOfBlocklengths,
                          int [] arrayOfDisplacements)
```

`arrayOfBlocklengths`	number of elements per block
`arrayOfDisplacements`	displacement of each block in units of old type
returns:	new datatype

Construct new datatype representing replication of this type into a sequence of blocks where each block can contain a different number of copies and have a different displacement. Java binding of the MPI operation *MPI_TYPE_INDEXED*. The number of blocks is taken to be size of the `arrayOfBlocklengths` argument. The second argument, `arrayOfDisplacements`, should be the same size. The base type of the new datatype is the same as the base type of this type. Assume the displacement sequence of this type is

$$\{disp_0, \ldots, disp_{n-1}\}$$

with extent ex. Let `B` be the `arrayOfBlocklengths` argument and `D` be the `arrayOfDisplacements` argument. Then the new datatype has a displacement sequence with $n \cdot \sum_{i=0}^{count-1} \texttt{B}[i]$ entries:

$$\begin{aligned}
\{ \quad & disp_0 + \texttt{D}[0] \cdot ex, \ldots, disp_{n-1} + \texttt{D}[0] \cdot ex, \\
& disp_0 + (\texttt{D}[0] + 1) \cdot ex, \ldots, disp_{n-1} + (\texttt{D}[0] + 1) \cdot ex, \\
& \ldots, \\
& disp_0 + (\texttt{D}[0] + \texttt{B}[0] - 1) \cdot ex, \ldots, disp_{n-1} + (\texttt{D}[0] + \texttt{B}[0] - 1) \cdot ex, \\
& \\
& \ldots, \\
& \\
& disp_0 + \texttt{D}[count - 1] \cdot ex, \ldots, disp_{n-1} + \texttt{D}[count - 1] \cdot ex, \\
& disp_0 + (\texttt{D}[count - 1] + 1) \cdot ex, \ldots, disp_{n-1} + (\texttt{D}[count - 1] + 1) \cdot ex, \\
& \ldots, \\
& disp_0 + (\texttt{D}[count - 1] + \texttt{B}[count - 1] - 1) \cdot ex, \ldots, \\
& \qquad disp_{n-1} + (\texttt{D}[count - 1] + \texttt{B}[count - 1] - 1) \cdot ex \quad \}
\end{aligned}$$

Here, *count* is the number of blocks.

```
Datatype Datatype.hindexed(int [] array_of_blocklengths,
                           int [] array_of_displacements)
```

`arrayOfBlocklengths`	number of elements per block
`arrayOfDisplacements`	displacement in buffer for each block
returns:	new datatype

Identical to `indexed` except that the displacements are expressed directly in terms of the buffer index, rather than the units of this type. Java binding of the MPI operation *MPI_TYPE_HINDEXED. Unlike other language bindings*, the values in `arrayOfDisplacements` are *not* measured in bytes. The displacement sequence of the new type is:

$$\begin{aligned}
\{ \quad & disp_0 + \mathtt{D}[0], \ldots, disp_{n-1} + \mathtt{D}[0], \\
& disp_0 + \mathtt{D}[0] + ex, \ldots, disp_{n-1} + \mathtt{D}[0] + ex, \\
& \ldots, \\
& disp_0 + \mathtt{D}[0] + (\mathtt{B}[0] - 1) \cdot ex, \ldots, disp_{n-1} + \mathtt{D}[0] + (\mathtt{B}[0] - 1) \cdot ex, \\
& \\
& \ldots, \\
& \\
& disp_0 + \mathtt{D}[count - 1], \ldots, disp_{n-1} + \mathtt{D}[count - 1], \\
& disp_0 + \mathtt{D}[count - 1] + ex, \ldots, disp_{n-1} + \mathtt{D}[count - 1] + ex, \\
& \ldots, \\
& disp_0 + \mathtt{D}[count - 1] + (\mathtt{B}[count - 1] - 1) \cdot ex, \ldots, \\
& \qquad disp_{n-1} + \mathtt{D}[count - 1] + (\mathtt{B}[count - 1] - 1) \cdot ex \quad \}
\end{aligned}$$

