Towards seamless computing and metacomputing in Java

Denis Caromel, Wilfried Klauser and Julien Vayssière

INRIA Sophia Antipolis
CNRS - I3S - Univ. Nice Sophia Antipolis
BP 93, 06902 Sophia Antipolis Cedex
tel: 33 93 65 76 31, fax: 33 93 65 78 58, email: First.Last@sophia.inria.fr
http://www.inria.fr/sloop/javall/

Abstract

Due to its platform-independent execution model, its support for networking, multithreading and mobile code, Java has given hope that easy Internet-wide high-performance network computing was at hand. Numerous attempts have then been made at providing a framework for the development of such metacomputing applications. Unfortunately, none of them addresses *seamless sequential, multithreaded and distributed computing*, i.e. the execution of the same application on a multiprocessor shared-memory machine as well as on a network of workstations, or on any hierarchical combination of both. In this article we first identify four requirements for the development of such metacomputing frameworks. We then introduce Java// (pronounce *Java Parallel*), a 100% Java library that provides transparent remote objects as well as asynchronous two-way calls, high reuse potential and high-level synchronization mechanisms. We also present the metaobject protocol (MOP) Java// is built on and describe a distributed collaborative raytracing test application built using Java//.

1 Introduction

In order to provide a framework for the development of *cross-paradigm metacomputing environments* [7][10][8] within the scope of the Java language [1] and environment [12], we identify four key requirements: polymorphism between local and remote objects, high-level synchronization mechanisms, reuse of sequential code and the availability of a 100% Java portable library.

1.1 Transparent remote objects

First, let us focus on cross-paradigm portability.
Cross-platform portability is genuinely achieved by the standard Java execution environment. An application written in Java is compiled into an architecture-neutral bytecode format, which then executes on a Java Virtual Machine (JVM) whose purpose is to hide the nature of the underlying platform.
Some JVM implementations provide access to native threads, which, when run on a multiprocessor machine, permits automatic mapping of Java threads onto the set of available processors. This feature abolishes the frontier between a monoprocessor machine and a multiprocessor, shared-memory machine when it comes to executing multithreaded Java applications. It results in instant speedup for applications built around concurrent activities, provided there actually is some parallelism between the threads.
Consequently, code reuse for porting Java threaded applications from a monoprocessor machine to a multiprocessor machine is not an issue since the application code for a monoprocessor machine does not need any modification at all to run on a multiprocessor shared-memory machine.
Nevertheless, a huge gap yet exists between multithreaded and distributed Java applications which forbids code reuse in order to build distributed applications from multithreaded applications. Both JavaRMI and JavaIDL, as examples of distributed object libraries in Java, put a heavy burden on the programmer because they require deep modifications of existing code in order to turn local objects into remote-accessible ones.
In these systems, remote objects need to be accessed through some specific interfaces. One could argue that programming to an interface is usually considered as a better practice than programming to an implementation. This is undoubtedly true, but the core of the problem is that implementation classes are forced to move from one place in the inheritance graph to another in order to become remote-accessible

classes. Method signatures also need to be modified in order to throw distribution-related exceptions, which does not allow a clear separation of concerns between functional code and distribution-related code. As a consequence, these distributed objects libraries do not allow polymorphism between local and remote objects. This feature is our first requirement for a metacomputing framework. It is strongly required in order to let the programmer concentrate first on modeling and algorithmic issues rather than lower-level tasks such as object distribution, mapping and load balancing.

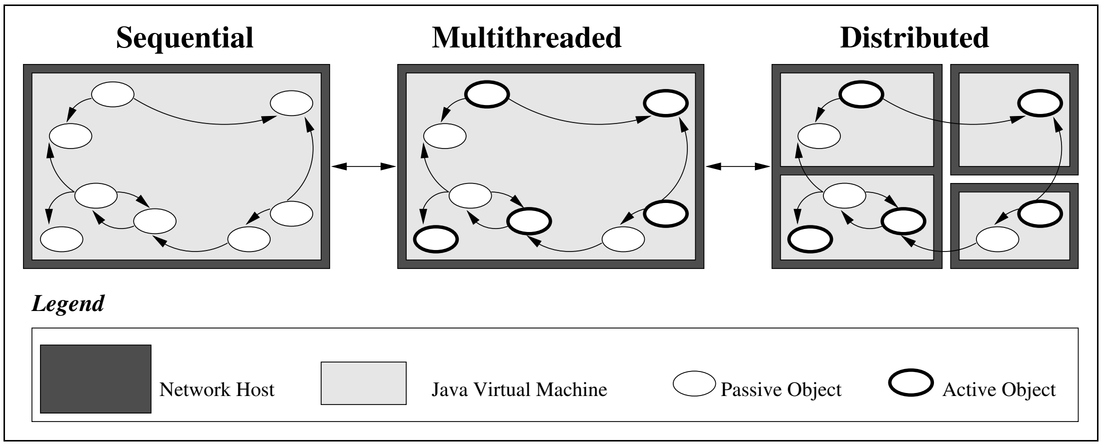

Figure 1: Seamless sequential, multithreaded and distributed programming

1.2 High-level synchronization mechanism

Our second requirement for metacomputing is higher-level synchronization mechanisms. Although monitor-like primitives [13] found in Java may be theoretically sufficient for expressing synchronization, implementing complex synchronization specifications using such low-level tools is definitely cumbersome and error-prone. Moreover, such architectures do not scale well and have some reuse problems [2]. Such an architecture also assumes a shared memory of some kind, which does not fit well in a system that needs to address distribution as well.

