Lazy Argument Passing in Java RMI

Christopher Line, K. R. Jayaram, Patrick Eugster
Purdue University, West Lafayette, USA
{caline,jayaram,p}@purdue.edu

ABSTRACT

Though often criticized for its inherent synchronization overhead and coupling, the *remote method invocation* (RMI) paradigm remains one of the most popular abstractions for building distributed applications. Many authors have suggested ways to overcome its drawbacks focusing mostly on the invoker's perspective, for example by multiplexing invocations to replicated server objects, through "future" return values, or even by prohibiting return values altogether. The more global perspective, and in particular the invokee side, has conversely only received little attention.

In this paper we take a fresh look at the RMI paradigm, elaborating on argument passing semantics. We identify three *lazy* ways of passing arguments by value, differing by the moment at which the transfer of the arguments synchronizes with the execution of the method body. We present a preliminary library implementation of our argument passing semantics in Java RMI, and illustrate their individual benefits through examples. We characterize analytically and empirically application scenarios which benefit from lazy argument passing.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Client/server*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*

General Terms

Languages, Performance

Keywords

argument passing, method remote, Java, future, invocation, asynchronous, lazy

PPPJ 2008, September 9–11, 2008, Modena, Italy.

1. INTRODUCTION

Oldie but goldie

The *remote method invocation* (RMI) paradigm has been often critcized, in particular because its underlying concept of remote object references can blur address space boundaries, thus covertly introducing direct dependencies across nodes. Invokers and invokees are strongly synchronized in request/reply invocations taking place over those references.

Nonetheless, the RMI abstraction remains to date one of the most popular paradigms for distributed programming. Its success can be ascribed to various conditions. One fact is that the RMI represents a simple abstraction in an object oriented setting, and a straightforward choice. In the end, any interaction in such a setting — remote or local — will result in invocations/executions of methods of some sorts, whether these have a prescribed signature or an application-defined one, whether they take place on a proxy or a special shared space abstraction, etc. An important step to sustaining the RMI's prominence has been the realization that remote invocations, though sharing the same core paradigm as local ones, do not need to appear the same way, but can manifest their specific nature for instance through typing constraints (e.g. restrictions on argument types, return types, exceptions). Now, with early misconceptions about distribution transparency out of the way, the focus can be shifted to the differences between local and remote method invocations, and to making RMI more efficient and better suited for different settings and different systems.

Enhancing RMI

Many sensible improvements have been suggested already. *Future* invocations, for instance, have been proposed for reducing synchrony on the invokee side [19]; asynchronous invocations devoid of return types have been proposed to avoid synchronization altogether [1, 16]; the passing of objects *by value* rather than *by reference* as arguments to remote invocations has been advocated as a way of reducing the tangling of remote references.

However most of those approaches still focus only on parts of the equation. The truth, however is more complex. Neither pass-by-value nor pass-by-reference can stand on their own, and (a)synchrony is more than just a client-side issue. As intuition suggests, not all objects passed as arguments to remote invocations are either very "small" objects that are immutable and need no synchronization on them (ideal for pass-by-value semantics), or "large" and location-dependent ones (pass-by-reference semantics). There are

many grayscales: objects with an intermediate size, such that the overhead of transferring them by value to a invokee would have to be balanced against the intensity, semantics, and their use by that invokee. And this exercise would potentially have to be repeated for every invokee for a same object. In this paper, we thus take a second look at RMI. We elaborate on object passing semantics, advocating for more flexibility in the choice of these semantics, rather than proposing a new, all-encompassing approach. We discuss the benefits of both pass-by-value and pass-by-reference semantics, and illustrate the need for intermediate solutions.

Lazy RMI

We have identified several scenarios in which a *lazy* transfer of arguments to remote method invocations yields benefits. We propose in this paper several variants of such *lazy argument passing*, depending mainly on how the transfer of such an argument is synchronized with the corresponding method body on the invokee:

On demand: With on-demand lazy argument passing, an argument is only transferred when the invokee explicitly requests its transfer. This allows an object to access another, remote, object by reference first, and to download it at need; this provides a migration mechanism, and a way to deal with idempotent invocations, e.g. as part of a callback.

On access: An argument passed with on-access semantics is automatically transferred upon its access on the invokee side. If never accessed, it will never be transferred. Similar to on-demand lazy arguments , on-access arguments provide a means of dealing with idempotent invocations. Transparency is however further increased by trading flexibility.

Imperative: If an argument is passed with imperative lazy argument passing semantics, then its transfer is immediately triggered just like with regular pass-by-value semantics. The method body may however start executing on the invokee before the transfer has completed. This leads to a future-like semantics, which is best exploited by a programmer by using corresponding arguments as late as possible in the method body.

While hints to schemes for obtaining similar effects as with lazy argument passing can be found in literature (e.g., [11, 15]), this paper, to the best of our knowlege, is the first to present a thorough study of its motivating scenarios, principles and implications.

Contributions

After reviewing traditional argument passing semantics, we describe our three variants of lazy argument passing in a general context, and compare them with other flexible parameter passing models. We discuss different ways of expressing lazy argument passing in Java, and present an implementation in Java RMI. We illustrate our flexible semantics through examples, focusing on the *semantics and syntax* of lazy parameters as a means of providing an additional degree of freedom.

We then present a preliminary performance analysis of imperative lazy arguments, establishing constraints for applications to experience speedups when using such arguments. Throughout the paper, we compare and contrast our contributions and proposals to other argument passing mechanisms.