```
static Datatype Datatype.struct(int [] arrayOfBlocklengths,
                                int [] arrayOfDisplacements,
                                Datatype [] arrayOfTypes)
```

`arrayOfBlocklengths`	number of elements per block
`arrayOfDisplacements`	displacement in buffer for each block
`arrayOfTypes`	type of elements in each block
returns:	new datatype

The most general type constructor. Java binding of the MPI operation *MPI_TYPE_STRUCT*. The number of blocks is taken to be size of the `arrayOfBlocklengths` argument. The second and third arguments, `arrayOfDisplacements` and `arrayOfTypes`, should be the same size. *Unlike other language bindings*, the values in `arrayOfDisplacements` are *not* measured in bytes. All elements of `arrayOfTypes` with definite base types *must have the* same *base type*: this will be the base type of new datatype. Let `T` be the `arrayOfTypes` argument. Assume the displacement sequence of the old type $\mathtt{T}[i]$ is

$$\{disp_0^i, \ldots, disp_{n_i-1}^i\}$$

with extent ex_i. Let `B` be the `arrayOfBlocklengths` argument and `D` be the `arrayOfDisplacements` argument. Then the new datatype has a displacement sequence with $\sum_{i=0}^{c-1} \mathtt{B}[i] \cdot n_i$ entries:

$$\begin{aligned}
\{ \quad & disp_0^0 + \mathtt{D}[0], \ldots, disp_{n_0-1}^0 + \mathtt{D}[0], \\
& disp_0^0 + \mathtt{D}[0] + ex_0, \ldots, disp_{n_0-1}^0 + \mathtt{D}[0] + ex_0, \\
& \ldots, \\
& disp_0^0 + \mathtt{D}[0] + (\mathtt{B}[0]-1) \cdot ex_0, \ldots, disp_{n_0-1}^0 + \mathtt{D}[0] + (\mathtt{B}[0]-1) \cdot ex_0, \\
& \\
& \ldots, \\
& \\
& disp_0^{c-1} + \mathtt{D}[c-1], \ldots, disp_{n_{c-1}-1}^{c-1} + \mathtt{D}[c-1], \\
& disp_0^{c-1} + \mathtt{D}[c-1] + ex_{c-1}, \ldots, disp_{n_{c-1}-1}^{c-1} + \mathtt{D}[c-1] + ex_{c-1}, \\
& \ldots, \\
& disp_0^{c-1} + \mathtt{D}[c-1] + (\mathtt{B}[c-1]-1) \cdot ex_{c-1}, \ldots, \\
& \qquad\qquad disp_{n_{c-1}-1}^{c-1} + \mathtt{D}[c-1] + (\mathtt{B}[c-1]-1) \cdot ex_{c-1} \quad \}
\end{aligned}$$

Here, c is the number of blocks.

If any elements of `arrayOfTypes` are `MPI.LB` or `MPI.UB`, the corresponding displacements are omitted in the displacement sequence. These displacements only affect the computation of `Datatype.lb`, `Datatype.ub` and `Datatype.extent`, as explained below.

`int Datatype.extent()`

returns: datatype extent

Returns the extent of a datatype. Java binding of the MPI operation *MPI_TYPE_EXTENT*. Return value is equal to

$$\mathtt{ub()} - \mathtt{lb()}$$

`int Datatype.size()`

returns: datatype size

Returns the total size of the type. Java binding of the MPI operation *MPI_-TYPE_SIZE*. Size is defined as the total number of buffer elements incorporated by the data type, or equivalently as the length of the displacement sequence. *Unlike other language bindings*, the size is *not* measured in bytes.

`int Datatype.lb()`

returns: displacement of lower bound from origin

Find the lower bound of a datatype. Java binding of the MPI operation *MPI_-TYPE_LB*. Assume the displacement sequence of the datatype is

$$\{disp_0, \ldots, disp_{n-1}\}$$

In general the return value of `lb` is

$$\min_j disp_j$$

However, if construction of the derived type involved any basic components of type `MPI.LB`, the return value is the least displacement specified for any of those `MPI.LB` fields, regardless of the values appearing in the displacement list proper.

`int Datatype.ub()`

returns: displacement of upper bound from origin

Find the upper bound of a datatype. Java binding of the MPI operation *MPI_-TYPE_UB*. Assume the displacement sequence of the datatype is

$$\{disp_0, \ldots, disp_{n-1}\}$$

In general the return value of `lb` is

$$\max_j (disp_j + 1)$$

However, if construction of the derived type involved any basic components of type `MPI.UB`, the return value is the largest displacement specified for any of those `MPI.UB` fields, regardless of the values appearing in the displacement list proper.

> *Advice to implementors.* Meeting the detailed requirements of the MPI standard regarding handling of `MPI.UB` and `MPI.LB` needs some extra

care. A possible implementation is to maintain two separate "pseudo-displacements" whose values are undefined unless a `MPI.UB` (respectively `MPI.LB`) has appeared somewhere. If defined, these extra displacements are propagated to derived types using the same formulae as for true displacements, except that only the greatest (respectively least) term contributed by any precursor pseudo-displacement need be retained as the pseudo-displacement for the new type. *(End of advice to implementors.)*

```
void Datatype.commit()
```

Commit a derived datatype. Java binding of the MPI operation *MPI_TYPE_COMMIT*.

```
void Datatype.finalize()
```

Destructor. Java binding of the MPI operation *MPI_TYPE_FREE*.

```
int Status.getElements(Datatype datatype)
```

`datatype`	datatype used by receive operation
returns:	number of received basic elements

Retrieve number of basic elements from status. Java binding of the MPI operation *MPI_GET_ELEMENTS*.

A.3.13 Pack and unpack

```
int Comm.pack(Object inbuf, int offset, int incount,
              Datatype datatype,
              byte [] outbuf, int position)
```

`inbuf`	input buffer array
`offset`	initial offset in input buffer
`incount`	number of items in input buffer
`datatype`	datatype of each item in input buffer
`outbuf`	output buffer
`position`	initial position in ouput buffer
returns:	final position in output buffer

Packs message in send buffer `inbuf` into space specified in `outbuf`. Java binding of the MPI operation *MPI_PACK*. The return value is the output value of `position`—the inital value incremented by the number of bytes written.

```
byte[] Comm.pack(Object inbuf, int offset, int incount,
                 Datatype datatype)
```

`inbuf`	input buffer array
`offset`	initial offset in input buffer
`incount`	number of items in input buffer
`datatype`	datatype of each item in input buffer
returns:	output buffer

Packs message in send buffer `inbuf` into buffer space allocated internally.

> *Rationale.* This variant of `pack` is essentially equivalent to calling *MPI_-PACK_SIZE* to find the amount of space needed for a packed message, allocating a suitable byte vector, then calling the previous variant of `pack`. The object serialization mechanisms of Java do not provide a convenient way of precomputing the amount of space required to pack objects (there is no analogue to *MPI_PACK_SIZE*). So if the base type of the message is `MPI.OBJECT` the second form of `pack` is more convenient. *(End of rationale.)*

```
int Comm.unpack(byte [] inbuf, int position,
                Object outbuf, int offset, int outcount,
                Datatype datatype)
```

`inbuf`	input buffer
`position`	initial position in input buffer
`outbuf`	output buffer array
`offset`	initial offset in output buffer
`outcount`	number of items in output buffer
`datatype`	datatype of each item in output buffer
returns:	final position in input buffer

Unpacks message in receive buffer `outbuf` into space specified in `inbuf`. Java binding of the MPI operation *MPI_UNPACK*. The return value is the output value of *position*—the inital value incremented by the number of bytes read.

```
int Comm.packSize(int incount, Datatype datatype)
```

`incount`	number of items in input buffer
`datatype`	datatype of each item in input buffer
returns:	upper bound on size of packed message

Returns an upper bound on the increment of `position` effected by `pack`. Java binding of the MPI operation *MPI_PACK_SIZE. It is an error to call this function if the base type of* `datatype` *is* `MPI.OBJECT`.

A.4 Collective Communication

A.4.1 Introduction and Overview

In general the Java binding of collective communication operations will realize the MPI functions as members of the `IntraComm` class.

A.4.2 Communicator argument

No special issues for Java binding.

A.4.3 Barrier synchronization

```
void Intracomm.barrier()
```

A call to `barrier` blocks the caller until all processes in the group have called it. Java binding of the MPI operation *MPI_BARRIER*.

A.4.4 Broadcast

