Lazy Parameter Passing*

P. Eugster**

Sun Microsystems

Abstract. Peer-to-peer (P2P) systems are strongly decentralized and asynchronous distributed settings involving a potentially large number of hosts. A common programing model for P2P infrastructures is that of a distributed hashtable (DHT), though which peers share resources they host. On the quest for a corresponding object-oriented high-level programming abstraction, a core question has turned out to be how to represent and handle resources, such that they could capture the broad variety of P2P applications.
Answering this question has brought us to revisit parameter passing semantics. This papers presents lazy parameter passing, a parameter passing model which combines the best of pass-by-value and pass-by-reference semantics, and is not limited to P2P settings, but is appealing for any asynchronous distributed object setting.
We have put lazy parameter passing to work in a high-level general abstraction for P2P programming, with prototypes for Java and .NET. In this paper, we illustrate the principle of lazy parameter passing through former prototype. We depict the role of Java's concept of dynamic proxies in the implementation of our novel parameter passing semantics, and other distinctive features of our abstraction, and present a general extension to that concept whose scope exceeds by far the context given by our abstraction.

keywords: peer-to-peer, distributed, asynchronous, parameter passing, dynamic proxies

1 Introduction

The recent success of so-called *peer-to-peer* (P2P) infrastructures has once more reveiled the massive amount of resources spread in a completely decentralized manner throughout the Internet, and the potential behind an infrastructure for providing the means of discovering and using these resources.

Towards a high-level abstraction for P2P programming. The model considered by such infrastructures for P2P computing (e.g., Can [22], Chord [23]) is that of a *distributed hashtable* (DHT), where peers can add their own resources in order to share them with others, and query for resources they are in need of.

* Financially supported by the European ESPRIT project PEPITO (Peer-to-Peer: Implementation and TheOry) under grant 5005-67322 (`http://www.sics.se/pepito/`).

** Former affiliation: Swiss Federal Institute of Technology in Lausanne.

High-level abstractions for P2P programming providing the asynchrony mandated by the strongly decentralized and dynamic nature of P2P settings are however still missing. Current practices include indeed useful schemes, yet are either targeted at a specific kind of application [10, 1], or are for the average use overly complex (e.g., forcing the programmer to view resources in raw XML format, and to employ complex lookup mechanisms for finding these resources [11]).

This observation has motivated our quest for a general high-level abstraction for P2P programming, similarly to (1) the *tuple space* for the distributed shared memory model of parallel and distributed systems, or (2) the *remote procedure call* and *publish/subscribe* and abstractions for one-to-one, and one-to-many (i.e., multicast), interaction in a message passing model respectively. More precisely, we started looking for a general object-oriented abstraction at the same level as the (1) *object* space (e.g., [18]), the (2) remote *method invocation* (RMI, e.g., [27]), and the (3) *type-based* publish/subscribe (e.g., [13]) paradigms respectively.

P2P Resources. When viewing resources in a P2P system as objects [7] — whether these resouces are *inodes* in distributed file systems [1] (of which the (in)famous MP3 file sharing systems would only represent a particular instance), or computing resources for peer-based distributed *grid* [10] computations, an important question becomes *how to pass (and access) these objects* between (from) remote address spaces.

This question has led us to revisiting object passing semantics in distributed systems. On the one hand, files downloaded by peers would be logically *passed by value*. On the other hand, processing resources for grid computing, could not be passed by value, but would be *passed by reference*, i.e., accessed by RMIs through proxies. Though the use of latter semantics for passing objects in distributed systems has been largely critized in the past (e.g, [20, 17]), it would seem to be the only reasonable way of capturing resources representing entire "services", whose transfer by value would in certain cases be simply infeasible, or appear disproportionate should they involve a large state and only a single invocation be performed on them on a target site.

The best of both worlds. But intuition suggests that not everything can be black or white; resources are not only either "small" objects that 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: resources with an intermediate size, such that the overhead of transferring them by value to a client would have to be balanced against the intensity and semantics of their use by the client. And this exercise would potentially have to be repeated for every client of a same resource.

This argument is valid for any asynchronous distributed object setting, but develops its full flavor in P2P applications, where peers interact with asynchronous query mechanisms: client peers express queries, and due to the absence of centralized knowledge, are subsequently passed a potentially large number of

resources matching their queries. Not the entire set of these resources is then effectively used; one can investigate these first, or grab the first n ones.

This is precisely where *lazy parameter passing* kicks in: objects are first passed by reference. At choice, and only if necessary, these objects are then either transferred by value (1) *implicitly*, i.e., upon first use (first invocation), or (2) *explicitly*, i.e., at a point dictated by the application. Intuitively, just like a lazy RMI (i.e., *future invocation* [31]) better captures the asynchronous nature of the Internet in pairwise client/server interaction, lazy parameter passing (*lazy pass-by-value* semantics) better captures multi-party interaction, typical of self-organized P2P settings.

Contributions. After revisiting parameter passing models in distributed settings and pointing out the need for more flexible models, this paper advocates for specific support for lazy parameter passing. While hints to schemes for explicitly transferring remote objects by value can be found in literature (e.g., [19]), this paper is, to the best our knowlege, the first to present a thorough study of motivating scenarios, principles and implications of lazy parameter passing.

We describe two flavors of lazy parameter passing in a general context, and compare them with related work on flexible parameter passing, and with future invocations. We then discuss different ways of expressing lazy parameter passing in Java, putting emphasis on a solution retained for supporting the first class notion of *resource objects* in our high-level abstraction for P2P programming called *borrow/lend* (BL) [7]. The BL abstraction is a general abstraction in the sense that it unites flavors of several established abstractions and could be built on top of most common DHTs, yet manifests (P2P-) specific characteristics, and has been used to validate protocols of our own (e.g., [8]) in prototypes for Java as well as .NET [2].

We present how we have made use of *dynamic proxies* [26] to implement not only lazy parameters, but also future invocations, (different levels of) structural conformance, and the type-safe expression of resource queries in Java itself, in our BL abstraction. Together with an example of the use of (lazy parameter passing with) our BL abstraction, we point out an inherent weakness of dynamic proxies, which does not only affect the implementation of the above-mentioned features of our BL abstraction, but is of more general nature. In the context of the implementation of lazy parameter passing in our BL abstraction, we hence focus on different extensions we made to the dynamic proxy mechanism. This includes a general scheme for transforming field accesses to invocations of automatically generated access method at class loading.

Roadmap. Section 2 reviews parameter passing models in distributed object systems. Section 3 motivates and introduces the two variants of our lazy parameter passing model. Section 4 discusses ways of expressing this model in a language such as Java. Section 5 presents our BL abstraction, and its use of dynamic proxies. Section 6 presents our extension to the dynamic proxy mechanism. Section 7 concludes the paper.