Roadmap

Section 2 reviews argument passing models in distributed object systems. Section 3 motivates and introduces the three variants of our lazy argument passing model. Section 4 outlines its implementation in Java RMI. Section 5 illustrates the use of on-demand lazy arguments. Section 6 presents a performance evaluation of imperative lazy arguments. Section 7 concludes the paper.

2. BACKGROUND: ARGUMENT PASSING

This section recalls the classic *pass-by-value* and *pass-by-reference* argument passing models in distributed settings, and discusses previous improvements to these models. This exercise is made in a general context. To illustrate argument passing models, consider an object-oriented application containing a nested sequence of method invocations by invokers on invokees according to the UML diagram of Figure 1(a).

2.1 Pass-by-Value Semantics

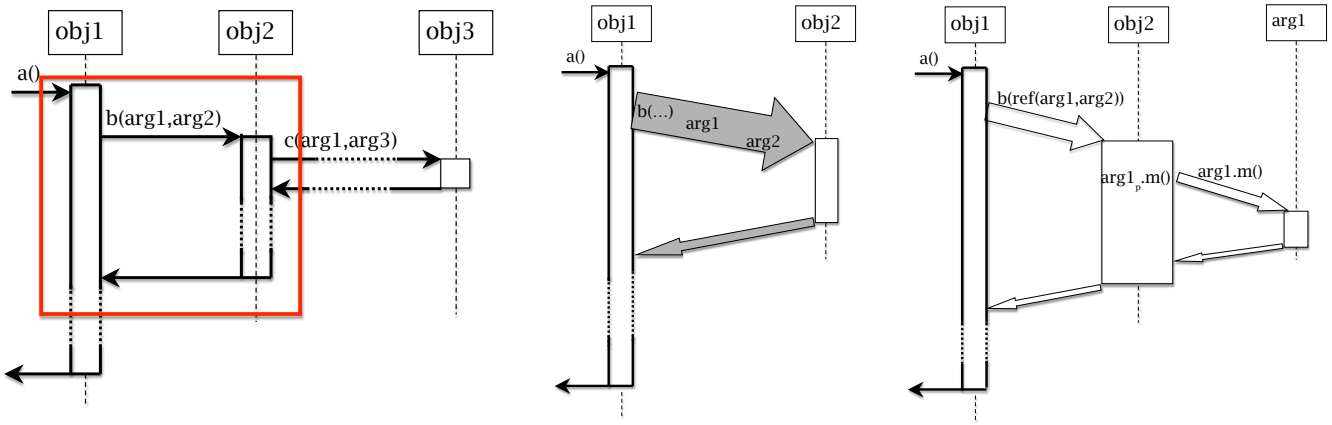
We focus on a single such invocation (see Figure 1(b)), which defies the *logical* nature of UML but reflects reality: the transfer of the marshaled invocation exhibits a certain latency (arrow is not horizontal), and does not take place instantaneously (arrow is more than a line). This is particularly flagrant with *pass-by-value* semantics, as an object `arg1` passed as argument is copied with its state (dark grey arrow) from the source address space to the target address space(s). For a target space, this leads to the creation of a clone in that space, i.e., a distinct object $\texttt{arg1}_c$ whose initial state is the same as the one of the original object at the moment the interaction was triggered. Any method invocation on the transfered `arg1`, for example `arg1.m()` is performed at the target site. The return value of method `b()`, if any is also passed by value. Some limitations of pass-by-value semantics are:

VS-NS (No Synchronization): Since any action performed on a clone on a target site has no effect on the original object, passing arguments by value does not *inherently* allow synchronization between hosts. (Synchronization can be achieved separately.)

VS-ET (Excessive Transfer): Repeatedly passing the same objects by value, or passing objects (possibly with their code) and in the end using only "small parts" of them, if at all, can waste network resources. This becomes particularly flagrant when not only the state of the object is transferred, but also the corresponding code.

2.2 Pass-by-Reference Semantics

With *pass-by-reference* semantics, as depicted by Figure 1(c), an object `arg1` passed as argument from one address space to another is most commonly incarnated on the target site as a *proxy* $\texttt{arg1}_p$. Transfer by reference is indicated by a white arrow. The proxy $\texttt{arg1}_p$ mimicks the original object `arg1`, forwarding any method invocation `m()` performed on it to `arg1`. The return value of such an invocation is also passed by reference (white arrow).



(a) Sequence of method invocations (b) Pass-by-value semantics (c) Pass-by-reference semantics

Figure 1: Classic argument passing models. Figure 1(a) shows a sequence of method invocations from `obj1` to `obj2` to `obj3`. The red box indicates the invocation used to illustrate parameter passing. Figure 1(b) shows `arg1` and `arg2` being copied by value from `obj1` to `obj2`'s address space (grey arrows).Figure 1(c) shows `arg1` and `arg2` being passed by reference (indicated by white arrows) `obj1` already holds these references before passing them to `obj2`. So a call to `arg1.m()` is dispatched object `arg1`, which may not be in either `obj1` or `obj2`'s address space.

Proxies, and thereby remote method invocations (RMIs), are intrinsically tied to pass-by-reference semantics, because there is no point in passing around first class references to objects, if one can not "use" (invoke) these.

The RMI paradigm, and thus pass-by-reference semantics, have been the subject of many debates around reliability and efficiency (e.g., [5, 13]):

RS-EC (Execution Coupling): Through bi-directional interaction between clients and remotely invoked objects, increased latency of a component can affect other components through cascaded invocations.

RS-FC (Failure Coupling): By creating links between proxies and original objects, the RMI abstraction increases dependencies. The failure of a single component might crash an entire application.

RS-FH (Failure Hiding): RMIs abstract over physical distribution, and in particular, over communication and host failures, making it impossible to deal with these realistic phenomena.