```
void Intracomm.bcast(Object buffer, int offset, int count,
                     Datatype datatype, int root)
```

`buf`	buffer array
`offset`	initial offset in buffer
`count`	number of items in buffer
`datatype`	datatype of each item in buffer
`dest`	rank of broadcast root

Broadcast a message from the process with rank `root` to all processes of the group. Java binding of the MPI operation *MPI_BCAST*.

A.4.5 Gather

```
void Intracomm.gather(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int recvcount, Datatype recvtype, int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of each item in receive buffer
`root`	rank of receiving process

Each process sends the contents of its send buffer to the root process. Java binding of the MPI operation *MPI_GATHER*.

```
void Intracomm.gatherv(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int [] recvcounts, int [] displs,
                       Datatype recvtype, int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	number of elements received from each process
`displs`	displacements at which to place incoming data
`recvtype`	datatype of each item in receive buffer
`root`	rank of receiving process

Extends functionality of `gather` by allowing varying counts of data from each process. Java binding of the MPI operation *MPI_GATHERV*. The sizes of arrays `recvcounts` and `displs` should be the size of the group. Entry i of `displs` specifies the displacement relative to element `recvoffset` of `recvbuf` at which to place incoming data. Note that if `recvtype` is a derived data type, elements of `displs` are in units of the derived type extent, (unlike `recvoffset`, which is a direct index into the buffer array).

A.4.6 Scatter

```
void Intracomm.scatter(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
```

```
                        int recvcount, Datatype recvtype,
                        int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items sent to each process
`sendtype`	datatype of send buffer items
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of receive buffer items
`root`	rank of sending process

Inverse of the operation `gather`. Java binding of the MPI operation *MPI_SCATTER*.

```
void Intracomm.scatterv(Object sendbuf, int sendoffset,
                         int [] sendcounts, int [] displs,
                         Datatype sendtype,
                         Object recvbuf, int recvoffset,
                         int recvcount, Datatype recvtype,
                         int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcounts`	number of items sent to each process
`displs`	displacements from which to take outgoing data
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of elements in receive buffer
`recvtype`	datatype of receive buffer items
`root`	rank of sending process

Inverse of the operation `gatherv`. Java binding of the MPI operation *MPI_SCATTERV*.

A.4.7 Gather-to-all

```
void Intracomm.allgather(Object sendbuf, int sendoffset,
                          int sendcount, Datatype sendtype,
                          Object recvbuf, int recvoffset,
                          int recvcount, Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items sent to each process
`sendtype`	datatype of send buffer items
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items in receive buffer
`recvtype`	datatype of receive buffer items

Similar to `gather`, but all processes receive the result. Java binding of the MPI operation *MPI_ALLGATHER*.

```
void Intracomm.allgatherv(Object sendbuf, int sendoffset,
                          int sendcount, Datatype sendtype,
                          Object recvbuf, int recvoffset,
                          int [] recvcounts, int [] displs,
                          Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items to send
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	number of elements received from each process
`displs`	displacements at which to place incoming data
`recvtype`	datatype of each item in receive buffer

Similar to `gatherv`, but all processes receive the result. Java binding of the MPI operation *MPI_GATHERV*.

A.4.8 All-to-All Scatter/Gather

```
void Intracomm.alltoall(Object sendbuf, int sendoffset,
                        int sendcount, Datatype sendtype,
                        Object recvbuf, int recvoffset,
                        recvcount, Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcount`	number of items sent to each process
`sendtype`	datatype of send buffer items
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcount`	number of items received from any process
`recvtype`	datatype of receive buffer items

Extension of `allgather` to the case where each process sends distinct data to each of the receivers. Java binding of the MPI operation *MPI_ALLTOALL*.

```
void Intracomm.alltoallv(Object sendbuf, int sendoffset,
                         int [] sendcount, int [] sdispls,
                         Datatype sendtype,
                         Object recvbuf, int recvoffset,
                         int [] recvcount, int [] rdispls,
                         Datatype recvtype)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`sendcounts`	number of items sent to each process
`sdispls`	displacements from which to take outgoing data
`sendtype`	datatype of each item in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	number of elements received from each process
`rdispls`	displacements at which to place incoming data
`recvtype`	datatype of each item in receive buffer

Adds flexibility to `alltoall`: location of data for send is specified by `sdispls` and location to place data on receive side is specified by `rdispls`. Java binding of the MPI operation *MPI_ALLTOALLV*.

A.4.9 Global Reduction Operations

```
void Intracomm.reduce(Object sendbuf, int sendoffset,
                      Object recvbuf, int recvoffset,
                      int count, Datatype datatype,
                      Op op, int root)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`count`	number of items in send buffer
`datatype`	data type of each item in send buffer
`op`	reduce operation
`dest`	rank of root process

Combine elements in input buffer of each process using the reduce operation, and return the combined value in the output buffer of the root process. Java binding of the MPI operation *MPI_REDUCE*.

The predefined operations are available in Java as `MPI.MAX`, `MPI.MIN`, `MPI.-SUM`, `MPI.PROD`, `MPI.LAND`, `MPI.BAND`, `MPI.LOR`, `MPI.BOR`, `MPI.LXOR`, `MPI.BXOR`, `MPI.MINLOC` and `MPI.MAXLOC`.

The handling of *MINLOC* and *MAXLOC* is modelled on the Fortran binding. The extra predefined types `MPI.SHORT2`, `MPI.INT2`, `MPI.LONG2`, `MPI.FLOAT2`, `MPI.DOUBLE2` describe pairs of Java numeric primitive types.

`Op.Op(UserFunction function, boolean commute)`

`function`	user defined function
`commute`	`true` if commutative, `false` otherwise

Bind a user-defined global reduction operation to an `Op` object. Java binding of the MPI operation *MPI_OP_CREATE*. The interface `UserFunction` is defined by

```
interface UserFunction {
  public abstract void call(Object invec, int inoffset,
                            Object inoutvec, int inoutoffset,
                            int count, Datatype datatype) ;
}
```

To define a new operation, the programmer should define a class that implements the `call` method of `UserFunction`, then pass an object from this class to the `Op` constructor. The `UserFunction.call` method plays exactly the same role as the `function` argument in the standard bindings of MPI. The actual arguments `invec` and `outvec` passed to `call` will be arrays with `count` elements of the type specified in the `datatype` argument. The user-defined `call` method combines these in the way specified in the MPI standard.