1.3 Reuse of sequential code

Reuse of sequential code does not here means reusing legacy applications in order to build concurrent and distributed Java applications. It is our opinion that sequential code should no more be considered as a single-threaded implementation of a problem but rather as a high-level expression of the fundamental abstractions and operations in a system. In this context, code reuse simply means adapting the sequential code to a particular metacomputing environment. This includes, but is not limited to, deciding which objects should have their own activity (thread) and how they should be distributed over a set of computing nodes.

When designing an object-oriented application, the programmer usually starts with creating high-level domain-dependent abstractions and turns these into objects and classes. These classes and objects are then connected together using inheritance, composition or any other technique, which eventually results in a modelling of the domain managed by the application.

Deciding which objects should have an activity on their own or distributing objects over several adress spaces is definitely a lower-level issue. As a matter of fact, object distribution or the expression of parallel activities is always constrained by the actual system the application should be implemented on.

This is why we believe a framework for metacomputing applications should provide a clear separation between high-level application design and lower-level implementation issues such as object distribution or managing concurrent activities, and postpone as much as possible deciding which objects should be active.

1.4 A portable, non-intrusive library

A rather large number of research projects have already been conducted on transparent remote objects in Java [19][17].

Two major implementation techniques are used: some change the Java Virtual Machine or the Java-to-bytecode compiler, other rely on some source pre-processing. These techniques lead to two different flaws. The first one fails at providing Internet-wide portability by requiring installation of a specific runtime environment on each possible node of the computation. The second one requires that the programmer has access to the source code of the objects he wants to make remote, which is barely never the case when using third-party libraries. Consequently, a library that aims at distributing Java objects transparently has to be 100% Java and only require access to the compiled representation of classes, not to the sources.

2 The Java// framework

In order to meet these requirements, we have designed and implemented Java// (pronounce *Java Parallel*), a Java library for seamless sequential, multi-threaded and distributed programming.
Java// only consists of a collection of 100% Java classes, thus requiring no change to the standard Java execution environment. The Java// model uses by default the following principles:

- heterogeneous model with both passive and active objects (threads, actors)
- sequential processes
- unified syntax between message passing and inter-process communication
- systematic asynchronous communications towards active objects
- wait-by-necessity (automatic and transparent futures)
- automatic continuations (a transparent delegation mechanism)
- no shared passive objects (call-by-value between processes)
- centralized and explicit control by default
- polymorphism between standard objects, active objects, and remote objects.

2.1 Model of execution

Given a standard Java object, there are several new behaviors we would like to transparently give it: location transparency, activity transparency and advanced synchronization mechanism.
Location transparency provides polymorphism between local and remote objects. Activity transparency hides the fact that methods invoked on an active object actually execute in a separate thread. This is achieved using transparent future objects and wait-by-necessity [5]. Advanced synchronization mechanisms allow an easy and safe implementation of potentially complex synchronization policies.
Let's have a look at how these different features can be obtained within the scope of the Java language. In most distributed objects systems, such as RMI or CORBA, location transparency is achieved using the proxy pattern [11]. A local object (the so-called *proxy*) acts as a representative for an object that resides in another address space, possibly on another machine across a network. This proxy encapsulates all communication details so that other local objects do not know they are actually sending messages to a remote object.

Adding a new behavior to an object, such as its own thread of execution, may be usually achieved using two different object-oriented techniques: multiple inheritance and composition. Multiple inheritance allows effortless extension of a class behavior, provided these different behaviors be rather orthogonal, like functional code and synchronization for example. Composition mimes a complex object by delegating different behaviors to different objects.
As Java features simple class inheritance and multiple interface inheritance, we have chosen to take the best from both worlds. We use composition for implementation of multiple behaviors while multiple interface inheritance is used for declaring these behaviors.

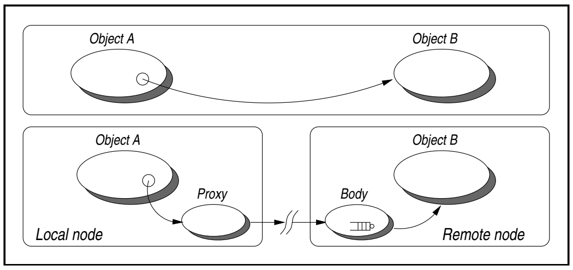

Figure 2: Standard and Java// models of execution contrasted.

In Java//, any standard object (figure 2 (top)) may be extended through composition with a pair of objects : a *proxy* and a *body* (figure 2 (bottom)). The *active object* is actually composed of two objects: the body and the standard object.
In terms of metaobjects, the proxy transparently reifies method invocations. Method invocations are 'trapped' and converted into instances of the `MethodCall` class. These method invocations may then be

manipulated as first-class objects in order to implement any new semantics.
The body receives these reified calls and stores them into a queue of pending calls. It then executes them in an order specified by a given synchronization policy. If none is provided, the body defaults to a FIFO behavior.
The programmer has the possibility to override the default FIFO-ordered policy by writing code for explicitely managing the queue of pending calls on an object. This gives him total control over the synchronization strategy. Note that any other synchronization abstraction may be implemented using the explicit one, such as, for example, an *implicit* one where the programmer declares a set of properties that constrain the default FIFO-ordered policy.