RS-EI (Excessive Invocation): Every invocation performed over the network incurs an overhead (compared to a local one).

2.3 Advanced Argument Passing Models

Based on arguments RS-EC..EI, early related work on argument passing emphasized on (1) reducing or entirely eliminating RMIs, and (2) diminishing VS-ET, i.e., the amount of data (unnecessarily) transferred by value.

2.3.1 *Adaptive argument passing*

Lopes [13] suggests programmer support (through a graphical editor) to define the parts of objects passed by value that are actually used on target sites, and proposes to transfer only those parts. This approach assumes that in all contexts in which instances of a given type `T` are transferred, the corresponding targets all require the exact same parts of the received objects of type `T`.

2.3.2 *Argument substitution*

The system described in [5] refrains from transferring an object `arg1` value from a source space to a target space in case the latter space already hosts a copy $\texttt{arg1}_c$ of that same logical entity (received earlier). This can lead to inconsistencies if the state of that copy in fact has been modified since it was created, for instance if `arg1` is modified between the creation of $\texttt{arg1}_c$ and the second time it is transferred. Focusing on performance here can be delusive; defining whether objects are immutable, or what modifications or inconsistencies can be tolerated, becomes an important matter.

3. LAZY ARGUMENT PASSING

This section presents the semantics of our lazy pass-by-value model, and discusses these in the face of other argument passing models, and also of other asynchronous interaction paradigms in distributed object settings.

3.1 Values vs References

After long debates about the pros and cons of argument passing models, there seems to be a clear need for both.

3.1.1 *No one size fits all*

On the one hand, pass-by-value semantics are necessary, and clearly apply well to objects which are small (e.g., numbers), and/or "passive" (i.e., whose methods mainly serve to access their fields, e.g., *events* [9]). On the other hand, objects which represent larger entities (e.g., "services", "components"), possibly with autonomous behavior, are simply better handled by references, especially if they are location-dependent (e.g., databases).

3.1.2 Improving pass-by-reference

Most critiques of the RMI abstraction were due to the way it was put to work in early systems, and have been addressed since. RS-EC is countered by several asynchronous variants of the RMI paradigm which have been adopted in practice (cf. CORBA Messaging [16]), or the concept of *group proxy* which opens the door to other than strict synchronous one-to-one interaction models. Group proxies also enable the replication of critical objects, and the combination with transactions further embraces fault tolerance – acting against RS-FC. RS-FH is countered through current implementations of the RMI abstraction which make distribution explicit, e.g., through specific exceptions of type `RemoteException`. Object caching is often used to deal with RS-EI, and can help also address VS-ET.

3.2 The Best of Both Worlds

Yet, the problem of judicious network resource utilization remains (VS-ET and RS-EI resp.). What model should be used when passing a given object? This question can not simply be answered by defining a threshold on the size of objects, which would solomonically divide the object universe into objects passed by value and objects passed by reference. Transferring a larger component for the purpose of invoking it once wastes network capacities the same way as performing a large number of remote invocations on that same components does, especially if the input arguments and the return values represent massive amounts of data. Also, even a small object might represent a crucial entity and thus require synchronization. The line between value- and reference-passing is hence hard to draw – it can vary even among executions of the same application.

The approach taken in this paper provides more flexibility to the programmer for handling the large family of objects which are between the extremes of string objects and database servers. This support consists of passing these objects as "references" initially to an invokee(see Figure 2), and by value later on, either (1) once the transfer has completed, or (2) at a precise instant, only if really required in the respective context, or (3) when explicitly requested by the invokee.

3.3 Variants

In the following we describe three variants of lazy argument passing, differing mainly in the definition of *when* and *how* to initiate the transfer of an object by value. Table 1 summarizes these choices.

3.3.1 On demand lazy argument passing

With *on-demand* lazy pass-by-value semantics, the program(mer) defines if and when an object `o` is to be transferred by value to the invokee address space. Initially, the arguments are passed by reference as in RMI and are incarnated at an invokee by proxies. In Figure 2(a), the arguments of `b()`, namely `arg1,arg2` are passed by reference. During the execution of `b()`, the programmer explicitly requests a transfer (download) of `arg1` by value, the transfer is initiated, and the thread blocks until `arg1` is received. Before transfer by value, `arg1` is incarnated at the invokee by a proxy $\texttt{arg1}_p$ (with pass-by-reference semantics) and methods can be invoked on $\texttt{arg1}_p$ in an RMI style (cf. Figure 1(c)). This can be done by the program(mer) to acquire more knowledge about that object and decide whether to transfer it by value.

The advantage is clear in P2P queries. When receiving multiple objects as arguments to respective *idempotent* calls to a listener-type interface in reply to a query, for instance, these objects can be further investigated before being selectively downloaded. On demand semantics thus can be used as a *migration* mechanism for remote objects.

3.3.2 On access lazy argument passing

With *on-access* lazy argument passing, an object `o` is similarly first incarnated on a target site by a proxy $\texttt{o}_p$. The first access to `o` through $\texttt{o}_p$ however automatically triggers the transfer of `o` by value. In Figure 2(b), the arguments of `b()`, namely `arg1,arg2` are initially passed by reference. When `arg1` is accessed (method `m()` is called on it), `arg1` is transfered by value. However, the semantics of the proxy $\texttt{arg1}_p$ until `arg1`'s transfer by value are different. The first access on the proxy triggers the actual object's transfer by value. The only "access" which triggers an object's transfer by value is a method invocation on its proxy. $\texttt{arg1}_p$ can be passed by `m()` as arguments to another method, doing so is not considered "access", and does not trigger its transfer by value.