`void Op.finalize()`

Destructor. Java binding of the MPI operation *MPI_OP_FREE*.

```
void Intracomm.allreduce(Object sendbuf, int sendoffset,
                         Object recvbuf, int recvoffset,
                         int count, Datatype datatype,
                         Op op)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`count`	number of items in send buffer
`datatype`	data type of each item in send buffer
`op`	reduce operation

Same as `reduce` except that the result appears in receive buffer of all process in the group. Java binding of the MPI operation *MPI_ALLREDUCE*.

A.4.10 Reduce-Scatter

```
void Intracomm.reduceScatter(Object sendbuf, int sendoffset,
                             Object recvbuf, int recvoffset,
                             int [] recvcounts,
                             Datatype datatype, Op op)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`recvcounts`	numbers of result elements distributed to each process
`datatype`	data type of each item in send buffer
`op`	reduce operation

Combine elements in input buffer of each process using the reduce operation, and scatter the combined values over the output buffers of the processes. Java binding of the MPI operation *MPI_REDUCE_SCATTER*.

A.4.11 Scan

```
void Intracomm.scan(Object sendbuf, int sendoffset,
                    Object recvbuf, int recvoffset,
                    int count, Datatype datatype,
                    Op op)
```

`sendbuf`	send buffer array
`sendoffset`	initial offset in send buffer
`recvbuf`	receive buffer array
`recvoffset`	initial offset in receive buffer
`count`	number of items in input buffer
`datatype`	data type of each item in input buffer
`op`	reduce operation

Perform a prefix reduction on data distributed across the group. Java binding of the MPI operation *MPI_SCAN*.

A.4.12 Correctness

No special issues for Java binding.

A.5 Groups, Contexts and Communicators

A.5.1 Introduction

No special issues for Java binding.

A.5.2 Basic Concepts

The constant *MPI_GROUP_EMPTY* is available as `MPI.GROUP_EMPTY`. The constants *MPI_COMM_WORLD*, *MPI_COMM_SELF* are available as `MPI.COMM_WORLD`, `MPI.COMM_SELF`.

A.5.3 Group Management

`int Group.size()`

returns: number of processors in the group

Size of group. Java binding of the MPI operation *MPI_GROUP_SIZE*.

`int Group.rank()`

returns: rank of the calling process in the group

Rank of this process in group. Java binding of the MPI operation *MPI_GROUP_RANK*. Result value is `MPI.UNDEFINED` if this process is not a member of the group.

`int [] Group.translateRanks(Group group1, int [] ranks1)`

`group1`	another group
`ranks1`	array of valid ranks in `group1`

returns: array of corresponding ranks in this group

Translate ranks within another group to ranks within this group. Java binding of the MPI operation *MPI_GROUP_TRANSLATE_RANKS*. Result elements are `MPI.UNDEFINED` where no correspondence exists.

`static int Group.compare(Group group1, Group group2)`

`group1` first group
`group2` second group

returns: result

Compare two groups. Java binding of the MPI operation *MPI_GROUP_COMPARE*. `MPI.IDENT` results if the group members and group order are exactly the same in both groups. `MPI.SIMILAR` results if the group members are the same but the order is different. `MPI.UNEQUAL` results otherwise.

`Group Comm.group()`

returns: group corresponding to this communicator

Return group associated with a communicator. Java binding of the MPI operation *MPI_COMM_GROUP*.

`static Group Group.union(Group group1, Group group2)`

`group1` first group
`group2` second group

returns: union group

Set union of two groups. Java binding of the MPI operation *MPI_GROUP_UNION*.

`static Group Group.intersection(Group group1, Group group2)`

`group1` first group
`group2` second group

returns: intersection group

Set intersection of two groups. Java binding of the MPI operation *MPI_GROUP_INTERSECTION*.

`static Group Group.difference(Group group1, Group group2)`

`group1` first group
`group2` second group

returns: difference group

Result contains all elements of the first group that are not in the second group. Java binding of the MPI operation *MPI_GROUP_DIFFERENCE*.

```
Group Group.incl(int [] ranks)
```

`ranks`	ranks from this group to appear in new group
returns:	new group

Create a subset group including specified processes. Java binding of the MPI operation *MPI_GROUP_INCL*.

```
Group Group.excl(int [] ranks)
```

`ranks`	ranks from this group *not* to appear in new group
returns:	new group

Create a subset group excluding specified processes. Java binding of the MPI operation *MPI_GROUP_EXCL*.

```
Group Group.rangeIncl(int [] [] ranges)
```

`ranges`	array of integer triplets
returns:	new group

Create a subset group including processes specified by strided intervals of ranks. Java binding of the MPI operation *MPI_GROUP_RANGE_INCL*. The triplets are of the form (first rank, last rank, stride) indicating ranks in this group to be included in the new group. The size of the first dimension of `ranges` is the number of triplets. The size of the second dimension is 3.

```
Group Group.rangeExcl(int [] [] ranges)
```

`ranges`	array of integer triplets
returns:	new group

Create a subset group excluding processes specified by strided intervals of ranks. Java binding of the MPI operation *MPI_GROUP_RANGE_EXCL*. Triplet array is defined as for `rangeIncl`, the ranges indicating ranks in this group to be excluded from the new group.

`void Group.finalize()`

Destructor. Java binding of the MPI operation *MPI_GROUP_FREE*.

A.5.4 Communicator Management

`int Comm.size()`

returns:	number of processors in the group of this communicator

Size of group of this communicator. Java binding of the MPI operation *MPI_COMM_SIZE*.

`int Comm.rank()`

returns:	rank of the calling process in the group of this communicator

Rank of this process in group of this communicator. Java binding of the MPI operation *MPI_COMM_RANK*.

`static int Comm.compare(Comm comm1, Comm comm2)`

`comm1`	first communicator
`comm2`	second communicator

returns:	result

Compare two communicators. Java binding of the MPI operation *MPI_COMM_COMPARE*. `MPI.IDENT` results if the `comm1` and `comm2` are references to the same object (ie, if `comm1 == comm2`). `MPI.CONGRUENT` results if the underlying groups are identical but the communicators differ by context. `MPI.SIMILAR` results if the underlying groups are similar but the communicators differ by context. `MPI.UNEQUAL` results otherwise.

`Object Comm.clone()`

returns:	copy of this communicator

Duplicate this communicator. Java binding of the MPI operation *MPI_COMM_DUP*. The new communicator is "congruent" to the old one, but has a different context.