2 Revisiting Parameter Passing Models

In this section we recall the pros and cons of the classic parameter passing models, i.e, *pass-by-value* and *pass-by-reference* semantics, and discuss previous efforts on improving over those models. This exercice is made in a general context; any remote interaction can be pictured as a *source* and a *target* each performing a method, whether these methods have the same signatures and are application-defined (cf. RMI) or predefined. The importance is the way the parameters to these methods are handled.

2.1 Pass-by-Value Semantics

With pass-by-value semantics, as illustrated by Figure 1(a), an object `o` of type `T` passed as parameter to some interaction between two remote sites is commonly copied entirely (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 $\texttt{o}_c$ of type `T` with an own identity (dark grey circle), whose initial state is the same as the state of the original object at the moment the interaction was triggered. An invocation on such a transferred object `o` (in the following illustrations, `T` is supposed to declare a single parameter-less method `m()` with return type `S`) is hence performed on the target site.

The main dangers with, and hence arguments against, pass-by-value semantics are the following:

VS-I (no synchronization): Since any action performed on a clone on a target site has no effect on the original object, passing parameters by value leads to a pure communication, and does not *inherently*[1] involve synchronization between hosts.

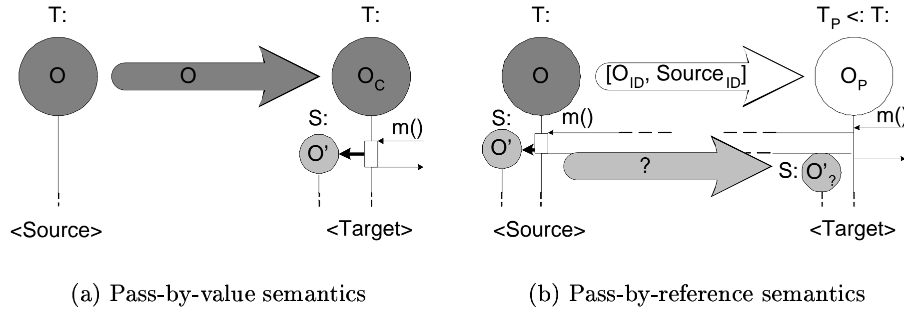
VS-II (excessive transfer): Network resources can be wasted with pass-by-value semantics if a "large" object is transferred and in the end only a "small part" of it is used, or without the object being used at all. This becomes particularly flagrant when not only the state of the object has to be transferred, but also the corresponding code (e.g., byte code, assembly).

VS-I is commonly countered through the argument that, for any form of remote interaction to take place, and be it pure synchronization, there must be a means of passing at least some data structures by value (even if transparently).

2.2 Pass-by-Reference Semantics

With pass-by-reference semantics, as depicted by Figure 1(b), an object `o` passed as parameter from one address space to another is most commonly incarnated on the target site as *proxy* (or *stub*) $\texttt{o}_p$ (white arrow/circle). Such a proxy for an

[1] Synchronization can be achieved separately. E.g., the tuple space allows the passing of an object by value to *only one* among an unlimited number of potential receivers.



(a) Pass-by-value semantics (b) Pass-by-reference semantics

Fig. 1. Classic parameter passing models

object of type `T` is usually of a subtype $\texttt{T}_p$ of `T` ($\texttt{T}_p$ <: `T`), and represents a reference to the original object `o`, forwarding any method invocation `m()` performed on it to that object. The return value `o′` of such an invocation can be itself passed either by value or by reference (light grey arrow/circle) to the client/target site. Proxies, and therethrough the RMI, are intrinsically tied to pass-by-reference semantics. There is no point in passing around first class references to objects, if one can not "use" (i.e., invoke) these.

RMIs, and thus pass-by-reference semantics, have been the subject of long debates around both reliability and efficiency (e.g., [20, 17, 12], to mention only few). The main arguments against reference semantics are the following:

RS-I (flow coupling): Through bi-directional interaction between clients and remotely invoked objects, bad performance of a component, or also network links, affects other components.

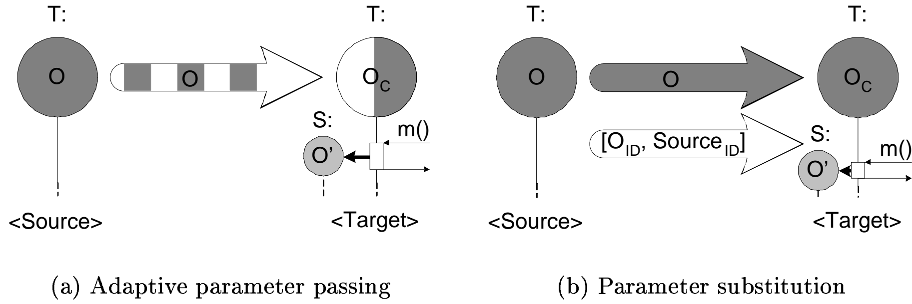
RS-II (failure coupling): By creating a spiderweb of links between proxies and original objects, the RMI tends to create strong dependencies, such that even a single host failure might make an entire application fail.

RS-III (failure hiding): The RMI abstracts over physical distribution, and in particular, over communication and host failures, making it impossible to deal with these realistic phenomenons.

RS-IV (excessive invocation): The wide use of RMI as a communication mechanism wastes network resources: rather than transferring an object once to a client site, and performing many operations on it from there, all operations have to pass through the network.

2.3 Advanced Parameter Passing Models

Based on the seemingly crushing evidence (RS-I..IV) against the RMI, early related work consisted in reducing or entirely eliminating RMIs, and diminishing (VS-II), i.e., the amount of data unnecessarily transferred by value.



(a) Adaptive parameter passing (b) Parameter substitution

Fig. 2. Advanced parameter passing models

Adaptive parameter passing. In order to avoid transferring large objects and using only "small parts" of them on the target site, [17] suggests programmer support (through a graphical editor) to define the "necessary" parts of objects with respect to pass-by-value semantics in remote interaction. This approach, depicted in Figure 2(a), has the shortcoming that it 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`.

Parameter substitution. The system described in [20] refrains from transferring an object `o` by value from a source space to a target space in the case latter space already hosts a copy o_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, e.g., `o` has been modified between the creation of o_c, and the second time it is transferred (and hence before the invocation of `m()`) in Figure 2(b). Defining whether objects are immutable, or what inconsistencies can be tolerated becomes the main task.

3 Lazy Parameter Passing Semantics

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

3.1 Motivation and Overview

Despite numerous efforts to ban the RMI, it remains, legitimately, to date one of the most popular paradigms for distributed programming.

On the one hand, pass-by-value semantics are necessary, and clear apply well to objects which are small (e.g., numbers, strings, or aggregates thereof), and/or "passive" (i.e., whose methods mainly serve as access means to their

fields, e.g., *events* [13]). On the other hand, objects which represent larger entities (e.g., "servers", or entire "services" [27]), possibly with "active" behavior, are simply better handled by references, especially if they are location-dependent (e.g., databases).

Hence there is a clear need for both parameter passing models. Most critiques of the RMI were mostly due to the way it was put to work in early systems, and have been addressed since:

Contra RS-I: Several asynchronous variants of the RMI have been adopted in practice (cf. CORBA Messaging [21]). The concept of group proxy (e.g., [3]) opens the door to other than strict synchronous one-to-one interaction models.