This variant is particularly useful in listener-type interfaces when the first, say n objects received as arguments to invocations are to be used, irrespective of their individual characteristics.

3.3.3 Imperative lazy argument passing

With *imperative* lazy argument passing, an object `o` is again first incarnated on a target site by a proxy $\texttt{o}_p$, but its transfer is immediately initiated together with the invocation. The method body can thus start to execute *before* the object is fully tranferred, with the proxy taking the place of the upcoming actual argument. Imperative lazy argument passing can thus be viewed as the server-side equivalent to *future* invocations (see below); an access to such a proxy leads to blocking if the corresponding object has not been received yet. In Figure 2(c), the arguments `arg1,arg2` are immediately transfered by value after the method call.

This argument passing semantics thus has immediate performance benefits with larger arguments being passed to remote method invocations, and, alike future invocations, is best exploited if corresponding arguments are accessed as late as possible in the method body. Methods which need to "prepare" for computation, for example by synchronizing (creating a transaction, acquiring a lock), by initiating more remote connections, or by retrieving information from a database to handle the request, are ideal candidates for this. The operations can be performed while the arguments are being transferred.

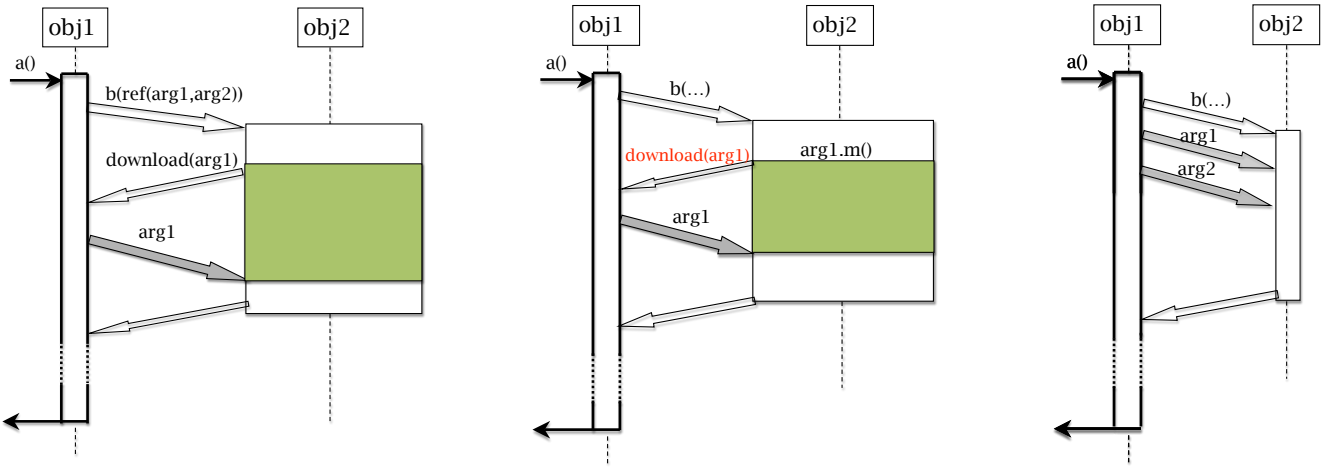
It is important to note that in all the three variants of lazy argument passing, the argument passed lazily is originally incarnated as a proxy, but the semantics of this proxy differs in each variant.

3.4 In Perspective

We discuss here lazy argument passing in the face of related argument passing models and of future invocations.

3.4.1 Vs alternative argument passing models

The two improvements for pass-by-value semantics analyzed in Section 2.3 can be viewed as orthogonal with respect



(a) On demand lazy argument passing

(b) On access lazy argument passing

(c) Imperative lazy argument passing

Figure 2: Lazy arguments vs lazy invocations. Grey arrows illustrate transfer by value, while white arrows illustrate transfer by reference. In Figure 2(b), `download(arg1)` is implicitly called when `arg1` is accessed. The shaded region in `obj2` indicates that the thread of execution is blocked, until `arg1` is transfered by value.

		Transfer trigger (how)	
		Implicit	Explicit
Transfer point (when)	Immediate	imperative	–
	Deferred	on-access	on-demand

Table 1: Overview of lazy argument passing semantics.

to each other, and also as orthogonal to our proposal. On demand and on-access lazy argument passing namely define *if and when exactly a given object should be passed by value*. Provided that an object should be passed by value, [5] specifies *whether it is actually necessary to pass the value of that object*. In case some "valid" copy of the object is already present on the target site, that one can be used. Otherwise, [13] aids in deciding *what exactly then the value to be passed is*, i.e., what parts of the object's representation are to be transferred. Imperative lazy argument passing only defines whether or not a method body may start to execute before all the corresponding arguments have been received.

3.4.2 Vs future invocations

Lazy argument passing share characteristics with *future (method) invocation* [19] originally introduced in MultiLisp [10] and used to repress RS-EC (and RS-FC), but are not identical to those. *Imperative* lazy arguments are in fact similar to the *future* objects returned by *implicit* future method invocations [3], except that they do not result from an explicit invocation on the invoker site but from the execution of a method body experienced on the invokee side.

Akin to implicit futures, imperative lazy argument passing thus hides the decoupling from the programmer to a large extent; i.e. the code does not reflect decoupling, though a programmer should be aware of it to best exploit it: while an implicit future object should be used *as late as possible* to make sure the corresponding value has been computed and transferred in the meantime, an object `arg1` (Figure 2) passed by value lazily should be used *only if necessary*.

Since future invocations and lazy arguments are orthogonal, they can be combined. A lazy passed argument can be invoked in a future style, yielding a future object in case the argument has not yet been transferred.