Rationale. The decision to use the standard Java `clone` method means the static result type must be `Object`. The dynamic type will be that of the `Comm` subclass of the parent. MPI-defined and user-defined subclasses of `Comm` will generally override `clone` to ensure all relevant attributes are copied. *(End of rationale.)*

`Intracomm Intracomm.create(Group group)`

`group` group which is a subset of the group of this communicator

returns: new communicator

Create a new communicator. Java binding of the MPI operation *MPI_COMM_-CREATE*.

`Intracomm Intracomm.split(int color, int key)`

`color` control of subset assignment
`key` control of rank assignment

returns: new communicator

Partition the group associated with this communicator and create a new communicator within each subgroup. Java binding of the MPI operation *MPI_COMM_-SPLIT*.

`void Comm.free()`

Destroy this communicator. Java binding of the MPI operation *MPI_COMM_-FREE*.

Rationale. An explicitly called `free` method is required rather than an implicitly called `finalize` method, because *MPI_COMM_FREE* is a collective operations. We cannot assume that the Java garbage collector will call a `finalize` method synchronously on all processors. *(End of rationale.)*

A.5.5 Motivating Examples

No special issues for Java binding.

A.5.6 Inter-Communication

```
boolean Comm.testInter()
```

returns: `true` if this is an inter-communicator, `false` otherwise

Test if this communicator is an inter-communicator. Java binding of the MPI operation *MPI_COMM_TEST_INTER*.

```
int Intercomm.remoteSize()
```

returns: number of process in remote group of this communicator

Size of remote group. Java binding of the MPI operation *MPI_COMM_REMOTE_SIZE*.

```
Group Intercomm.remoteGroup()
```

returns: remote group of this communicator

Return the remote group. Java binding of the MPI operation *MPI_COMM_REMOTE_GROUP*.

```
Intercomm Comm.createIntercomm(Comm localComm, int localLeader,
                               int remoteLeader, int tag)
```

`localComm`	local intra-communicator
`localLeader`	rank of local group leader in `localComm`
`remoteLeader`	rank of remote group leader in this communictor
`tag`	"safe" tag

returns: new inter-communicator

Create an inter-communicator. Java binding of the MPI operation *MPI_INTERCOMM_CREATE*.

> *Rationale.* This operation is defined as a method on the "peer communicator", making it analogous to a `send` or `recv` communication with the remote group leader. *(End of rationale.)*

```
Intracomm Intercomm.merge(boolean high)
```

`high` true if the local group has higher ranks in combined group

returns: new intra-communicator

Create an intra-communicator from the union of the two groups in the inter-communicator. Java binding of the MPI operation *MPI_INTERCOMM_MERGE*.

A.5.7 Caching

It is assumed that, to achieve the effect of caching attributes in user-customized communicators, programmers will create subclasses of the library-defined communicator classes with suitable additional fields. These fields may be copied or deleted by suitably overridden `clone` and `finalize` methods.

Hence the only "caching" operation surviving here is the binding of *MPI_ATTR_GET*, which is needed to access values of attributes predefined by the implementation. According the standard, the key values for such attributes include `MPI.TAG_UB`, `MPI.HOST`, `MPI.IO` and `MPI.WTIME_IS_GLOBAL`.

`Object Comm.attrGet(int keyval)`

`keyval` one of the key values predefined by the implementation

returns: attribute value

Retrieves attribute value by key. Java binding of the MPI operation *MPI_ATTR_GET*.

> *Advice to implementors.* If the attribute value is naturally represented by primitive type such as an `int`, this function may wrap the return value as an object such as an `Integer`. *(End of advice to implementors.)*

A.6 Process Topologies

A.6.1 Introduction

Communicators with Cartesian or graph topologies will be realized as instances of the subclasses `Cartcomm` or `Graphcomm`, respectively of `Intracomm`.

A.6.2 Virtual Topologies

No special issues for Java binding.

A.6.3 Embedding in MPI

No special issues for Java binding.

A.6.4 Overview of the Functions

No special issues for Java binding.

A.6.5 Topology Constructors

```
Cartcomm Intracomm.createCart(int [] dims, boolean [] periods,
                              boolean reorder)
```

`dims`	the number of processes in each dimension
`periods`	`true` if grid is periodic, `false` if not, in each dimension
`reorder`	`true` if ranking may be reordered, `false` if not
returns:	new Cartesian topology communicator

Create a Cartesian topology communicator whose group is a subset of the group of this communicator. Java binding of the MPI operation *MPI_CART_CREATE*. The number of dimensions of the Cartesian grid is taken to be the size of the `dims` argument. The array `periods` must be the same size.

```
static Cartcomm.dimsCreate(int nnodes, int [] dims)
```

`nnodes`	number of nodes in a grid
`dims`	array specifying the number of nodes in each dimension

Select a balanced distribution of processes per coordinate direction. Java binding of the MPI operation *MPI_DIMS_CREATE*. Number of dimensions is the size of is `dims`. Note that `dims` is an *inout* parameter.

```
Graphcomm Intracomm.createGraph(int [] index, int [] edges,
                                boolean reorder)
```

`index`	node degrees
`edges`	graph edges
`reorder`	`true` if ranking may be reordered, `false` if not
returns:	new graph topology communicator

Create a graph topology communicator whose group is a subset of the group of this communicator. Java binding of the MPI operation *MPI_GRAPH_CREATE*. The number of nodes in the graph, *nnodes*, is taken to be size of the `index` argument. The size of array `edges` must be `index`$[nnodes-1]$.

```
int Comm.topoTest()
```

returns: topology type of communicator

Returns the type of topology associated with the communicator. Java binding of the MPI operation *MPI_TOPO_TEST*. The return value will be one of `MPI.GRAPH`, `MPI.CART` or `MPI.UNDEFINED`.

```
GraphParms Graphcomm.get()
```

returns: object defining node degress and edges of graph

Returns graph topology information. Java binding of the MPI operations *MPI_GRAPHDIMS_GET* and *MPI_GRAPH_GET*. The class of the returned object is

```
public class GraphParms {
  public int [] index ;
  public int [] edges ;
}
```

The number of nodes and number of edges can be extracted from the sizes of the `index` and `edges` arrays.

```
CartParms Cartcomm.get()
```

returns: object containing dimensions, periods and local coordinates

Returns Cartesian topology information. Java binding of the MPI operations *MPI_CARTDIM_GET* and *MPI_CART_GET*. The class of the returned object is

```
public class CartParms {
  public int [] dims ;
  public booleans [] periods ;
  public int [] coords ;
}
```

The number of dimensions can be obtained from the size of (eg) the `dims` array.