2.2 Programming active objects

Given a sequential Java program, it takes only minor modifications from the programmer to turn it into a multithreaded, ready-for-metacomputing program. We'll first focus on active object creation and then discuss inter-object synchronization.

Active Object Creation

Java// actually only requires instanciation code to be modified in order to transform a standard object into an active one. All the code that had previously been written for the passive version of the same object remains perfectly valid. Besides the standard constructor parameters for the object, the creation of an active object requires at least the node to create the object on (represented by an instance of class `Mapping`. Depending on special semantics requirements, additional parameters may be passed.
Here's a sample of code with several techniques for turning a passive instance of class `A` into an active, possibly remote, one.
A standard object created through such a statement:

```
A a = new A ("foo", 7);
```

becomes either

- **instanciation-based**

  ```
  Object[] params = {"foo", new Integer (7)};
  A a = (A) Javall.newActive ("A", params, myMapping);
  ```

- **or class-based**

  ```
  class pA extends A implements Active {}
  Object[] params = {"foo", new Integer (7)};
  A a = (A) Javall.newActive ("pA", params, myMapping);
  ```

- **or object-based**

  ```
  A a = new A ("foo", 7) ;
  a = (A) Javall.turnActive (a, myMapping);
  ```

This piece of code creates an instance of class `A` or `pA` on node `myMapping`. The mapping mechanism between nodes and actual virtual machines, processors and network hosts will be described later. The active instance just created owns its own thread that executes methods invoked on this object in a default FIFO order. The semantics of calls to such an object are transparently asynchronous, with no code modification being required on the caller's side. This sample also illustrates instanciation-based reification (see section 4) contrasted with class-based reification and object-based reification.

- Instanciation-based reification is much of a convenience technique. It allows the programmer to create an active instance of `A` with a FIFO behavior without defining any new class.
- Class-based reification is the core of Java//'s philosophy. Given a class `A`, the programmer writes a subclass `pA` that inherits directly from `A` and implements a specific interface such as `Active`. He or she may also provide a `live` method in class `pA` for giving a specific activity or managing synchronization, as we'll see in section 2.3

- Object-based reification makes use of the `Javall.turnActive` method, which enables us to attach an active behavior to an existing object at any time after its creation. This is especially useful when we do not have access to the code that creates the standard object we want to make active.

We suggest the use of the *factory method* pattern [11] in order to nicely encapsulate the code needed to instanciate active objects. This would result in a static method `newActive` in class `pA` :

```
public static pA newActive (String s, int i, Mapping m)
{
  Object[] params = {s, new Integer (i)};
  return (pA) Javall.newActive ("pA", params, m);
}
```

As another important advantage, this technique reduces the amount of code needed to instanciate active objects using Java// :

```
a = pA.newActive ("foo", 7);
```

Inter-object synchronization

Asynchronous message-passing would not be of much interest if the user had to explicitely add synchronization to the code that invokes methods on active objects. Fortunately, Java// provides a mechanism of transparent futures.
When a method is invoked on an active object, it immediately returns a future object. This object acts as a placeholder for the result of the not-yet-performed method invocation. As a consequence, the calling thread can go on with executing its code, as long as it doesn't need to invoke methods on the returned object, in which case the calling thread is automatically blocked if the result of the method invocation is not yet available.
Future objects in Java// are said to be transparent because they do not require any modification of the caller's code. They are automatically created when a method is invoked on an active object: this is the *wait-by-necessity* principle [3][5]. Transparent future objects are possible because the automatically-created future object is actually an instance of a subclass of the returned object, which is compliant with all compile- and runtime type checks and does not weaken software quality.
We believe asynchronous calls and future objects can dramatically improve performance of Internet-wide computations. Because huge latency is the plague of today's Internet, wait-by-necessity can help automatically overlap computations and communications. As a consequence, the Java virtual machine that runs at a node in a computation spends less time in the idle state waiting for some remote computation to complete.
There are a few cases where future objects are not available. Primitive types cannot lead to future objects because they are not standard objects and thus cannot be subclassed. The same limitation applies to final classes, which includes all arrays.
We have also chosen to forbid the use of future objects for methods that throw checked exceptions. If this were allowed, the execution of a method on an active object could throw an exception in the calling thread at a point where the calling thread has exited the `try` clause. This would result in an exception being thrown in a context where it cannot be caught, thus modifying the semantics of the application and most likely resulting in an application crash.
We have seen that when using wait-by-necessity, a thread is automatically suspended if it needs an object that is not yet available. This is why wait-by-necessity is said to be a *data-driven* synchronization mechanism, as opposed to *control-driven* synchronization. Yet, there are some situations where synchronization is not directly connected to the call of a method on an object. This is why Java// provides the `Javall.waitFor(Object obj)` static method which performs an *explicit wait* on object `obj`, without calling any method on this object. Another static method, `Javall.isAwaited(Object obj)` returns a boolean that indicates if object `obj` is awaited or not. This enables the thread to do something useful instead of being suspended while `obj` is awaited.