3.4.3 Synchronization and exceptions

The possibility that the state of an object transferred lazily by value alters in between its initial "reception" and the effective transfer must be kept in mind by programmers, but is not further elaborated here as it is also encountered with future invocations [4]. Similarly, exceptions may appear outside of their original scope. Any solution for dealing with these issues in the presence of future invocations [2, 17, 12, 6] can be straightforwardly applied with lazy arguments. Replication techniques can be applied for ensuring availability of lazily transferred objects [8, 14]. In fact, specific protocols have been proposed for handling large "messages" [7], which is a case for imperative lazy arguments.

4. IMPLEMENTING LAZY ARGUMENTS

We present here a simple way of putting explicit and implicit lazy argument passing to work in Java, in a way following the spirit of the existing support for remote argument passing in Java RMI.

4.1 Expressing Lazy Arguments

Objects passed lazily (instances of the `Lazy` type in Figure 3) are both `Serializable` and `Remote` The root `Lazy` type represents imperative lazy semantics. Any lazy arguments can be verified for their locality, or can be dereferenced to obtain the actual object (see Section 4.3).

An imperative lazy argument is represented by the `Imperative` interface and an on-access lazy argument is represented by the `OnAccess` interface. Instances of the

OnDemand type can, in addition be explicitly transfered by value by invoking the download method.

At first glance one might expect OnDemand to be the only one to subtype Remote, as only such objects can be invoked at a distance as well. However, since any invocation of a lazy argument might yield a remote exception (see Section 4.4), the Remote interface is inherited along with its constraints for exceptions by all Lazy types.

No subtyping relationship is present between the different variants of lazy argument passing. Introducing such relationships could lead to simplifying the design, but would lead to inconsistencies as the semantics would not be preserved.

Class LazyHandling offers the same methods of the Lazy type and its subtypes type for convenience. Additional methods are provided, spanning both blocking and non-blocking versions. Similarly, the subtypes of Lazy offer more methods for convenience, which are not reported for simplicity.

```
public interface Lazy extends Serializable, Remote {
  public boolean isLocal() throws RemoteException {...}
  public Object deref() throws RemoteException {...}
  ...
}

public interface Imperative extends Lazy { }

public interface OnAccess extends Lazy { }

public interface OnDemand extends Lazy {
  public void download() throws RemoteException {...}
  ...
}

public final class LazyHandling {
  public static boolean isLocal(Lazy o)
                throws RemoteException {...}
  public static void download(OnDemand o)
                throws RemoteException {...}
  public static Object deref(Lazy o)
                throws RemoteException {...}
  ...
}
```

Figure 3: A simple approach to lazy argument passing.

Note that since the use of lazy pass-by-value semantics has little direct impact on the implementation of corresponding objects, a backward-compatible solution could consist in accepting any object which is both Serializable and Remote as lazy pass-by-value object.

4.2 A Library-based Implementation

On the invoker-side, if an argument implements the Lazy interface, our implementation has code to create a proxy for it and marshall it to the invokee.

On the invokee side, with the scheme presented in Figure 3, an object arg1 in Figure 2 passed lazily is first represented, i.e., incarnated, in the target address space as a proxy $\texttt{arg1}_p$.

In on-demand lazy argument passing, the invokee calls arg1.download() to get arg1 by value.

In on-access lazy argument passing, the first method invocation on $\texttt{arg1}_p$ is intercepted, and its transfer by value is initiated. The calling thread blocks until the transfer is complete.

If the lazy argument passing is imperative, the transmission of arguments to the invokee starts immediately. Our implementation blocks access (via the proxy $\texttt{arg1}_p$) to imperative lazy arguments until they are fully transfered.

Note that the invoker might not host the lazy argument, the invoker might only have a reference to it (proxy). So the invoker is not always responsible for the eventual transfer of the lazy argument by value. The proxy has information about the actual host of the object, and the download method of LazyHandling class takes care of sending messages to that host to transfer the lazy argument by value.

Without hooks into the Java runtime (i.e., the virtual machine), the object that a variable is pointing to can not be swapped against another object. Thus, a proxy can not be swapped, transparently to all local references to it, against the actual object it represents (e.g. $\texttt{arg1}_p$ cannot be swapped with $\texttt{arg1}_c$ after transferring that object arg1 by value). In other terms, after downloading an object, that object is still accessed, indirectly, via the proxy $\texttt{arg1}_p$. Method deref has been added to class LazyHandling to offer the possibility of obtaining the referenced object when desired. Such a scheme based on proxies makes the handling of *aliasing* straightforward. More precisely, if an object is passed for *two or more* formal arguments in the context of a same interaction, or in different interactions between the same target and source sites, the first transfer of the object by value also leads to linking the copy created on the target site to *all* respective proxies.

4.3 Dynamic Proxies

The proxies outlined above are in fact implemented as *dynamic proxies* [18], in order to avoid using a precompiler à la rmic and better integrate with Java RMI – itself now endorsing such dynamic proxies.

In short, a dynamic proxy is an object which conforms to a set of interfaces, for which (the class of) that proxy was created through class java.lang.reflect.Proxy outlined in Figure 4. A dynamic proxy can be cast to (and used as an instance of) any of the interfaces it was created for and invoked accordingly. An invocation m made on such a dynamic proxy object is however *reified* and passed to the generic invoke method of the InvocationHandler associated with the proxy, stepping from a statically typed context to dynamic interaction where *any* actions can be performed in the confines of a method invocation. Figure 5 presents an overview of the interaction with a dynamic proxy and its invocation handler;

Dynamic proxies represent a straightforward means to implement the download and isLocal methods of lazy arguments in a *delegator* pattern style. To help avoiding overhead when using lazily transferred objects, the deref method allows to access directly the object shielded by the proxy after the transfer; avoiding such proxies completely would require support in the virtual machine for lazy arguments.