> *Rationale.* The inquiries *MPI_GRAPHDIMS_GET*, *MPI_GRAPH_GET*, *MPI_CARTDIM_GET*, and *MPI_CART_GET* are unusual in returning multiple independent values from single calls. This is a problem in Java. The Java binding could split these inquiries into several independent ones, but that would complicate JNI-based wrapper implementations. Hence we introduced the auxilliary classes `GraphParms` and `CartParms` to hold multiple results. *(End of rationale.)*

`int Cartcomm.rank(int [] coords)`

`coords`	Cartesian coordinates of a process
returns:	rank of the specified process

Translate logical process coordinates to process rank. Java binding of the MPI operation *MPI_CART_RANK*.

`int [] Cartcomm.coords(int rank)`

`coords`	rank of a process
returns:	Cartesian coordinates of the specified process

Translate process rank to logical process coordinates. Java binding of the MPI operation *MPI_CART_COORDS*.

`int [] Graphcomm.neighbours(int rank)`

`coords`	rank of a process in the group of this communicator
returns:	array of ranks of neighbouring processes to one specified

Provides adjacency information for general graph topology. Java binding of the MPI operations *MPI_GRAPH_NEIGHBOURS_COUNT* and *MPI_GRAPH_NEIGHBOURS*. The number of neighbours can be extracted from the size of the result.

`ShiftParms Cartcomm.shift(int direction, int disp)`

`direction`	coordinate dimension of shift
`disp`	displacement
returns:	object containing ranks of source and destination processes

Compute source and destination ranks for "shift" communication. Java binding of the MPI operation *MPI_CART_SHIFT*. The class of the returned object is

```
public class ShiftParms {
  public int rankSource ;
  public int rankDest ;
}
```

`Cartcomm Cartcomm.sub(boolean [] remainDims)`

`remainDims`	by dimension, `true` if dimension is to be kept, `false` otherwise
returns:	communicator containing subgrid including this process

Partition Cartesian communicator into subgroups of lower dimension. Java binding of the MPI operation *MPI_CART_SUB*.

`int Cartcomm.map(int [] dims, boolean [] periods)`

`dims`	the number of processes in each dimension
`periods`	`true` if grid is periodic, `false` if not, in each dimension
returns:	reordered rank of calling process

Compute an optimal placement. Java binding of the MPI operation *MPI_CART_MAP*. The number of dimensions is taken to be size of the `dims` argument.

`int Graphcomm.map(int [] index, int [] edges)`

`index`	node degrees
`edges`	graph edges
returns:	reordered rank of calling process

Compute an optimal placement. Java binding of the MPI operation *MPI_GRAPH_MAP*. The number of nodes is taken to be size of the `index` argument.

A.7 MPI Environmental Management

A.7.1 Implementation information

The constants *MPI_TAG_UB*, *MPI_HOST* and *MPI_IO* are available as `MPI.TAG_UB`, `MPI.HOST`, `MPI.IO`.

`static String MPI.getProcessorName()`

returns: A unique specifier for the actual node.

Returns the name of the processor on which it is called. Java binding of the MPI operation *MPI_GET_PROCESSOR_NAME*.

A.7.2 Error handling

The constants *MPI_ERRORS_ARE_FATAL*, *MPI_ERRORS_RETURN* are available as `MPI.ERRORS_ARE_FATAL`, `MPI.ERRORS_RETURN`.

If the effective error handler is *MPI_ERRORS_RETURN*, the wrapper codes will throw appropriate Java exceptions (see section A.7.3).

For now, we omit a specification of Java interface for creating new MPI error handlers, because the detailed interface of the handler function depends on unstandardized features of the MPI implementation.

`static void Comm.errorhandlerSet(Errhandler errhandler)`

`errhandler` new MPI error handler for communicator

Associates a new error handler with communicator at the calling process. Java binding of the MPI operation *MPI_ERRORHANDLER_SET*.

`static Errhandler Comm.errorhandlerGet()`

returns: MPI error handler currently associated with communicator

Returns the error handler currently associated with the communicator. Java binding of the MPI operation *MPI_ERRORHANDLER_GET*.

A.7.3 Error codes and classes

The `Exception` subclasses

```
MPIErrBuffer
MPIErrCount
MPIErrType
MPIErrTag
MPIErrComm
MPIErrRank
MPIErrRequest
MPIErrRoot
MPIErrGroup
MPIErrOp
MPIErrTopology
MPIErrDims
MPIErrArg
MPIErrUnknown
MPIErrTruncate
MPIErrOther
MPIErrIntern
```

correspond to the standard MPI error classes.

A.7.4 Timers

`static double MPI.wtime()`

returns: elapsed wallclock time in seconds since some time in the past

Returns wallclock time. Java binding of the MPI operation *MPI_WTIME*.

`static double MPI.wtick()`

returns: resolution of `wtime` in seconds.

Returns resolution of timer. Java binding of the MPI operation *MPI_WTICK*.

A.7.5 Startup

`static void MPI.init(String[] argv)`

`argv` arguments to `main` method.

Initialize MPI. Java binding of the MPI operation *MPI_INIT*.

`static void MPI.finish()`

Finalize MPI. Java binding of the MPI operation *MPI_FINALIZE*.

`static boolean MPI.initialized()`

returns: `true` if `init` has been called, `false` otherwise.

Test if MPI has been initialized. Java binding of the MPI operation *MPI_INITIALIZED*.

`void Comm.abort(int errorcode)`

`errorcode` error code for Unix or POSIX environments

Abort MPI. Java binding of the MPI operation *MPI_ABORT*.

B Full public interface of classes

Section names appearing in comments refer to the preceding appendix. Specification of the methods immediately following those comments should be found in the referenced section.

(Notably missing from this draft are `throws` clauses for exceptions.)