Contra RS-II: Group proxies also enable the replication of critical objects, and the combination of RMI with transactions further embraces fault tolerance.

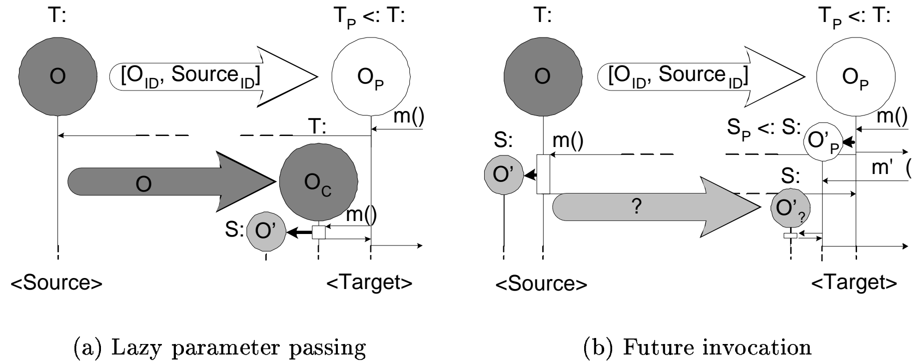
Contra RS-III: Current implementations of RMI make distribution explicit. Proxies do not (*should* not, and actually *can* not [15]) pretend to be the original objects, and distribution-related failures are reflected through specific exceptions. In Java for instance, methods declared by types of remotely invocable objects must all be able to throw `RemoteExceptions` raised by the underlying communication protocols.

However, the problem of judicious network resource utilization remains (RS-IV and VS-II resp.). What model to use when passing a given object? This question can not simply be answered by defining a threshold on the size of object representations, which would solomonically divide the object universe into such passed by value and such passed by reference. Transferring an entire service for the mere purpose of invoking it once wastes network capacities the same way as performing a large number of invocations on that very same service does, especially if the input parameters and the return values also represent massive amounts of data.

The approach taken in this paper consists in providing support for the representatives of the large family of objects which are somewhere between the extremes of string objects and database servers. This support consists in passing these objects by reference initially for a "first contact" (see Figure 3(a)), and by value later on, at a precise instant, and only if really required in the respective context.

This parameter passing model is useful in any distributed system where interaction, taking place asynchronously, is proposed to components without questioning their interest in it. But even in (synchronous) RMIs, where due to the inability/inadequacy of updating static interfaces of services regularly not all parameters to these RMIs are always used, this model is relevant.

A yet more pertinent example is given by the P2P model. When attempting to borrow resources, i.e., querying other peers for resources, a peer is most likely to be provided *several* resources. This is a consequence of the absence of centralized and hence strongly consistent knowledge in the large scale, dynamic, and self-organized settings that P2P systems are [22, 23].



(a) Lazy parameter passing

(b) Future invocation

Fig. 3. Implicit lazy parameter passing vs implicit lazy invocations

3.2 Variants

In the following we describe two variants of lazy parameter passing, differing in the definition of *when* to trigger the passing of an object value.

Implicit lazy parameter passing. With *implicit* lazy parameter passing, an object `o` is first incarnated on a target site by a proxy $\texttt{o}_p$. Upon the first use of `o` through $\texttt{o}_p$, the "download" of `o` is triggered. This is shown by the invocation of `m()` in Figure 3(a), which blocks until it can be locally processed.

In P2P queries, this variant is particularly useful when the first, say n, resources received in reply to a query are to be used, irrespective of their individual characteristics.

Explicit lazy parameter passing. With *explicit* lazy pass-by-value semantics, the program(mer) defines at what moment a corresponding object should be transferred by value to the considered target address space. For the time being, the object is handled like an object passed by reference (cf. Figure 1(b)), meaning that it can be invoked in a RMI style. This can for instance help to acquire knowledge about more detailed characteristics of that object, aiding in making the decision of whether to trigger a transfer by value at all.

The advantage is clear in P2P queries. When receiving multiple resources in reply to such a query, these can be futher investigated before being selectively "downloaded".

3.3 In Perspective

We discuss here lazy parameter passing in the face of related work introduced in the previous section, and more importantly, of future invocations. This discussion is indicated at this point, as latter concept will be combined with lazy parameter passing in the following.

Vs alternative parameter passing models. Interestingly, the two improvements for pass-by-value semantics analyzed in Section 2.3 can be viewed as orthogonal with respect to each other, and also as orthogonal to our proposal.

Let us elucidate this further. Lazy parameter passing defines *if and when exactly a given object should be passed by value.* Provided that an object should be passed by value, [20] defines *whether it is effectively necessary to pass the value of the given object.* In case some "valid" copy of the object is already present on the target site, that one can be used. Otherwise, [17] designates *what exactly then the value to be passed is*, i.e., what parts of the object representation are to be passed.

Vs future invocations. Lazy parameter passing is close in spirit to the paradigm of *future (method) invocation* [31] introduced to repress RS-I (and RS-II): as illustrated by Figure 3(b), a two-way (remote) invocation of a method `m()` returns immediately a *future object* o'_p, which will be linked to the effective "value" o' as soon as that value has been computed and received ($o'_?$) on the target/client site (no matter the passing semantics). In the meantine, the client is only blocked upon further invocation of a method `m'()` on o'_p.

Yet, lazy parameters are not simply future objects. The transfer of a (return) value represented by a future object starts immediately, once that value is available. The goal is thus to decouple the control flows of source and target sites by masking the computation/transfer of such a value. With a lazy parameter, it is not even clear whether the corresponding object will have to be transferred eventually, i.e., the goal is to reduce unnecessary transfers (Figure 3(a)). Passing from one abstraction to the other is hence not straightforward. To pass an object with lazy pass-by-value semantics by making use of future invocations, one would have to query the object's entire state through future invocations, and reconstruct a new object on the target site — a largely inefficient task presupposing that the entire state of the object can be queried remotely.

Lazy parameter passing however, just like future invocations, come in an explicit and an implicit (a.k.a. *wait-by-necessity* [4]) flavor. Akin to implicit futures, implicit lazy parameter passing 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 o'_p (Figure 3(b)) should be used *as late as possible* to make sure the corresponding value o' has been computed and transferred in the meantime, an object o_p (Figure 3(a)) passed by value lazily should be used *only if necessary.* With both paradigms, explicit flavors provide more control to the advanced programmer. However, whereas explicit future invocations require the return types of methods to change, explicit lazy parameter passing applies to objects as a whole, and not to (the invocation of) their methods, and thus respects original method signatures.