4.4 Failures

An object transfer, like any remote interaction, can suffer from a failure in the underlying distributed infrastructure. A failure during an explicit download of an object is reflected by throwing a specific kind of RemoteException (package

```
public interface InvocationHandler {
  public Object invoke(Object proxy, Method method,
                       Object[] args) throws Throwable;
}

public class Proxy implements Serializable {
  protected InvocationHandler h;

  protected Proxy(InvocationHandler h) { this.h = h; }

  public static
    InvocationHandler getInvocationHandler(Object proxy)
      throws IllegalArgumentException {...}
  public static Class getProxyClass(ClassLoader loader,
                                    Class[] interfaces)
      throws IllegalArgumentException {...}
  public static
    Object newProxyInstance(ClassLoader loader,
                            Class[] interfaces,
                            InvocationHandler handler)
      throws IllegalArgumentException {...}
  ...
}
```

Figure 4: Types `InvocationHandler` and `Proxy`.

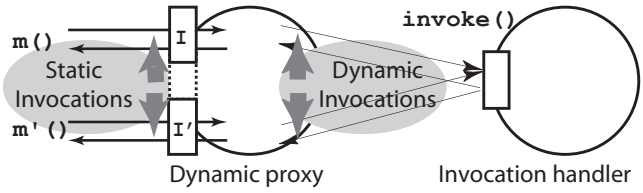


Figure 5: Interacting with dynamic proxies.

`java.rmi`). When implicitly transferring an object by value upon its invocation, the same exception can be thrown as "result" of the invocation. Methods of objects passed by value in a lazy style must hence, like those of any `Remote` objects, reflect these exceptions in their signatures.

5. ILLUSTRATION

Consider the (in)famous scenario of songs being shared throughout the Internet. Figure 6 is a part of an music sharing application implemented with lazy on-access argument passing. Figure 6 proposes a Java type `Song` for incarnating such objects. An instance of `Song` conveys information about the title, artist, album, year and the actual song bytestream. Songs are hosted by `SongServers` which are not necessarily huge servers but are peers sharing their songs. `Song` is a lazy argument. When a song is added to a `SongServer`, it is uploaded lazily to a lookup server (`MusicLookupServer`), which doesn't store the song, but only a reference to it. `Tags` is a class which contains metadata associated with a song, like artist, genre, year etc., and is stored in `MusicLookupServer` along with a `Song`. Tags are used for searching songs in a library. A client can search the `MusicLookupServer` by specifying a subset of the fields in the `Tags` class, and the corresponding `Song` objects are returned by the lookup server (lazily). Only when the `Song` is about to be played by the Jukebox (the bytestream of the `Song` is obtained by calling `getByteStream`), it is actually downloaded. Figure 6 shows a simple application and omits a lot of details (like exception handling, implementation the actual search by tags, adding songs to a playlist, ordering a playlist etc.) for brevity and space constraints.

Figure 7 how parts of the same application change when lazy on-demand argument passing is used.

6. EVALUATION

In this section we present initial performance evaluations of lazy arguments in Java RMI. We focus on the case of imperative lazy arguments in order to evaluate the conditions under which these can yield benefits.

6.1 Analysis

The latency experienced by a invoker when invoking an invokee with a standard synchronous semantics can be roughly broken down as expressed below by equation 1:

$$L_{\square} = T + E + R \tag{1}$$

The latency is the sum of (a) the time T it takes to trigger the method on the invokee, including the transfer of all arguments (b) the time E for the actual method body execution, and (c) the time R it takes for the return value to be transferred back to the invokee.

In a simple lazy imperative argument passing scenario we can approximate the invocation latency as expressed below by equation 2:

$$L_{\diamond} = T_{hd} + max(T_{arg}, E_{\ominus arg}) + E_{\oplus arg} + R \tag{2}$$

Here, the latency is given as the sum of (a) the time T_{hd} for transferring the method *header* (excluding lazy arguments), (b) the larger of (b.1) the time T_{arg} to transfer the lazy arguments and (b.2) the time $E_{\ominus arg}$ to execute the initial portion of the method not relying on the lazily transferred arguments, (c) the time $E_{\oplus arg}$ for the execution of the remaining method body making use of the lazy arguments, and (d) the time R it takes for the return value to be transferred back to the invokee.

The link between the two latencies $L_{\square}$ (equation 1) and $L_{\diamond}$ (equation 2) is given by the fact that $T = T_{hd} + T_{arg}$ and $E = E_{\ominus arg} + E_{\oplus arg}$. Thus, the following equation

$$L_{\square} > L_{\diamond} \quad \Leftrightarrow \quad T_{arg} + E_{\ominus arg} > max(T_{arg}, E_{\ominus arg}) \tag{3}$$

expresses the conditions under which imperative lazy argument passing becomes beneficial for a given method. This condition is always trivially satisfied, but as we will see below is not fully accurate.

6.2 Measurement

The advantages of imperative lazy argument passing were assessed by comparing the completion time of a single method invocation with lazy arguments with an identical invocation without lazy arguments. Two experiments were conducted; the first evaluated the overall performance of lazy argument passing when utilizing arguments of varying size, and the second measured the level of overhead added by the implementation.