B.1 MPI

```
public class MPI {
  public static Intracomm COMM_WORLD;

  public static Datatype BYTE, CHAR, SHORT, BOOLEAN, INT, LONG,
                         FLOAT, DOUBLE, PACKED, LB, UB ;

  public static int ANY_SOURCE, ANY_TAG ;

  public static int PROC_NULL ;

  public static int BSEND_OVERHEAD ;

  public static int UNDEFINED ;

  public static Op MAX, MIN, SUM, PROD, LAND, BAND,
                   LOR, BOR, LXOR, BXOR, MINLOC, MAXLOC ;

  public static Datatype SHORT2, INT2, LONG2, FLOAT2, DOUBLE2 ;

  public static Group GROUP_EMPTY ;

  public static Comm COMM_SELF ;

  public static int IDENT, CONGRUENT, SIMILAR, UNEQUAL ;

  public static int GRAPH, CART ;

  public static ErrHandler ERRORS_ARE_FATAL, ERRORS_RETURN ;

  public static int TAG_UB, HOST, IO ;


  // Buffer allocation and usage

  public static void bufferAttach(byte [] buffer) {...}

  public static byte [] bufferDetach() {...}


  // Environmental Management

  public static void init(String[] argv) {...}

  public static void finish() {...}

  public static String getProcessorName() {...}
```

```
  public static double wtime() {...}

  public static double wtick() {...}

  public static boolean initialized() {...}

  ...
}

public class Errhandler {
  ...
}
```

B.2 Comm

```
public class Comm {

  // Communicator Management

  public int size() {...}

  public int rank() {...}

  public Group group() {...}  // (see ``Group management'')

  public static int compare(Comm comm1, Comm comm2) {...}

  public Object clone() {...}

  public void free() {...}


  // Inter-communication

  public boolean testInter() {...}

  public Intercomm createIntercomm(Comm localComm, int localLeader,
                                   int remoteLeader, int tag) {...}


  // Caching

  public Object attrGet(int keyval) {...}


  // Blocking Send and Receive operations

  public void send(Object buf, int offset, int count,
                   Datatype datatype, int dest, int tag) {...}

  public Status recv(Object buf, int offset, int count,
                     Datatype datatype, int source, int tag) {...}


  // Communication Modes

  public void bsend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag) {...}

  public void ssend(Object buf, int offset, int count,
                    Datatype datatype, int dest, int tag) {...}
```

```
public void rsend(Object buf, int offset, int count,
                  Datatype datatype, int dest, int tag) {...}


// Nonblocking communication

public Request isend(Object buf, int offset, int count,
                     Datatype datatype, int dest, int tag) {...}

public Request ibsend(Object buf, int offset, int count,
                      Datatype datatype, int dest, int tag) {...}

public Request issend(Object buf, int offset, int count,
                      Datatype datatype, int dest, int tag) {...}

public Request irsend(Object buf, int offset, int count,
                      Datatype datatype, int dest, int tag) {...}

public Request irecv(Object buf, int offset, int count,
                     Datatype datatype, int source, int tag) {...}


// Probe and cancel

public Status iprobe(int source, int tag) {...}

public Status probe(int source, int tag) {...}


// Persistent communication requests

public Prequest sendInit(Object buf, int offset, int count,
                         Datatype datatype, int dest, int tag) {...}

public Prequest bsendInit(Object buf, int offset, int count,
                          Datatype datatype, int dest, int tag) {...}

public Prequest ssendInit(Object buf, int offset, int count,
                          Datatype datatype, int dest, int tag) {...}

public Prequest rsendInit(Object buf, int offset, int count,
                          Datatype datatype, int dest, int tag) {...}

public Prequest recvInit(Object buf, int offset, int count,
                         Datatype datatype, int source, int tag) {...}
```

```
  // Send-receive

  public Status sendrecv(Object sendbuf, int sendoffset,
                         int sendcount, Datatype sendtype,
                         int dest, int sendtag,
                         Object recvbuf, int recvoffset,
                         int recvcount, Datatype recvtype,
                         int source, int recvtag) {...}

  public Status sendrecvReplace(Object buf, int offset,
                                int count, Datatype datatype,
                                int dest, int sendtag,
                                int source, int recvtag) {...}

  // Pack and unpack

  public int pack(Object inbuf, int offset, int incount,
                  Datatype datatype,
                  byte [] outbuf, int position) {...}

  byte[] pack(Object inbuf, int offset, int incount,
              Datatype datatype) {...}

  public int unpack(byte [] inbuf, int position,
                    Object outbuf, int offset, int outcount,
                    Datatype datatype) {...}

  public int packSize(int incount, Datatype datatype) {...}


  // Process Topologies

  int topoTest()


  // Environmental Management

  public static void errorhandlerSet(Errhandler errhandler) {...}

  public static Errhandler errorhandlerGet() {...}

  void abort(int errorcode) {...}

  ...
}
```

B.3 `Intracomm` and `Intercomm`

```
public class Intracomm extends Comm {

  public Object clone() { ... }

  public Intracomm create(Group group) {...}

  public Intracomm split(int colour, int key) {...}

  // Collective communication

  public void barrier() {...}

  public void bcast(Object buffer, int offset, int count,
                    Datatype datatype, int root) {...}

  public void gather(Object sendbuf, int sendoffset,
                     int sendcount, Datatype sendtype,
                     Object recvbuf, int recvoffset,
                     int recvcount, Datatype recvtype, int root) {...}

  public void gatherv(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int [] recvcount, int [] displs,
                      Datatype recvtype, int root) {...}

  public void scatter(Object sendbuf, int sendoffset,
                      int sendcount, Datatype sendtype,
                      Object recvbuf, int recvoffset,
                      int recvcount, Datatype recvtype, int root) {...}

  public void scatterv(Object sendbuf, int sendoffset,
                       int [] sendcount, int [] displs,
                       Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int recvcount, Datatype recvtype, int root) {...}

  public void allgather(Object sendbuf, int sendoffset,
                        int sendcount, Datatype sendtype,
                        Object recvbuf, int recvoffset,
                        int recvcount, Datatype recvtype) {...}

  public void allgatherv(Object sendbuf, int sendoffset,
                         int sendcount, Datatype sendtype,
                         Object recvbuf, int recvoffset,
                         int [] recvcounts, int [] displs,
```

```
                         Datatype recvtype) {...}

  public void alltoall(Object sendbuf, int sendoffset,
                       int sendcount, Datatype sendtype,
                       Object recvbuf, int recvoffset,
                       int recvcount, Datatype recvtype) {...}

  public void alltoallv(Object sendbuf, int sendoffset,
                        int [] sendcount, int [] sdispls,
                        Datatype sendtype,
                        Object recvbuf, int recvoffset,
                        int [] recvcount, int [] rdispls,
                        Datatype recvtype) {...}

  public void reduce(Object sendbuf, int sendoffset,
                     Object recvbuf, int recvoffset,
                     int count, Datatype datatype,
                     Op op, int root) {...}

  public void allreduce(Object sendbuf, int sendoffset,
                        Object recvbuf, int recvoffset,
                        int count, Datatype datatype,
                        Op op) {...}

  public void reduceScatter(Object sendbuf, int sendoffset,
                            Object recvbuf, int recvoffset,
                            int [] recvcounts, Datatype datatype,
                            Op op) {...}

  public void scan(Object sendbuf, int sendoffset,
                   Object recvbuf, int recvoffset,
                   int count, Datatype datatype,
                   Op op) {...}


  // Topology Constructors

  public Graphcomm createGraph(int [] index, int [] edges,
                               boolean reorder) {...}

  public Cartcomm createCart(int [] dims, boolean [] periods,
                             boolean reorder) {...}

  ...
}

public class Intercomm extends Comm {
```

```
  public Object clone() { ... }

  // Inter-communication

  public int remoteSize() {...}

  public Group remoteGroup() {...}

  public Intracomm merge(boolean high) {...}

  ...
}
```