Combining lazy parameter passing and future invocations. Being two different paradigms, wait-by-necessity (in the following, we will only more focus

on implicit future invocations) and lazy parameter passing can be provided side-by-side. Care is only required when invoking in a wait-by-necessity style an object passed with implicit lazy pass-by-value semantics to the invokee site: to respect wait-by-necessity, the object should return a future object *immediately* in reply, and not block until the object itself is received. This subtlety will in fact be exploited in the following to provide the choice, when triggering the download of an object with lazy pass-by-value semantics, between non-blocking (wait-by-necessity enabled) and blocking (disabled) semantics for the method invocation triggering the download.

Note that one could also think of a third variant of lazy parameter passing inspired by wait-by-necessity, where the transfer of a parameter by value would start immediately, and a proxy would, similarly to a future object, be passed to the target site for the time being. This could be useful to perform an interaction involving large state-full parameters, before those objects have been yet passed. This variant will however not be further pursued in this paper, as we believe that with parameters of such important size (in P2P computing) it is advisable to first query the target site about any intended use.

4 Expressing Lazy Parameter Passing

This section first outlines mechanisms for supporting the expression of different parameter passing models, before presenting two ways of expressing lazy pass-by-value semantics in Java; (1) a simple scheme, and (2) a scheme more specifically targeted at the use in our BL abstraction. Alternatives are discussed on the fly.

4.1 Mechanisms for Expressing Parameter Passing Models

Intra-process parameter passing occurs in most object-oriented programming languages, similarly to distributed contexts, by value or by reference. Alternative models, such as the *pass-by-name* semantics of Algol [16], are rare. Support for expressing different parameter passing modes in (object-oriented) programming languages consists mainly in providing *tags* or specific *types.*

Tags. C++ [24] is an example of a language leveraging on the distinction between parameter passing modes. Instances of the same type can be passed with different semantics in different contexts. The choice is made via tags, which make part of the syntax of formal argument/variable declarations, e.g., `&`, `*`, `const`.

The possibility of passing the same objects with different semantics in different contexts offers much flexibility, and is easy to achieve inside a single address space, as the objects are readily available.[2]

A sample tag-based solution for distributed contexts is given by [19], where the keyword `separate` denotes variables or formal arguments potentially referencing objects under control of a distinct processing unit, and possibly residing

[2] For the same reason, lazy pass-by-value semantics are less relevant for local contexts.

in a different space. Object types can be also a priori declared as `separate` for convenience. A second tag, `expanded`, is introduced for pass-by-value semantics. As mentioned, [19] describes a scheme for explicitly transferring `separate` objects by value, which is however confusing since the two kinds of tags are said to be mutually exclusive.

Types. The more common approach in distributed contexts consists in associating remote semantics with the types of objects, i.e., using explicit supertypes as tags. This is motivated by the fact that the remote use of objects can impose constraints on them/their types. In (Java) RMI, restrictions can occur on method signatures, and the notion of object identity is delicate, as a proxy represents (1) another object, possibly even (2) on a remote host.

Also, programmer support for serialization and deserialization of objects passed remotely by value is sometimes necessary, or at least useful. Certain objects should simply not be passed by value, and the possibility of customizing serialization yields a base for the scheme proposed in [17].

Java consequently offers two abstract supertypes for objects which can be passed by value and by reference respectively, namely `Serializable` (package `java.io`) and `Remote` (`java.rmi`). Java RMI demonstrates the advantage of tags over types in terms of flexibility, by invariably passing objects which are serializable *and* remote by reference.

Mainly for achieving interoperability, CORBA [21] introduces keywords `in` and `out` to denote parameters and return values, respectively, passed by value. These tags can also be combined to denote pass-by-reference semantics. For passing objects other than instances of primitive types by value, the corresponding types nonetheless still explicitly must inherit from a root "value" type.

4.2 A Simple Scheme in Java

For illustration purposes, we present here a simple way of putting explicit and implicit lazy parameter passing to work in Java, in a way following the spirit of the existing support for distributed parameter passing in Java.

Lazy types. Objects passed by value lazily (instances of the `Lazy` type in Figure 4)) are both serializable and remote. The handling of such instances is performed through the `LazyHandling` class. With its `isLocal()` method, one can verify whether an object has been transferred already. Implicit lazy semantics appear as specialization of explicit lazy semantics only for obtaining a root "lazy" type for the formal arguments of methods in class `LazyHandling`. The solution presented here makes it possible to explicitly `download()` also an instance of `ImplicitLazy`, if one knows that it will be used anyway. This could be easily ruled out by inverting the subtyping order, and using the resulting `ExplicitLazy` type in the signature of `download()`.

```
public interface Lazy extends Serializable, Remote {}

public interface Implicit extends Lazy {}

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

Fig. 4. A simple approach to lazy parameter passing

Failures. The transfer of an object, like any remote interaction, can suffer from a failure in the underlying distributed infrastructure. A failure in an explicit `download()` of an object is reflected by throwing a specific kind of `RemoteException` (`java.rmi`). When implicitly transferring an object by value upon its invocation, the same exception is 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.

Proxies. The `rmic` precompiler can be easily extended to support lazy parameter passing, by applying the same type checking rules to subtypes of `Lazy` as to any subtype of `Remote`, and creating corresponding proxies. Interestingly, adding a new type `WaitByNecessity` as subtype of `Remote` would in contrast require the rules of Java RMI (and hence `rmic`) to be strengthened, at least in the context of subtypes of that new type. Indeed, any method of such a type could only return a value of interface type, as this is the only way of ensuring that a proxy can be created for such a future object (like for all `Remote` types in Java). We will come back to this in Section 5.4.

With the scheme presented in Figure 4, an object (`o` in Figure 3(a)) passed by value lazily is first represented, i.e., incarnated, in the target address space as a proxy ($\texttt{o}_p$). Unless modifying the Java runtime (i.e., the virtual machine), this object can not be swapped, transparently to all local references to it, against the effective object it represents ($\texttt{o}_c$) after downloading latter object. In other terms, after downloading an object, that object is still accessed, indirectly, via a proxy (see Section 5.3). Method `deref()` has been added to class `LazyHandling` to offer the possibility of obtaining the referenced object when required.

Note that since the use of lazy pass-by-value semantics has no direct impact on the implementation of corresponding objects, and hence on their types, a backward-compatible solution could consist in accepting any object which is both serializable and remote as lazy pass-by-value object. This would however not obsolete the (re-) creation of proxies (supporting lazy pass-by-value semantics) for these types.

4.3 Resources in P2P Computing

The second approach presented here is the one retained in the context of our BL abstraction. The flexibility required to make such an abstraction general (i.e., to support a wide variety of P2P applications), has been achieved by offering the same possibilities as with tags: applications can switch, back and forth, between (1) synchronous and asnchronous (wait-by-necessity) invocations for *any* remote object, and *simultaneously* between (2) implicit and explicit semantics for any object subject to lazy parameter passing. And this independently for any client (target) of the object.

Contract methods. Two alternatives to the previously outlined simple scheme are worth mentioning in order to introduce the present approach. These consist namely in declaring the method `download()` (1) such as to return an object (dereferencing implicitly), and/or (2) as part of the `Lazy` type. The former proposition only would have helped in the case of *explicit* transfer of an object by value, and in the case of a pass-by-value triggered implicitly (by invoking a method other than `download()`) the programmer would have still be left with a proxy. The latter proposal would have contributed to making lazy pass-by-value semantics both more explicit and simpler to use on a target site, but at the expense of polluting the types of objects passed with lazy pass-by-value: whom would be left the responsibility of implementing the `download()` method?

We can conclude from the first alternative that one can not avoid incarnating implicit lazy parameters by proxies on target sites in all cases. But if this is so, why don't we let proxies do the things they do best, such as implementing the `download()` method mentioned above in a *decorator pattern* [9] style. This is to some extent already exploited in Java RMI, through the `RemoteExceptions` declared in the signatures of methods of remote objects, but effectively thrown by the underlying communication protocols.

This concept has been systematically expanded in the context of our BL abstraction. Individual characteristics of the different resource types (see Figure 5) introduced to support various applications types, are captured by so-called *contract methods*. These are methods predefined for individual resource types, which reflect the contracts incurred by the use of such resources. These methods vary strongly in semantics. While some can be implemented such as to do nothing since corresponding proxies will provide the implementation, others can be specifically implemented by a resource class to override default behavior.

Resource types. Types `ValueResource` and `ReferenceResource` are self-explanatory. Any object of latter type can benefit from wait-by-necessity. It is sufficient for a client of such a resource to change the invocation style from the default synchronous mode to the asynchronous, wait-by-necessity mode. When doing so, a handler for exceptions thrown by possible communication or peer failures must be provided. For a same resource, this mode can be repeatedly switched.

