MPI for Java

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

Abstract

Recently, there has been a large amount of interest in parallel programming using Java. However, efforts exploring the use of Java for parallel programming have been hindered by lack of a standard Java parallel programming API. To alleviate this problem, many groups have initiated research designing Java implementations of the successful Message Passing Interface (MPI). Unfortunately, MPI bindings are currently only defined for C, Fortran, and C++; as a result, initial Java MPI implementations have been divergent. This paper represents an effort to establish a consensus on Java bindings for MPI, and thereby to greatly enhance the viability of parallel programming using Java.

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 [5]. 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 [6], 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 [5]. While the MPI-2 release of the standard added a C++ binding [6], 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 [3]. Some of these "proof-of-concept" implementations have been available for more than a year with successful ports on clusters of workstations running Solaris, Windows NT, Irix, AIX, HP-UX, MacOS, and Linux, as well as the 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 modeled 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) [7] 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. MPIJ implements a large subset of MPI functionality including point-to-point communication (all modes), intracommunicator operations, groups, user-defined reduction operations. Notable capabilities that are not yet implemented include process topologies, intercommunicators, and user-defined datatypes (these are arguably needed for legacy code only).

MPIJ communication uses native marshaling of primitive Java types. On Win32 platforms this technique allows MPIJ to achieve communication speeds comparable to, and in many instances 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 marshaling to platforms other than Win32, investigation of standard libraries (e.g. BLAS) for improved performance, and porting to a completely applet-based version of DOGMA. This forthcoming version of DOGMA will allow non-technical users to run MPI programs on clusters of workstations without installing any system code or application code at all.

3 Java-MPI – Draft 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.

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.

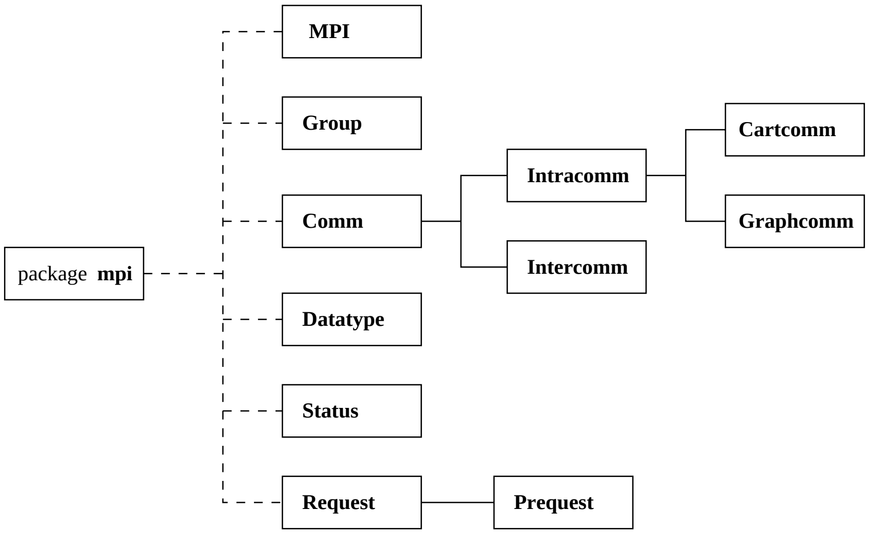

Figure 1: Principal classes of Java-MPI

3.1 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.

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 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.

3.2 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 [8]. 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 ("_"). 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.

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 interpreted 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 interpreted 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. Instead, the Java exception mechanism will be used to report errors as defined in [2].

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 auxiliary 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. 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 (e.g., 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.

Concurrent access to arrays. 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 [4] 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.

4 Example: Point-to-Point Communication

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();
  }
}
```

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.

```
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

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`. 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.

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.)

5 Open Issues

Derived datatypes. 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.

Probably the most compelling factor in favor of including MPI derived data types in the binding is the support for legacy MPI applications. For wrapper based implementations another factor is the relative simplicity with which derived data types can be supported. Lastly, the possible need to interact with native code that uses derived datatypes is best supported by including derived datatypes in the Java standard. However, native code interaction is arguably only a large concern for wrapper-based implementations.

The primary argument against including derived datatypes is that their functionality is already provided by the standard Java objects, and that their addition only adds unneeded complexity. Implementers of pure Java MPI systems tend to favor this approach.

Overloaded communication operations. 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 primary argument in favor of this approach is that it simplifies user code. For instance,

```
MPI.COMM_WORLD.send(message, 0, message.length, MPI.CHAR, 1, 99);
```

becomes

```
MPI.COMM_WORLD.send(message, MPI.CHAR, 1, 99);
```

The obvious counter-argument is that it significantly increases the total number of methods in the API.

A possible compromise is to provide overloaded versions only of specific common functions such as point-to-point communication functions. The counter-argument to this is that it is inconsistent.

Multidimensional arrays. It has been suggested that some specific support for multidimensional arrays may be desirable. In the current proposal, communicating multidimensional arrays depends either on sending arrays one row at a time or on Java object serialization, both of which might be a performance bottleneck. For instance, MPIJ sends a 200x200 array of doubles over Fast Ethernet three times faster when multidimensional array support is included than when individual rows are sent.

A plausible position is to wait and see what happens in Java regarding multidimensional arrays and efficient object serialization before complicating this API.

User defined operations. It has been suggested that the specification of user defined operations be modified. The current design is modeled after a procedural approach where users define functions and cannot simply define a new operation class. So, in the current proposal users create a `UserDefinedOperation` and use this to create an `Op`. This results in the creation of a class (`UserDefinedOperation`) which is not really necessary.

An alternative design for user defined operations would be to simply have users define subclasses of `Op` which would have a method named something like `userMethod` or `call`. This design would also eliminate the overhead of a method invocation.

Error handling. The manner in which MPI errors are handled in Java is largely still an undefined issue. What has been clearly defined is that, as mentioned previously, the Java exception mechanism is used to replace error codes. However, the exact format of these exceptions is undefined.

One possibility is that all MPI exceptions be derived from two classes: `MPIException` and `MPIRuntimeException`. Subclasses of `MPIException` would represent errors that the user would be required to catch whereas subclasses of `MPIRuntimeException` would represent uncommon or unusual errors. Also, it has been proposed that certain MPI exceptions carry subexceptions when the cause for the MPI exception is another exception.

The use of user defined and predefined error handlers is still an open question. These could still serve a purpose in addition to the exception mechanism mentioned earlier. For instance, the predefined error handler *MPI_ERRORS_ARE_FATAL* could call `MPI.abort` while the predefined error handler *MPI_ERRORS_RETURN* would allow the user control over when `MPI.abort` was called.

Profiling interface. A profiling interface for Java MPI has not yet been defined. A possible general design approach is for profiling class and method names to exactly match those of the non-profiling classes and methods. Implementors would then place the compiled binary files in different locations. As Java linking is always dynamic, this would allow users to enable or disable profiling simply selecting the appropriate codebase (e.g. by changing the CLASSPATH).

6 Future Work

- 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
- MPI-2

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 marshaling and unmarshaling 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.

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] Bryan Carpenter, Vladimir Getov, Glenn Judd, Tony Skjellum, Geoffrey Fox. MPI for Java: Position Document and Draft Specification. Technical Report JGF-TR-03, Java Grande Forum, November 1998.
http://www.javagrande.org/reports.htm

[3] 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.

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

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

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

[7] 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.

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

[9] Glenn Judd, Mark Clement, Quinn Snell. Distributed Object Group Metacomputing Environment, January 1998.
http://ccc.cs.byu.edu/DOGMA/.