Mapping active objects to nodes

Hiding platform-specific details from the programmer is one of the great strengths of Java. Yet, this fundamental design choice sometimes prevents the programmer from fine-tuning parameters like performance or distribution, because it does not allow any deep insight on what the actual platform is. More specifically, it influences the way Java// deals with mapping active objects to actual processing ressources

(processors).
Java// library provides a unique abstraction for representing processing power resources: the class `Mapping`. This is the only locator used for specifying where to create active objects. Since the programmer is entirely free to instanciate these `Mapping` objects wherever he or she wants, having a single locator class does not mean less flexibility. A `Mapping` object may, for example, represent a multiprocessor machine where a single JVM transparently maps its threads onto the set of available processors. On the other hand, there may be different `Mapping` objects within one single VM, so that it is possible to test a distributed application on a single workstation, thanks to the seamless transition between multithreaded and distributed applications Java// provides. When working with a real cluster of workstations, the programmer may instanciate one `Mapping` object for each workstation. In the case of a metacomputing application, a `Mapping` object would be instanciated in every Web browser that participates.
As a consequence, the mapping of active objects to actual computing ressources through the `Mapping` class enables the distribution of the application over a set of abstract nodes. Switching between different execution environments (such as a single workstation, a multiprocessor machine, a cluster of workstations or a collection of Web browsers) becomes a matter of only a few seconds, as demonstrated by the examples in the next section. It also allows for easily implementing any load-balancing policy, either generic or application-specific.

2.3 Intra-object Synchronization

Active objects instanciated through the `Javall.newActive` static method are transparently given their own thread that executes invoked methods with a default FIFO order. This thread is started by the object that owns the queue of pending method invocations on an object: the *body*.
Java// provides a mechanism for specifying synchronization of method invocations on a given active object. The purpose of this mechanism is to enhance the standard thread synchronization mechanism [16] with an *explicit* method for specifying synchronization. The biggest difference with the standard thread synchronization mechanism is that synchronization is now centralized in one special method of a class, instead of being disseminated in all methods of a class.
The responsibility for specifying the synchronization policy for a class is placed on its `live(Body myBody)` method. If no `live` method is provided by the class of the reified object, the body uses its own default `live` method. For most bodies, the default policy is FIFO.
If the class implements `Active`, the default mechanism, a thread of control is explicitely available and it is then the responsibility of the `live` method to explicitely manage the queue of pending requests, if the programmer wishes to override the default FIFO policy. It does so by invoking methods on the Body, such as `serveOldest`, `serveOldest (Method met)`, `serveOldestBut (Method met)`, `waitARequest()`. This methods are provided as a service library for managing the queue of pending calls. This service library also includes iterators for safe traversing of the queue of pending requests with access to the effective value of the parameters of the calls.

The FIFO behavior provided by default is simply implemented as follows

```
live (Body myBody)
{
  while (true)
  {
    myBody.serveOldest ();
    myBody.waitARequest ();
  }
}
```

Please note that `waitARequest` blocks if the state of the queue of pending requests is modified (no active wait).
Now consider the case of the canonical Bounded Buffer example. We assume we have a class `FixedBuffer` that implements a fixed-length buffer and features methods `put` and `get` as well as `isEmpty` and `isFull`. In order to achieve consistency, a typical programming could be :

```
class BoundedBuffer extends FixedBuffer implements Active
{
 live (Body myBody)
 {
```

```
    while (true)
    {
      if      (this.isFull())   myBody.serveOldest ("get");
      else if (this.isEmpty())  myBody.serveOldest ("put");
      else myBody.serveOldest();

      myBody.waitARequest ();
    }
  }
}
```

Given these two synchronization constraints, the Body object manages the queue of pending request properly. As several methods in the same class may have the same name and different parameter types, we provide a convenience mechanism of shortcuts that associates a string to a given method, which results in less code for constraints declaration. If for example class `A` contains two methods `foo` with different argument types, shortcuts may be created as follows:

```
java.lang.reflect.Method method1, method2;
// Some code for obtaining Method objects for
// these two 'foo' methods through Reflection API.
// [...]

Javall.setShortcut ("A", "foo1", method1);
Javall.setShortcut ("A", "foo2", method2);
```

2.4 Abstraction for implicit synchronization

In the case of an *implicit* synchronization declaration (which is implemented on top of the basic explicit mechanism), the programming of the buffer synchronization policy would be as follows.

```
class BoundedBuffer extends FixedBuffer implements ImplicitActive
{
 live (Body myBody)
 {
   myBody.forbid ("put", "isFull");
   myBody.forbid ("get", "isEmpty");
 }
}
```

Implicit synchronization has proven to be better than its explicit counterpart with respect to ease of reuse and better scalability. On the other hand, the overhead needed to decide which call to execute given a set of constraints may not be neglectible and in general the explicit technique has more expressiveness. However, high-performance computing often relies on relatively simple synchronization policies.
This technique is easily extensible and the programmer is free to implement new abstractions for intra-object synchronization [4]. Each of these implementations should result in a body class that implements the synchronization policy and an interface inheriting directly or indirectly from `Active` which declares the name of the proxy class (usually the default asynchronous proxy) and the name of the body class (see figure 3). Such an interface helps organize synchronization abstractions logically and is used by classes such as `pA` (see 2.2) in order to choose which synchronization technique they would like to use. The interface `Reflect` does not provide any functionality but acts as a common root interface for all behaviors implemented using the Java// metaobject protocol.