```
public interface Resource {
  public Object deref() throws NoSupportException;
  public boolean isLocal() throws NoSupportException;
  public void setProtocol(Protocol p) throws NoSupportException;
  public void setQoS(QoS qos) throws NoSupportException;
}
public interface ValueResource extends Resource, Serializable {}
public interface ReferenceResource extends Resource, Remote
{
  public void setInvocation(boolean asynchronous, RemExcHandler h)
    throws NullPointerException;
}
public interface LazyResource extends ValueResource, RemoteResource
{
  public void setDownload(boolean automatic);
  public void download() throws RemoteException, NoSupportException;
}
public interface RemExcHandler { public void handle(RemoteException re); }
```

Fig. 5. Basic resource types in the BL abstraction

Resource types whose instances can be used with either flavor of lazy pass-by-value semantics subtype type `LazyResource`. Through the `setDownload()` contract method, a potential client peer can choose between the two variants (default is explicit). The `download()` method finally triggers the transfer of such a resource. Depending on the chosen invocation style, this call (like an implicitly triggered download of such a resource) either blocks until the resource has been transferred, or returns immediately.

The BL abstraction encompasses many further abstract resource subtypes (not presented in detail for the sake of brevity), such as *dynamic resources* (devoid of statically defined interfaces), *replicated resources* (fault tolerance through automatic replication in the face of updates), or *replaceable resources* (transparent updating of resources). Abstract resource types can of course also be combined (according to given rules), and can also be used recursively, e.g., for types of return values or arguments of (remote) resource methods.

Dynamic proxies. To avoid the implementation and use of a precompiler à la `rmic` for the creation of proxy classes, we have made use of the concept of *dynamic proxies* [26] added to Java at version 1.3.

In short, a dynamic proxy is an object which conforms to a non-empty set of interfaces, for which that proxy ('s class) was created through class `Proxy`.[3]

[3] For presentation simplicity, we omit the package name `java.lang.reflect` common to all types for reflection (except class meta-objects, i.e., `java.lang.Class`).

A proxy can be used as an instance of any of the interfaces it was created for, i.e., it can be cast and invoked accordingly. An invocation performed on such a dynamic proxy object is however *reified*, somehow stepping from a statically typed context to dynamic interaction where *any* action can be performed in the confines of a method invocation.

Hence, it is an ideal means for implementing the targeted decorator pattern. Consequently, all resource objects passed by parameter are incarnated by dynamic proxies on target sites.

5 Putting Lazy Parameter Passing to Work

In this section we depict how we have put explicit and implicit lazy pass-by-value semantics to work in our borrow/lend (BL) abstraction. We illustrate this through our Java prototype of BL, pointing out how dynamic proxies have been used to implement the various parameter passing semantics, and also other distinctive features of the BL abstraction aiming at decoupling peers. Thereby, we also pinpoint limitations of dynamic proxies.

5.1 Borrowers and Lenders

The BL abstraction (Figure 6) is based on the model of P2P programming sketched previously (cf. Section 1), leading to two classes `Borrower` and `Lender` which are instantiated by peers whenever they intend to *borrow* or *lend* resources, respectively. Our prototype relies on Sun's compiler prototype for genericity [25] to improve type safety by introducing a type parameter to borrowers and lenders.

This type parameter represents the first of three criteria through which a resource borrower can describe the resources it is interested in:

Type: Borrower intererests are expressed for objects of a given type `B`, with which the type `L` of a resource borrowed from a lender must "conform". Different "depths" of conformance between `L` and `B` are possible, ranging from nominal (explicit) conformance over different levels of structural (implicit) conformance to a form of completely dynamic interaction (dynamic resources). More precisely, with a depth of 0, `L` has to be a declared subtype of `B` or `B` itself, while with a depth of 1, `L` only has to provide all the members declared by `B`. With depth 2, types of fields and method parameters in `L` only have to be conformant at depth 1 with those of `B`, etc.

Key: Lenders can explicitly attach a key in the form of an array of bytes to a resource when lending that resource. This key plays the role of access control mechanism, and a corresponding argument is hence present on the borrower side as well.

Predicates: "Content-based" criteria can be expressed through predicates, based on the members of the type specified by borrowers (and lenders). This is best illustrated by the following example.

```
public interface Participant<R extends Resource> {
  public void activate() throws ActiveException, RemoteException;
  public void deactivate() throws InactiveException, RemoteException;
  public R setConstraints() throws InvalidConstraintException;
}
public class Lender<R> implements Participant<R> {
  public Lender(R lent, byte[] key) {...}
  ...
}
public class Borrower<R> implements Participant<R> {
  public Borrower(Inbox<R> in, byte[] key) {...}
  public R setConformance(int depth) throws {...}
  ...
}
public interface Inbox<R> { public void deliver(R r); }
```

Fig. 6. `Borrower` and `Lender` types (excerpt)

5.2 Exchanging Music

Consider the (in)famous scenario of songs being shared throughout the Internet. A typical Java type for incarnating such songs is proposed in Figure 7. An instance of `Song` hence conveys information about (1) the title, (2) the artist, and (3) the effective track (type `AudioInputStream` in package `javax.sound.sampled`) for a song.

A music track can now be shared with other users as follows (exception handling omitted for simplicity):

```
Song lSong = new Song("The next love song", "The next boys band", ...);
Lender<Song> sLender = new Lender<Song>(lSong, ...);
sLender.activate();
```

Symmetrically, interest in songs can be expressed by peers as follows:

```
Borrower<Song> songs =
  new Borrower<Song>(new Inbox<Song>() {
    public void deliver(Song bSong) {
      if (!Jukebox.isQueued(bSong));
      bSong.download(new ftp(...));
      Jukebox.queue(bSong);
    }
  }, ...);
Song pSong = songs.setConstraints();
pSong.setInvocation(true, new RemoteExceptionHandler{...});
pSong.getArtist().equals("The next boys band");
songs.activate();
```

```
public class Song implements LazyResource {
  public String getTitle() throws RemoteException {...}
  public String getArtist() throws RemoteException {...}
  public AudioInputStream getStream() throws RemoteException {...}
  public Song(String title, String artist, AudioInputStream s) {...}
  ...
}
```

Fig. 7. Representing songs

Here, duplicates are filtered by storing songs in a jukebox and checking whether a (new) received song is already present. To that end, songs are compared through `getTitle()` and `getArtist()` invoked remotely on the new song represented through `bSong`, by the jukebox (details omitted). Only if not already present, the song is then downloaded by invoking the `download()` method with wait-by-necessity remote invocation. Thanks to the lazy synchronization of that download procedure, the song can be enqueued by the jukebox before having effectively been entirely received at that point — releasing the corresponding thread. This lazy synchronization is achieved by making that call on a dynamic proxy, which is marked in the example by *emphasizing* that call. Similarly, all invocations made in the context of predicates, including such based on contract methods, are made on dynamic proxies, and hence *emphasized.* In the example, interests are expressed in objects which are instances of `Song` *and* of a certain band (an *or* of several conditions requires these to be expressed on individual proxies obtained by successive calls to `setConstraints()`).

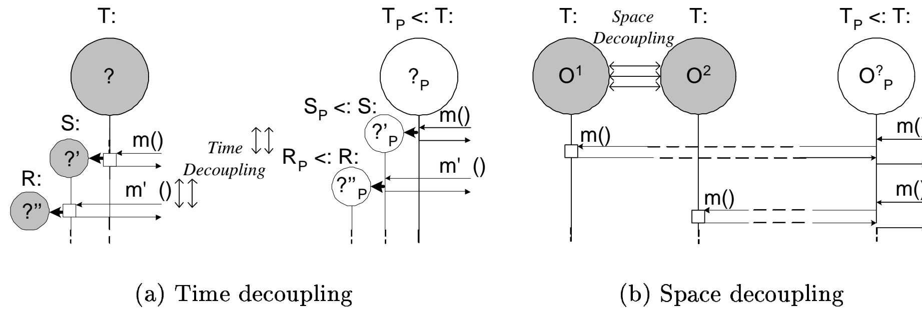
5.3 Dynamic Proxies in the BL Abstraction

We illustrate in the following the different places in the BL abstraction in which dynamic proxies are used to achieve strong decoupling of remote peers in response to the asynchronous and dynamic nature of P2P systems:

Time decoupling: Upon *constraint expression*, i.e., when "registering" invocations of contract methods as well as resource-specific methods with proxies (e.g., through `m().m'()` on $?_p$ in Figure 8(a)), time decoupling is achieved between components. In the example above, the borrower can receive new songs of "The next boys band" (e.g., `?`) lent by peers *after* the borrower expressed its interests through `pSong`.

Flow decoupling: By delivering resources as proxies, *lazy synchronization* can be provided when invoking remote resources (wait-by-necessity, e.g., `o` invoked through o_p in Figure 3(b)), even when triggering an automatic lazy pass-by-value transfer of resources with such invocations. In the above example, a song can be queued for playing before it has been entirely downloaded.

Space decoupling: Space decoupling is similarly nicely demonstrated through lazy synchronization. When an object is finally passed by value (`o`, resp. o_c in



(a) Time decoupling

(b) Space decoupling

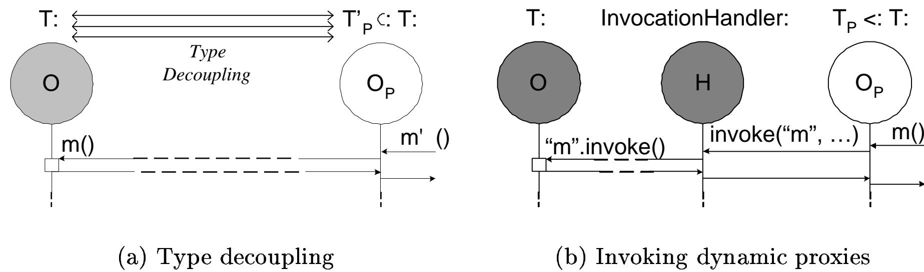
Fig. 8. Decoupling and dynamic proxies (I)

Figure 3(a)), a reference to that object (e.g., $\mathtt{o}^?{}_p$ in Figure 8(b)) is switched from a remote "reference" to a local copy of the object ($\mathtt{o}^1$ and $\mathtt{o}^2$ resp. in Figure 8(b)). Without hooks into the virtual machine, this is the only means of transparently to the program swapping at any moment the object pointed to by a variable of arbitrary type.
Also, the chosen decorator pattern does not pollute resource types, unlike the regrouping of contract methods in abstract resource classes to be subclassed by application-defined resource ones. This is a substantial benefit in a language with single inheritance such as Java.

Type decoupling: The adapter pattern supported by dynamic proxies can be used to implement *structural conformance* between types of lent and borrowed resources (Figure 9(a)). When expressing interests in a type `Track` not explicitly related to `Song` but encompassing a subset of the members of `Song` (`Song` $\subset$: `Track`), a borrower could nevertheless receive the published songs by adapting its constraints (e.g., `songs.setConformance(1)`).

5.4 Limitations

The elucidations above hide one important caveat: dynamic proxies are only available for interfaces, i.e., it is not possible to create a dynamic proxy and assign it to a variable of class type. Hence, wherever a parameter is to benefit from wait-by-necessity or lazy passing, the corresponding formal argument must be declared as an interface type. This constraint applies recursively. E.g., return values of methods invoked with wait-by-necessity, or lazy pass-by-value parameters to invocations on remote resources, have to be interface types as well. Also queries are strongly limited by this. None of the constraints expressed in the example can be put to work as presented, since `Song` is a class. Circumventing this shortcoming boils down to using exclusively interface types in variable and formal argument declarations, and making use of classes only in instantiations. This constraint is the more annoying, as unlike with the use of a precompiler



(a) Type decoupling

(b) Invoking dynamic proxies

Fig. 9. Decoupling and dynamic proxies (II)

à-la `rmic` for compile-time proxy generation, it can not be verified statically (see Section 4.2).

This shortcoming of dynamic proxies is not tied to our uses. It awakes serious doubts as to their usefulness for realizing *behavioral reflection* [14], which they are often praised for. Indeed, while the absence of transparency in the association of a behavior, represented by a *meta-level* [14] object, with a *base-level* object, through a proxy, turns out to be an advantage in our case (see “Contra RS-III”, Section 3.1), other known lacks of a proxy approach are namely amplified:

Self-invocations: Invocations made by methods of base-level objects on other methods of themselves can not intercepted (cf. “self problem” [15]).

Self-references: Similarly, *self-references* returned by a base-level object invoked through a proxy would have to be recognized, such that a proxy could be returned instead (cf. “encapsulation problem” [15]).

Latter issue includes also references provided by an object to its fields. To not break with reflection, these fields have to be shielded again behind dynamic proxies, meaning that the return type of any method providing such access has to be an interface.

6 Implementing Lazy Parameter Passing

In this section, we first present in more detail the concept of dynamic proxies introduced with Java 1.3., before describing our extension to that mechanism motivated by the limitations pointed out above.

6.1 Internals of Dynamic Proxies

As conveyed by Section 4.3, dynamic proxies are a type-safe means of obtaining, at run-time, an object which conforms to any set of interfaces.