B.4 `Op`

```
public class Op {
  Op(UserFunction function, boolean commute) {...}

  void finalize() {...}

  ...
}
```

B.5 Group

```
public class Group {

  // Group Management

  public int size() {...}

  public int rank() {...}

  public int [] translateRanks(Group group1, int [] ranks1) {...}

  public static int compare(Group group1, Group group2) {...}

  public static Group union(Group group1, Group group2) {...}

  public static Group intersection(Group group1, Group group2) {...}

  public static Group difference(Group group1, Group group2) {...}

  public Group incl(int [] ranks) {...}

  public Group excl(int [] ranks) {...}

  public Group rangeIncl(int [] [] ranges) {...}

  public Group rangeExcl(int [] [] ranges) {...}

  public void finalize() {...}

  ...
}
```

B.6 Status

```
public class Status {

  public int index ;


  // Blocking Send and Receive operations

  public int getCount(Datatype datatype) {...}

  public int getSource() {...}
  public int getTag() {...}

  // Nonblocking communication

  public int getIndex() {...}

  // Probe and Cancel

  public boolean testCancelled() {...}


  // Derived datatypes

  public int getElements(Datatype datatype) {...}

  ...
}
```

B.7 Request and Prequest

```
public class Request {

  // Nonblocking communication

  public Status wait() {...}

  public Status test() {...}

  public Request() {...}

  public void finalize() {...}

  public boolean isVoid() {...}

  public static Status waitAny(Request [] arrayOfRequests) {...}

  public static Status testAny(Request [] arrayOfRequests) {...}

  public static Status [] waitAll(Request [] arrayOfRequests) {...}

  public static Status [] testAll(Request [] arrayOfRequests) {...}

  public static Status [] waitSome(Request [] arrayOfRequests) {...}

  public static Status [] testSome(Request [] arrayOfRequests) {...}


  // Probe and cancel

  public void cancel() {...}

  ...
}

public class Prequest extends Request {

  // Persistent communication requests

  public void start() {...}

  public static void startAll(Request [] arrayOfRequests) {...}

  ...
}
```

B.8 Datatype

```
public class Datatype {

  // Derived datatypes

  public Datatype contiguous(int count) {...}

  public Datatype vector(int count, int blocklength, int stride) {...}

  public Datatype hvector(int count, int blocklength, int stride) {...}

  public Datatype indexed(int [] arrayOfBlocklengths,
                          int [] arrayOfDisplacements) {...}

  public Datatype hindexed(int [] arrayOfBlocklengths,
                           int [] arrayOfDisplacements) {...}

  public static Datatype struct(int [] arrayOfBlocklengths,
                                int [] arrayOfDisplacements,
                                Datatype [] arrayOfTypes) {...}

  public int extent() {...}

  public int size() {...}

  public int lb() {...}

  public int ub() {...}

  public void commit() {...}

  public void finalize() {...}

  ...
}
```

B.9 Classes for virtual topologies

```
public class Cartcomm extends Intracomm {

  public Object clone() { ... }

  // Topology Constructors

  static public dimsCreate(int nnodes, int [] dims) {...}

  public CartParms get() {...}

  public int rank(int [] coords) {...}

  public int [] coords(int rank) {...}

  public ShiftParms shift(int direction, int disp) {...}

  public Cartcomm sub(boolean [] remainDims) {...}

  public int map(int [] dims, boolean [] periods) {...}
}

public class CartParms {

  // Return type for Cartcomm.get()

  public int [] dims ;
  public booleans [] periods ;
  public int [] coords ;
}

public class ShiftParms {

  // Return type for Cartcomm.shift()

  public int rankSource ;
  public int rankDest ;
}

public class Graphcomm extends Intracomm {

  public Object clone() { ... }

  // Topology Constructors

  public GraphParms get() {...}

  public int [] neighbours(int rank) {...}
```

```
  public int map(int [] index, int [] edges) {...}
}

public class GraphParms {

  // Return type for Graphcomm.get()

  public int [] index ;
  public int [] edges ;
}
```

References

[1] Bryan Carpenter, Geoffrey Fox, Guansong Zhang, and Xinying Li. A draft Java binding for MPI., November 1997. http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html.

[2] George Crawford III, Yoginder Dandass, and Anthony Skjellum. The JMPI commercial message passing environment and specification: Requirements, design, motivations, strategies, and target users, December 1997. http://www.mpi-softtech.com/publications.

[3] Rob Gordon. *Essential JNI: Java Native Interface*. Prentice Hall, 1998.

[4] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[5] Message Passing Interface Forum. MPI-2: Extension to the message passing interface. Technical report, University of Tennessee, July 1997. http://www.mpi-forum.org.

[6] Sava Mintchev and Vladimir Getov. Towards portable message passing in Java: Binding MPI. In M. Bubak, J. Dongarra, and J. Waśniewski, editors, *Recent Advances in PVM and MPI*, volume 1332 of *Lecture Notes in Computer Science*, pages 135–142. Springer Verlag, 1997.

[7] Sun Microsystems. Java code conventions. http://java.sun.com/docs/codeconv/.