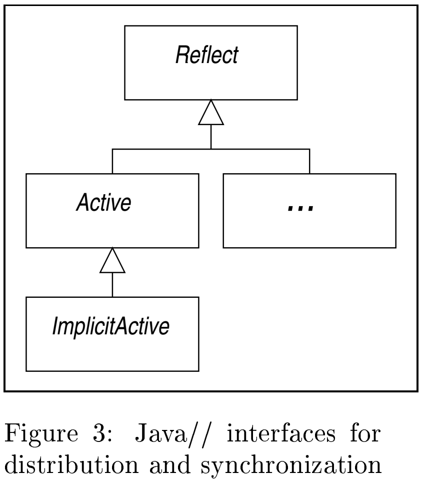

Figure 3: Java// interfaces for distribution and synchronization

1. Sequential design and programming
2. Active objects identification
   - Initial activities
   - Shared objects
3. Active objects programming
   - Define each active object class
   - Define the activity (**live**)
   - Use the active objects classes
4. Adaptation to constraints
   - Refine the topology
   - Define new active objects

Figure 4: The 4 steps of the method

2.5 A method for reuse

As Java// is an extension of Eiffel// [3] and C++// [6], it may be the support for a method for reuse first described in [5]. Its main feature is to postpone the identification of active objects in the design of an application. The programmer may then concentrate on application design and not mix it with the division of the application in concurrent activities. The main steps of this method are shown in figure 4.

3 Example and performances

3.1 Distributed matrix-vector product

We have implemented an example proposed by Raje, William and Boyles in [18]: a matrix-vector product, the rows of the matrix being split between two machines. The matrix is a square matrix of size 1000 containing `float` numbers.

We make extensive use of wait-by-necessity in order to automatically overlap local and remote calculations. The time we consider includes sending the vector, performing the calculation and returning the result. It does not include the initial transmission of the remote rows of the matrix.

Here is the code for the `main` method of the **sequential** version of the program :

```
public static void main (String args[])
{
  // Size of the matrix
  int n = 1000;
  // number of rows on the local node
  int m;
  // One initial matrix and two submatrixes
  Matrix m0, m1, m2;
  // Initial, temporary and final vectors
  Vector v0, v1, v2, v3;

  // Some initialization code
  [...]

  // Creates submatrixes of sizes  m and n-m
  m1 = m0.getBlock (0, 0, m, n-1);
  m2 = m0.getBlock (m+1, 0, n-1, n-1);

  // Computes both right products
```

```
  v1 = m1.rightProduct (v0);
  v2 = m2.rightProduct (v0);

  // Creates result vector
  v3 = v1.concat (v2);
}
```

Now assume we want to get a **multithreaded and distributed** version of this program. The only modifications we need to bring to the source code are located in the portion of code where we create the objects we want to make active.

```
m1 = m0.getBlock (0, 0, m, n-1);
m2 = m0.getBlock (m+1, 0, n-1, n-1);
```

If we had access to the code of the `Matrix` class, we would like to modify it in such a way that the implementation of method `getBlock` in class `Matrix` now returns an active object instead of a standard one. But this method would have several flaws :

- Every invocation of this method would return an active object, even if we do not want to.
- Its signature would have to be modified in order to take into account a new argument: the `Mapping` object specifying where to create the active object.

This is why we provide the `Javall.turnActive` method in order to attach an active behavior to an active object after its creation. As a consequence, we only need to add these two lines to method `main`, after the standard `m1` and `m2` :

```
m1=(Matrix) Javall.turnActive(m1, remoteMapping);
m2=(Matrix) Javall.turnActive(m2, localMapping);
```

As a general rule, we do not assume we have access to the code of the linear algebra library. Consequentely, using `Javall.newActive` is not always possible, since submatrixes might be instanciated inside the library (actually inside the body of method `getBlock`) and returned as a result of this method invocation.
Parallelism is achieved here because, as both `m1` and `m2` are active objects, both calls to `rightProduct` are asynchronous and return future objects for representing the not-yet-available result vectors (namely `v1` and `v2`). As a consequence, the thread that executes `main` launches this two products and is then blocked on the call to `concat` because `v1` is not available at the moment. Both products are executed in parallel on two different nodes of the computation (the local node `localMapping` and a remote one designated as `remoteMapping`.
Let us now assume we want to run the same program on an SMP machine with a JVM using native threads. The only modification needed would be to change `remoteMapping` to the current node (`localMapping` here), through the `javall-mapping` file.

3.2 Performances

Figure 5 shows the time needed to compute the product with respect to the number of rows on the remote machine. Both the local and the remote machine were UltraSparcs. As in [18], the minimum is reached for 400 remote rows and 600 local ones. This is not surprising at all since both Java// and ARMI are implemented on top of RMI.
Our implementation of Java// is based on Java RMI. Our experience with RMI lead us to the following conclusion: RMI shows catastrophic performance when it comes to exchanging large-size messages, such as a whole matrix or a very large graph of objects. RMI itself is not directly responsible for this, but the default serialization mechanism is. As a general rule, it is today hard to achieve speedup on a network of workstations when the communication/computation ratio is too high.