```
public class Proxy implements Serializable {
  public static Class getProxyClass(ClassLoader l, Class[] is)
    throws IllegalArgumentException {...}
  public static Object newProxyInstance(ClassLoader l, Class[] is,
                                        InvocationHandler h)
    throws IllegalArgumentException {...}
  public static InvocationHandler getInvocationHandler(Object p)
    throws IllegalArgumentException {...}
  ...
}
public interface InvocationHandler {
  public Object invoke(Object p, Method m, Object[] as) throws Throwable;
}
```

Fig. 10. `Proxy` and `InvocationHandler` types (excerpt)

Creating dynamic proxies. Dynamic proxies are implemented as a library. Invoking the `getProxyClass()` method in class `Proxy` leads to creating a proxy[4] class, directly as byte code, as a subclass of `Proxy` which implements the interfaces specified by `Class` meta-objects (unless a proxy class has already been created for that precise set of interfaces). The `newProxyInstance()` method in addition instantiates the possibly generated proxy class.

The `Proxy` class hence has a dual purpose. First, it contains class methods, described above, which permit the generation of proxies/proxy classes, thus serving as "factory". Second, the `Proxy` class serves as supertype for all dynamic proxy classes, making it possible to easily verify (`instanceof`) whether a given object is a proxy.

Note that in certain cases it is impossible to create a proxy class for a set of interfaces ([26]). Failures might especially occur whenever conflicts would also arise if a class was explicitly, i.e., statically, defined implementing the specified set of interfaces.

Invocation handlers. An invocation performed on a dynamic proxy is reified and passed to an `InvocationHandler` (Figure 10) object, associated to that proxy at instantiation, through the handler's `invoke()` method (Figure 9(b)). The arguments for `invoke()` include (1) the proxy object on which the method was originally invoked, (2) a meta-object representing the method (`Method`) that was originally invoked, and (3) the effective arguments (an array of `Objects`) to that invocation. The `invoke()` method is hence capable of handling *any* method invocation (parameters of primitive types are transformed to the corresponding wrapper types). It can be pictured as the symmetric counterpart to the `invoke()` method of class `Method`: while the latter method defers to run-time the choice of *which method to invoke*, the former method defers to run-time *what to do upon*

[4] In the following the qualifier "dynamic" might be omitted when given by the context.

invocation of a method (Figure 9(b)). Behavioral reflection is easily achieved by combining the two, as illustrated by Figure 9(b).

According to the specification [26], particular exceptions can be thrown upon invocation of a proxy (signalled by `Throwable`, cf. `invoke()`) — typically when the returned values or thrown exceptions do not match the specification of the invoked method in the interface the proxy was created for.

6.2 Dynamic Proxies for Classes

Our approach to supporting "dynamic proxies for classes" builds on the principle applied for the generation of proxies for interfaces, that is, a proxy class for a set of types including a class is generated when needed at run-time as byte code, loaded, and linked. By retaining the original principle, the corresponding libraries have remained fully backward compatible.

Extension approach. A proxy class created for a set of types including a class `C` must not only implement the specified interfaces, but must in addition subclass `C`. This approach can be characterized as an *extension* approach (extending classes to be reflected upon) in contrast to prominent implementations of behavioral reflection which rather follow an *envelopment* approach (wrapping code instructions with jumps to the meta-level, e.g., [30, 28]). In order for a proxy to be able to reify any action performed on it, its class must hence "override" all superclass members. Quite obviously, this poses problems in the case of `final` (cf. [28]) and `private` methods, as well as fields. In the following, we propose a set of transformations performed at byte code level by an instrumenting class loader for dealing with those cases.

Proxy types and access handlers. Since proxy classes created for a set of types including a class cannot subclass class `Proxy`, we introduce the `ProxyType` interface as common supertype for all proxy types. The `Proxy` class still serves as superclass for proxy classes created for interfaces exclusively, and hence implements `ProxyType`. This is depicted in Figure 11, in which modifications/additions in the new backwards-compatible version of the `Proxy` class are reported, and *emphasized.* Type `AccessHandler` is introduced to deal with (instance) field accesses in addition to (instance) method invocations in the case of proxies for classes.

Handling field accesses. Since in Java, like in many other languages, fields can not be overridden by subclasses, but merely *shadowed,* we propose to transform field accesses to invocations of automatically generated access methods, and to override these in proxy classes to enable interception. Due to this same principle of shadowing, simply defining a getter/setter method pair à la `getf()/setf()` for each field `f` does however not suffice.

Our solution consists in conveying information about the declaring classes of fields by their respective getter/setter methods. This is illustrated in the

```
public class Proxy implements ProxyType {
  ...
  public static Class getProxyClass(ClassLoader l, Class[] is, Class cl)
    throws IllegalArgumentException {...}
  public static Object newProxyInstance(ClassLoader l, Class[] is,
                                        Class cl, InvocationHandler ih,
                                        AccessHandler ah) throws ...
  public static AccessHandler getAccessHandler(Object p) throws ...
  ...
}
public interface ProxyType extends Serializable {}
public interface AccessHandler {
  public Object get(Object p, Field f) throws RuntimeException;
  public void set(Object p, Field f, Object val) throws RuntimeException;
}
```

Fig. 11. Additions to the `Proxy` class and auxiliary types (excerpt)

following through three recursive subclasses (in source code rather than byte code for readability):

```
class C1 {                  ⟹    class C1 {
  String f;                        String f;
}                                  String get$C1$f() { return f; }
                                   void set$C1$f(String f) { this.f = f; }
                                 }
class C2 extends C1 {}      ⟹    class C2 extends C1 {}
class C3 extends C2 {       ⟹    class C3 extends C2 {
  String f;                        String f;
}                                  String get$C3$f() { return f; }
                                   void set$C3$f(String f) { this.f = f; }
                                 }
```

Observe the corresponding transformations when accessing these fields (variable `cx` is of type `Cx`):