6.2.1 Setting

Both experiments were run in JDK1.6. All client/server interactions took place on machines running Ubuntu 7.10,

```
public class Song implements OnAccess{
        private Tags t;
        private byte[] data;
        public byte[] getByteStream() {
                return data;
        }
        ...
}

public class Tags {
        private String name;
        private String artist;
        private String genre;
        private String year;
        ...
}

public class MusicLookupServer {
        private MusicLibrary lib;
        public void upload(Tags t,Song s) {
                lib.add(t,s);
                ...
        }
        public Song[] getSongs(Tags t) {
                lib.get(t);
        }
        ...
}

public class SongServer {
        private MusicLookupServer lookup;
        private HashTable<Key,Song> library;

        public void add(Song s) {
                Key k = getKey(s);
                library.add(k,s);
                Tags t = getTags(s);
                lookup.upload(t,s);
        }
        ...
}

public class MusicClient {
        private Jukebox j;
        private MusicLookupServer s;
        public void getSongs(Tags userPrefs) {
                j.add(s.getSongs(userPrefs));
        }
        ...
}

public class Jukebox {
        //to illustrate on-access parameter passing
        private MP3Player m;
        public void play(Song s) {
                m.play(s.getByteStream());
                //This is the point at which the song gets
                //transfered by value
        }
        ...
}
```

Figure 6: Exchanging Music with lazy on access parameter passing.

```
public class Song implements OnDemand {
        // The class is the same as Figure 6
}

public class OndemandJukebox {
        //to illustrate on-demand parameter passing
        private MP3Player m;
        public void play(Song s) {
                LazyHandling.download(s);
                m.play(s.getByteStream());
        }
        ...
}
```

Figure 7: Exchanging Music with lazy on demand parameter passing, showing how parts of Figure 6 change.

with single-core 2.2 MHz processors and 1 GB of RAM. Interactions took place over a 100 Mbps, 51ms latency network.

6.2.2 Overhead

Due to the nature of the implementation, imperative lazy argument passing incurs an overhead beyond that which is normally associated with Java RMI due to proxy declaration, context switches and additional setting up of sockets needed for transferring lazy arguments. This was tested using a simple remote method invocation that would reveal overhead for three separate scenarios (see Figure 8): lazy argument passing, non-lazy argument passing in the Java RMI implementation including support for lazy arguments, and argument passing using unaltered Java RMI. The remote method accepted a single argument and immediately returned. The argument consisted in a single empty object, whose class implements the `Lazy` interface..

Without any checks for this interface, the overhead incurred was what would be expected of normal Java RMI method invocation. In the lazy RMI implementation, the `Lazy` interface is checked for in two locations: the client stub, when it is determined if the argument should be passed by value or if a proxy should be sent in its place; and on the server skeleton, when arguments are unmarshaled and, in the event of a lazy argument, a connection is accepted from the client to begin downloading the argument in a separate thread.

Predictably, overhead added for non-lazy arguments in the lazy implementation was minimal. Overhead for lazily passing an object was not insignificant, but as the next evaluation shows, there are many scenarios when lazy passing is advantageous. In fact, equation 3 would need to be added a constant overhead on the right-hand side of the inequality sign.

6.2.3 Latency

The key advantage of imperative lazy argument passing lies in methods that do not immediately access passed arguments. To confirm analytical prediction of Section 6.1, a simple client-server interaction was used to simulate a remote method invocation that performs a certain amount of work before accessing a lazy argument.

To do this, a remote method was defined on the invokee that accepted a single argument. This argument contained an integer array, the dimensions of which were altered to con-

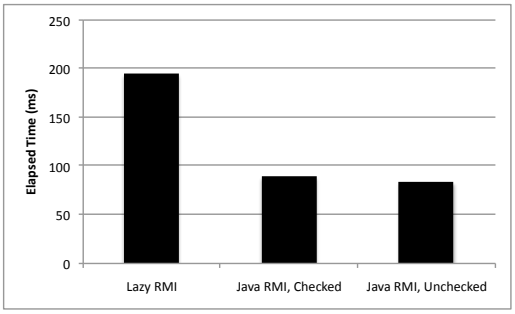


Figure 8: Overhead of lazy arguments in Java RMI.

trol the object's size. Upon method invocation, the server would wait for a predefined period of time before accessing and returning a value based on the contents of the passed argument.

The size of arguments passed varied from 12 B to 160 KB. These arguments were tested with 500, 1000 and 2000 ms of simulated "work" occurring before argument access.

The results showed that, while the overhead incurred by lazily passing very small arguments causes it to be less efficient than unaltered Java RMI, as argument size increases the effectiveness of lazy passing became clear.

Understandably, the time saved by lazy passing has an upper bound dependent on the amount of work performed before argument access, $E_{\ominus arg}$. If one performs a task that takes 500 ms, passing large arguments lazily will not reduce computation time by any more than that 500 ms. In the right situations, however, such a performance increase would be significant (see Figure 9). It falls to the developer to determine where an application's use of remote invocation would benefit from this lazy passing technique, and to arrange the code correspondingly as in the case of future method invocations. Figures 10 and 11 illustrate the potential for reducing latency with $E_{\ominus arg}$ being 1000ms and 2000ms respectively.

7. CONCLUSIONS

Programming applications for *asynchronous* distributed systems is intrinsically hard. "Harmful" characteristics become predominant in settings such as mobile ad hoc networks or large-scale decentralized peer-to-peer settings.

The fundamentals of distributed object-oriented programming have to evolve with these new constraints. This paper can be seen as a step in that direction, focusing on argument passing semantics. We have presented three variants of lazy argument passing, of which two offer increased flexibility, and a third one is mainly geared at reducing latency in the case of large arguments passed. We have illustrated the benefits of the former two variants through a programming scenario, and have provided an analytical and empirical framework to determine situations which can benefit from imperative lazy argument passing.