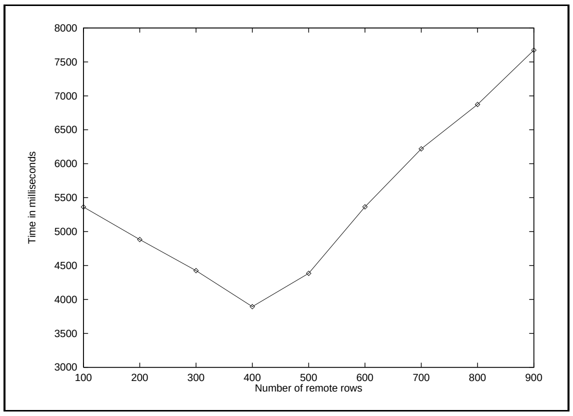

Figure 5: Java// performances for Matrix-Vector computation

3.3 A parallel and distributed collaborative application

Introduction

In order to demonstrate the power of Java// regarding synchronization, we have implemented a collaborative application. It showcases three of the major benefits of Java//: ease of distributed programming, speedup through parallel calculation, and ease of synchronization. Several users can collaboratively view and manipulate a 3D scene, which involves election and synchronization issues. The image for the scene is calculated by a dynamic set of rendering engines using a raytracing algorithm, everything being controlled by a central dispatcher.

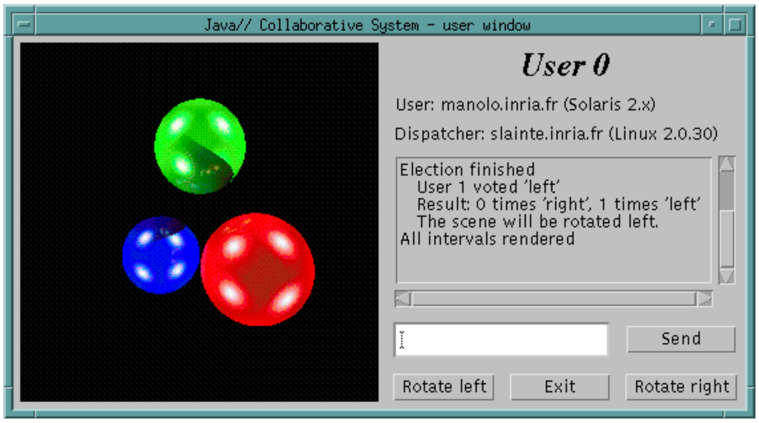

Figure 6: Screen shot of a user window

Users can rotate the scene and send messages to other users. They are informed by the dispatcher on events of public interest (i.e. user set changes, elections, rendering).

Design

An instance of class `Dispatcher` (see figure 7) acts as the central management module. It uses Java// to dynamically distribute the calculation on remote hosts, to centralize all synchronizations and to provide collaborative services to the users. Furthermore, it serves as a static entry point for the users (each running an instance of `UserFrame`), it manages the set of users and holds all the cross-references.

The first user to register at the `Dispatcher` instance determines the scene to be rendered. The rendering task is split up by the `Dispatcher` in a stack of image `Interval` instances. Those are assigned to the set of `RenderingEngine` instances with a first-returned, next-assigned load-balancing. Each `RenderingEngine` receives one initial `Interval` first, the remainder of the `Interval` stack is given to the engines when they return the results. The `Dispatcher` collects the image data and forwards it to the users `UserFrame` instance.

Thanks to Java//, all method calls between the application modules are asynchronous, which provides efficient communication-calculation overlapping. Neither the users nor the `Dispatcher` is waiting for the calculation results, they are asynchronously called back instead.

`UserFrame` and `DispatcherFrame` inherit from the AWT `Frame`, which is a heavy-weight class with a lot of public methods. To keep the stub small and to minimize overhead, we choose to activate `Redirector` and `Dispatcher` instead.

Note that each active object can be selectively and dynamically mapped into a Java virtual machine (JVM), which can be either local or remote. This gives a wide range of distribution options, from having all active objects in the same local JVM, to having each active object distributed in its own remote JVM. The standard objects always reside in the JVM of the active object they are controlled by.

Synchronization

All the `Dispatcher` synchronizations are centralized in the `live` method. It uses Java// service methods (see section 2.3) to manipulate the queue of pending method calls.

An instance of `Election` is created when two or more users are registered at the dispatcher and one user clicks to rotate in any direction. This request is converted into the first vote. The new `Election`