```
c1.f = ...;            ⟹    c1.set$C1$f(...);
... = c1.f;                 ... = c1.get$C1$f();
c2.f = ...;            ⟹    c2.set$C1$f(...);
... = c2.f;                 ... = c2.get$C1$f();
c3.f = ...;            ⟹    c3.set$C3$f(...);
... = c3.f;                 ... = c3.get$C3$f();
```

Without loss of validity, the package name of a class is viewed as part of its name.

Private fields and methods. The dispatch of a `private` (getter/setter or not) method resembles field lookup, in that such a dispatch starts at the *declaring* class and proceeds upwards in the superclass hierarchy, hence mocking any attempt of overriding such a method in a subclass.

This resemblance has suggested the adoption of a similar scheme for interception of application-defined `private` methods as for field accesses, consisting in complementing `private` methods with *stub methods*, through which former methods are indirectly invoked. A stub method differs from its corresponding original method in its visibility modifier (package visibility), and name (the name of the original method is prefixed by `C$`, `C` being the name of the declaring class). These prefixes, just like the infixes in getter/setter methods, are used to avoid accidental overriding in subclasses.

Note that modifying (renaming) `private` methods directly would invalidate lookup tables of corresponding `native` methods.

Final classes and methods. The workaround to the limitations introduced by the `final` keyword consists in handling such classes and methods as non-`final` ones when *linking* these, but remembering occurrences of these keywords for the *verification* of classes, in order to indicate violations.

As a result, unlike in the original implementation of dynamic proxies, `final` methods of the root object type `Object` can now be overridden, and intercepted.

Proxy class instantiation, safety and security. [6] provides more details on various issues, such as the implementation of our instrumenting class loader, the creation of constructors for the instantiation of proxy classes, or safety: e.g., how to deal with invocations and field accesses made through introspection, or how to add keys to names of generated methods for obfuscating these.

With respect to the issue of security often raised in the context of reflection [5], we can safely say that, considering our class loader to be trusted (a common assumption [30, 28]), our approach does not introduce any further implications compared to the original dynamic proxy mechanism.

.

7 Summary and Conclusions

With the continuously increasing scale of distributed applications — and nowadays nearly any industrial-scale application is distributed — it becomes ever more important to reveil certain characteristics of the underlying distributed infrastructure to the application developer, rather than hiding them. “Harmful” characteristics culminate in P2P settings, which namely involve an unprecedented number of participants (large scale), joining and leaving (dynamic) without coordination (completely decentralized). P2P computing is not a topic made up by research. It is a reality, whose recent thorough investigation has been inversely driven by the desire to understand the fundamentals of a concept already widely applied practice.

We believe that the fundamentals of (distributed) object-oriented programming and software design have to evolve with these new constraints. This paper can be seen as a step in that direction, focusing on parameter passing semantics in modern distributed systems. Just like a lazy RMI (future invocation) better captures the asynchronous nature of the Internet in pairwise client/server interaction, the lazy parameter passing model proposed in this paper in two different flavors better captures multi-party interaction in modern asynchronous decentralized distributed systems.

While implementing lazy parameter passing in our BL abstraction for P2P computing, our efforts have been stifled by the limitations of dynamic proxies in Java, which hamper the potential of this mechanism (e.g., for behavioral reflection) overall. While several authors have suggested ways of augmenting Java's reflection capabilities in the large, we have in this paper presented an approach to broadening the scope of Java's own concept of dynamic proxies.

The proposed solution relies on byte code manipulations performed at load-time — an established technique in Java. These manipulations include a general scheme for transforming field accesses to invocations of automatically generated getter/setter methods, making a case against the claim that field access interception is impossible without modified virtual machine [5].

The issues addressed in this paper are not only relevant for Java. Microsoft's .NET platform [29] for instance proposes a closely related concept of dynamic proxies. When implementing our .NET prototype of the BL abstraction, we had to deal with the same limitations with respect to dynamic proxies as in Java.

References

1. A. Adya, W.J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J.R. Douceur, J. Howell, J.R. Lorch, M. Theimer, and R.P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Usenix Symposium on Operating Systems Design and Implementation Symposium*, 2002.
2. S. Baehni, P.Th. Eugster, R. Guerraoui, and P.Altherr. Pragmatic Type Interoperability. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems*, 2003.
3. A.P. Black and M.P. Immel. Encapsulating Plurality. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 56–79, 1993.
4. D. Caromel. Towards a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36:90–102, 1993.
5. D. Caromel and J. Vayssière. Reflections on MOPs, Components, and Java Security. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 256–274, 2001.
6. P.Th. Eugster. Dynamic Proxies for Classes: Towards Type-Safe Remote Object Interaction. TR 200317, Swiss Federal Institute of Technology in Lausanne, 2003.
7. P.Th. Eugster and S. Baehni. Abstracting Remote Object Interaction in a Peer-to-Peer Environment. In *2002 ACM Java Grande Conference*, pages 46–55, 2002.
8. P.Th. Eugster, R. Guerraoui, S.B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight Probabilistic Broadcast. *ACM Transactions on Com-*

puter Systems, to appear 2003 (preliminary version in the 2001 IEEE International Conference on Dependable Systems and Networks).
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
10. V. Getov, G. v.Laszewski, M. Philippsen, and I. Foster. Multiparadigm Communications in Java for Grid Computing. *Communications of the ACM*, 44(10):118–125, 2001.
11. L. Gong. *Project JXTA: A Technology Overview*, 2001.
12. R. Guerraoui. What Object-Oriented Distributed Programming Does not Have to Be, and What it May Be. *Informatik*, 2, 1999.
13. T. Harrison, D. Levine, and D.C. Schmidt. 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*, pages 184–200, 1997.
14. G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
15. H. Liebermann. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 214–223, 1986.
16. C.H. Lindsey. A History of Algol 68. In *Second ACM SIGPLAN Conference on History of Programming Languages*, pages 97–132, 1993.
17. C.V. Lopes. Adaptive Parameter Passing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*, 1996.
18. S. Matsuoka and S. Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *Proceedings of the 3rd ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 276–284, 1988.
19. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2nd ed., 1998.
20. 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*, pages 411–418, 1996.
21. OMG. *Common Object Request Broker: Architecture and Specification 3.0*, 2002.
22. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 161–172, 2001.
23. I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.
24. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd ed., 1997.
25. Sun. *Adding Generics to the Java Programming Language. Java Specification Request (JSR) 000014*.
26. Sun. *Dynamic Proxy Classes*, 1999.
27. Sun. *Java Remote Method Invocation - Distributed Computing for Java*, 1999.
28. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 236–244, 2001.
29. T. Thai and H. Lam. *.NET Framework Essentials*. O'Reilly and Associates, 2001.
30. I. Welch and R.J. Stroud. Kava - A Reflective Java Based on Bytecode Rewriting. In *Reflection and Software Engineering*, pages 155–167. Springer, 1999.
31. A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. *Object-Oriented Concurrent Programming*, chapter 4: Modeling and Programming in an Object-Oriented Concurrent Language ABCL/1, pages 55–89. MIT Press, 1987.