We are currently in the process of improving our prototype, by making use of thread pools and socket factories in order to avoid the overhead of catering for lazy arguments in our extended implementation of Java RMI. We are furthermore conducting experiments with automatically distributed Java applications; the byte code translations used for automatic partitioning will be extended to rearrange

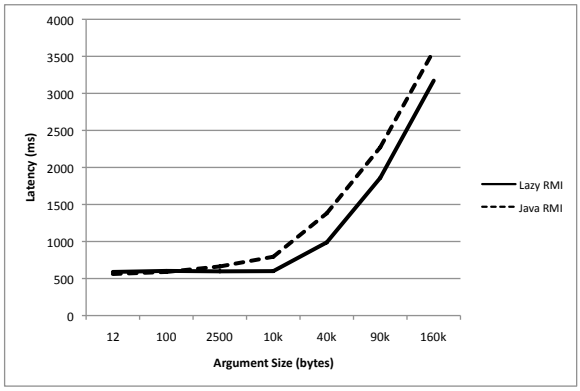


Figure 9: Decreased latency with imperative lazy arguments in Java RMI with 500ms for $E_{\ominus arg}$.

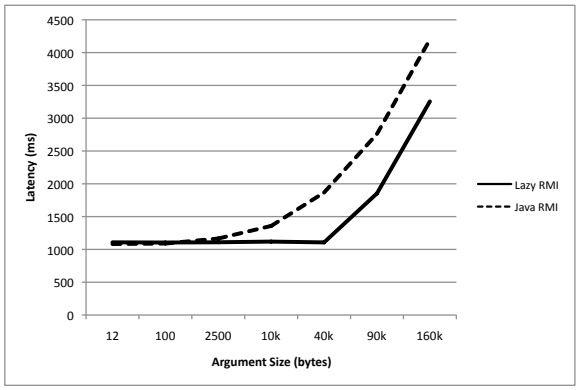


Figure 10: Decreased latency with imperative lazy arguments in Java RMI with 1000ms for $E_{\ominus arg}$.

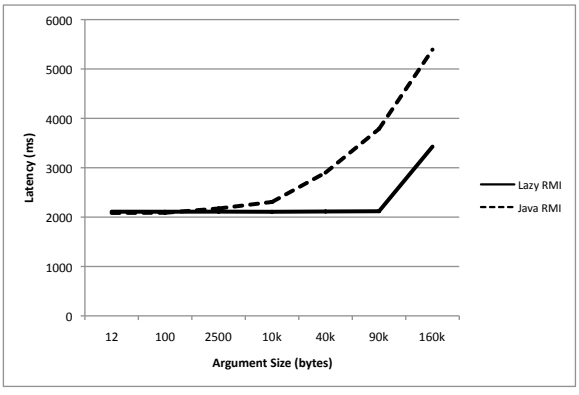


Figure 11: Decreased latency with imperative lazy arguments in Java RMI with 2000ms for $E_{\ominus arg}$.

code to exploit imperative lazy parameters. Last but not least, an integration of lazy arguments with future invocations will provide additional performance benefits.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for high quality feedback which helped improve this paper.

9. REFERENCES

[1] J.-P. Briot. Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP '89)*, pages 109–129, July 1989.

[2] P. A. Buhr and W.Y.R. Mok. Advanced Exception Handling Mechanisms. *IEEE Transactions on Software Engineering*, 26(9):820–836, September 2000.

[3] D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36:90–102, September 1993.

[4] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and Deterministic Objects. In *Conference Record of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '04)*, pages 123–134, January 2004.

[5] M. Mira da Silva, M.P. Atkinson, and A.P. Black. Semantics for Parameter Passing in a Type-Complete Persistent RPC. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems (ICDCS '96)*, pages 411–418, May 1996.

[6] C. Dony, J.L. Knudsen, A. Romanovsky, and A. Tripathi, editors. *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2006.

[7] R. Ekwall and A. Schiper. Solving Atomic Broadcast with Indirect Consensus. In *Proceedings of the 7th IEEE International Conference on Dependable Systems and Networks (DSN 2006)*, pages 156–165, June 2006.

[8] R. Friedman and A. Kama. Transparent Fault-tolerant Java Virtual Machine. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems (SRDS'03)*, pages 319–328, October 2003.

[9] T. Harrison, D. Levine, and D.C. Schmidt. The Design and Performance of a Real-Time CORBA Event Service. In *Proceedings of the 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97)*, pages 184–200, October 1997.

[10] R.H. Halstead Jr. MULTILISP: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[11] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6:109–133, February 1988.

[12] A. W. Keen and R. A. Olsson. Exception Handling during Asynchronous Method Invocation. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing (Euro-Par '02)*, pages 656–660, August 2002.

[13] C.V. Lopes. Adaptive Parameter Passing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software (ISOTAS'96)*, February 1996.

[14] J. Napper, L. Alvisi, and H. Vin. A Fault-tolerant Java Virtual Machine. In *Proceedings of the 4th IEEE International Conference on Dependable Systems and Networks (DSN 2003)*, pages 425–434, June 2003.

[15] P. Nienaltowski, V. Arslan, and B. Meyer. Concurrent Object-oriented Programming on .NET. *IEE Proceedings Software, Special Issue on ROTOR*, 150(5):308–314, October 2003.

[16] OMG. *The Common Object Request Broker: Architecture and Specification, version 3.0*. OMG, December 2002.

[17] A. B. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*. Springer-Verlag, June 2001.

[18] Sun. *Dynamic Proxy Classes*, 1999.

[19] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. 4: Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1. In *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, April 1987.