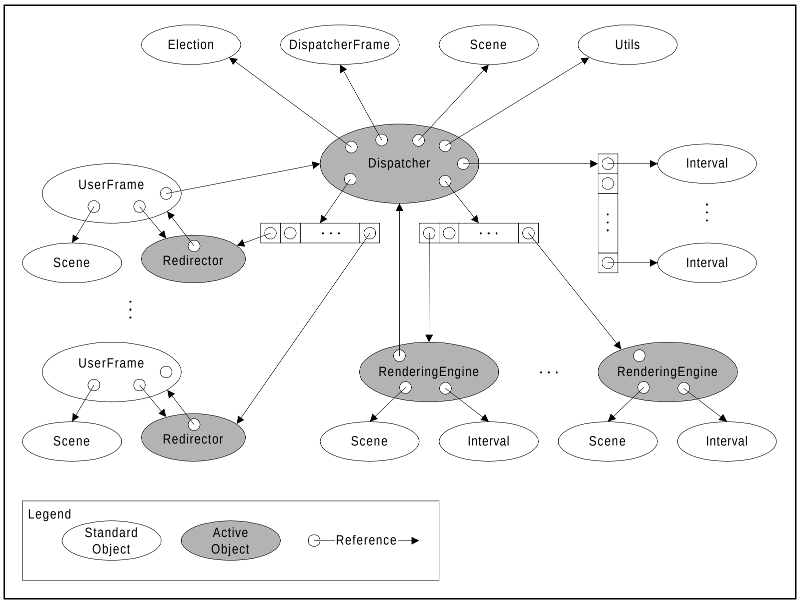

Figure 7: Runtime instance graph

instance starts its own standard Java thread, notifies all users of the election, and runs until either all users have voted or a 5 second time-out is reached. It then counts the votes, picks the decision (either unanimity or majority mode) and starts the rotation. While the election is running, rotate requests are taken as votes, other requests are ignored. While the image is rendered, rotate requests are refused, other requests are served.

```
public void live(Body body) {
  // Handles the election, will run in its own thread
  Election election;

  // Loops over lifetime of this Dispatcher
  while (body.isActive()) {
    // Waits on any method call
    body.waitForRequest();
    // Reads the call from the line
    MethodCall c = body.readOldest();
    // Obtains the name of the method called
    String s_method = c.getMethodName();

    // Somebody has called rotateLeft()
    if (s_method.equals("rotateLeft")) {
      // Gets the number of the user who has called
      int i_user = c.getIntegerArgument(0);

      if (Election.isRunning()) {
        body.removeOldest();
        // casts the vote
        int i_votes = election.vote(i_user, Election.LEFT);
        if (i_votes == users.number()) {
          // Everybody voted, stops the election immediately
          election.finish();
        }
      } else {
        if (isRendering()) {
          // Rendering in progress, ignores request
          body.removeOldest();
        } else if (users.number() == 1) {
          // Only one user, starts immediately
          body.serveOldest();
        } else {
```

```
          body.removeOldest();
          // Several users, starts a new election with the current vote
          election = new Election(i_user, Election.LEFT);
        }
      }
    } else if (s_method.equals("rotateRight")) {

      [...] // Similar code as above for rotateLeft

    } else if (!Election.isRunning()) {
      // No election is running, serves the request immediately
      body.serveOldest();
    } else {
      // Election is running, refuses the request
      body.removeOldest();
    }
  }
}
```

4 Implementation: a Meta-Object Protocol

Java// is built on top of a metaobject protocol (MOP) [14] that permits reification of method invocation and constructor call. As this MOP is not limited to the implementation of our transparent remote objects library, it also provides an open framework for implementing powerful libraries for the Java language.
As for any other element of Java//, this MOP is entirely written in Java and does not require any modification or extension to the Java Virtual Machine, as opposed to other metaobject protocols for Java [15]. It makes extensive use of the Java Reflection API, thus requiring JDK 1.1 or higher. JDK 1.2 is required in order to suppress default Java language access control checks when executing reified non-public method or constructor calls.
If the programmer wants to implement a new metabehavior using our metaobject protocol, he or she has to write both a concrete (as opposed to abstract) class and an interface. The concrete class provides an implementation for the metabehavior he or she wants to achieve while the interface contains its declarative part. The concrete class implements interface `Proxy` and provides an implementation for the given behavior through the method `reify`:

```
public Object reify (MethodCall c)
          throws InvocationTargetException, IllegalAccessException;
```

This method takes a reified call as a parameter and returns the value returned by the execution of this reified call. Automatic wrapping and unwrapping of primitive types is provided. If the execution of the call completes abruptly by throwing an exception, it is propagated to the calling method, just as if the call had not been reified.
The interface that holds the declarative part of the metabehavior has to be a subinterface of `Reflect` (the root interface for all metabehaviors implemented using Java//). The purpose of this interface is to declare the name of the proxy class that implements the given behavior. Then, any instance of a class implementing this interface will be automatically created with a proxy that implements this behavior, provided that this instance is not created using the standard `new` keyword but through a special static method: `MOP.newInstance`. This is the only required modification to the application code. Another static method, `MOP.newWrapper`, adds a proxy to an already-existing object; the `turnActive` function of Java//, for example, is implemented through this feature.
Here's the implementation of a very simple yet useful metabehavior : for each reified call, the name of the invoked method is printed out on the standard output stream and the call is then executed. This may be a starting point for building debugging or profiling environments.

```
class EchoProxy extends Object implements Proxy
{
  // here are constructor and variables declaration
  // [...]

  public Object reify (MethodCall c) throws InvocationTargetException, IllegalAccessException
  {
```

```
    System.out.println (c.getMethodName());
    return c.execute (targetObject);
  }
}

interface Echo extends Reflect
{
  public String PROXY_CLASS= "EchoProxy";
}
```

Instanciating an object of any class with this metabehavior can be done in three different ways : instanciation-based, class-based or object-based. Let's say we want to instanciate a `Vector` object with an `Echo` behavior.

- Standard Java code would be :

  ```
  Vector v = new Vector (3);
  ```

- Java// code, with instanciation-based declaration of the metabehavior :

  ```
  Object[] params = {new Integer (3)} ;
  Vector v = (Vector) MOP.newInstance
    ("Vector", params, "EchoProxy", null) ;
  ```

  The last parameter is because we do not have any additional parameter to pass to the proxy.

- with class-based declaration:

  ```
  public class MyVector extends Vector
                         implements Echo {}

  Object[] params = {new Integer (3)} ;
  Vector v = (Vector) MOP.newInstance
                ("Vector", params, null);
  ```

- with object-based declaration:

  ```
  Vector v = new Vector (3);
  v=(Vector) MOP.newWrapper ("EchoProxy",v);
  ```

which is the only way to give a metabehavior to an object that is created in a place where we cannot edit source code. A typical example could be an object returned by a method that is part of an API distributed as a JAR file, without source code. Please note that, when using `newWrapper`, the invocation of the constructor of the class `Vector` is not reified.

All the interfaces used for declaring *metabehaviors* inherit directly or indirectly from `Reflect`. This leads to a hierarchy of metabehaviors such as shown in figure 8.
Note that `ImplicitActive` inherits from `Active` to highlight the fact that implicit synchronization somewhere always relies on some hidden explicit mechanism. Interfaces inheriting from `Reflect` can thus be logically grouped and assembled using multiple inheritance in order to build new metabehaviors out of existing ones.

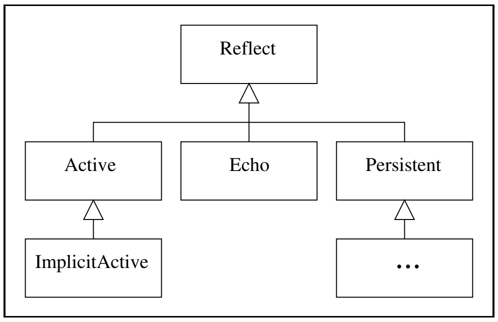

Figure 8: Java// interfaces

Due to its commitment to be a 100% Java library, the MOP has a few limitations:

- Calls sent to instances of `final` classes (which includes all arrays) cannot be reified.
- Primitive types cannot be reified because they are not instance of a standard class.
- Final classes (which includes all arrays) cannot be reified because they cannot be subclassed.

5 Conclusion and Future Work

We have designed and implemented Java//, a 100% Java library aimed at providing a framework for the seamless development of sequential, multithreaded and distributed applications. It features transparent active and remote objects as well as asynchronous calls, transparent future objects and wait-by-necessity. The most important feature of Java// is that it provides very smooth transition between sequential, multithreaded and distributed programming.

Java// is implemented without any modification of the Java Virtual Machine or any element of the standard Java environment. It is only made of 100% Java classes and heavily relies on Java Reflection API and Java RMI.

We are currently working on a new implementation of Java// which will take advantage of new JDK 1.2 features (suppression of language access control checks, Reflection and RMI enhancements, weak references,...) as well as take into account deprecated parts of the thread API.

We're also working on an implementation of the Salishan problems [9] as a test bed.

Java// is available for download along with source code, documentation and examples at `http://www.inria.fr/sloop/javall`.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, USA, May 1996.

[2] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *European Conference on Object-Oriented Programming (ECOOP'87)*, pages 32–40. Springer-Verlag, LNCS 276, 1987.

[3] Denis Caromel. Service, Asynchrony, and Wait-By-Necessity. *Journal of Object Orientated Programming (JOOP)*, pages 12–22, November 1989.

[4] Denis Caromel. Programming Abstractions for Concurrent Programming. In *Technology of Object-Oriented Languages and Systems, PACIFIC (TOOLS PACIFIC '90)*, November 1990.

[5] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.

[6] Denis Caromel, Fabrice Belloncle, and Yves Roudier. *The C++// System*. MIT Press, 1996.

[7] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35:44–152, 1992.

[8] Geoffrey C.Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modelling. 1996.

[9] John T. Feo. *A comparative study of parallel programming languages: the Salishan problems*, volume 6 of *Special topics in supercomputing*. North-Holland Publishing Co., Amsterdam, The Netherlands, 1992.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Professional Computing Series. AW, 1995.

[12] James Gosling and H. McGilton. *The Java Language Environment*. Sun Microsystems Computer Company, May 1995.

[13] C.A.R Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 10:549–557, October 1974.

[14] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[15] Juergen Kleinoeder and Michael Golm. Metajava: An efficient run-time meta architecture for java. Techn. Report TR-I4-96-03, Univ. of Erlangen-Nuernberg, IMMD IV, 1996. english.

[16] Doug Lea. *Concurrent programming in Java: design principles and patterns.* Addison/Wesley Java series. Addison-Wesley, Reading, MA, USA, November 1996.

[17] Michael Philippsen and Matthias Zenger. Javaparty - transparent remote objects in java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[18] Rajeev R. Raje, Joseph I. William, and Michael Boyles. An asynchronous remote method invocation (armi) mechanism for java. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.

[19] W. M. Yu and A. L. Cox. Java/DSM: a platform for heterogeneous computing. In *ACM 1997 Workshop on Java for Science and Engineering Computation*, June 1997.