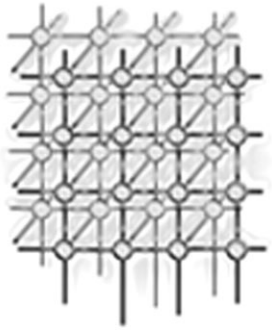
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; **17**:1053–1078
Published online 23 February 2005 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.842

Object-oriented programming in peer-to-peer systems



Patrick Th. Eugster[1,2,*,†] and Sebastien Baehni[1]

[1] *Distributed Programming Laboratory, Swiss Federal Institute of Technology, Lausanne, Switzerland*
[2] *Sun Microsystems, Switzerland*

SUMMARY

Leveraged by the success of applications aiming at the 'free' sharing of data in the Internet, the paradigm of peer-to-peer (P2P) computing has had substantial consideration devoted to it recently. This paper presents a high-level abstraction for remote object interaction in a P2P environment, called borrow/lend (BL). We present the principles underlying our BL abstraction, and illustrate how this abstraction can be used to program P2P applications in Java. We contrast our abstraction with established abstractions for distributed programming such as the remote method invocation or the tuple space, illustrating how the BL abstraction, obviously influenced by such previous abstractions, unifies flavors of these, but also how it captures the constraints specific to P2P environments. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: borrow/lend; peer-to-peer; abstraction; type; Java

1. INTRODUCTION

Through the overnight success stories of many non-profit programs enabling the collaboration of users in the goal of distributing data throughout the Internet (e.g. Gnutella [1], Freenet [2]), the paradigm of *peer-to-peer* (P2P) [3] computing has become extremely popular. Under the name of P2P, most authors in the field agree upon a completely decentralized distributed setting in which basically any of the potentially many hosts has resources to share with others.

When this data 'freely' shared among hosts is commercial multimedia, such as music in MP3 format, the moral promoted by some of the above-mentioned programs is rather dubious, and has become the source of much polemics. Nevertheless, the practical value of the P2P paradigm in a broader sense is unquestionable. This observation has led to many important efforts in the field, and is emphasized

*Correspondence to: Patrick Th. Eugster, Distributed Programming Laboratory (LPD), Swiss Federal Institute of Technology, Lausanne CH-1015, Switzerland.
†E-mail: patrick.eugster@epfl.ch

Contract/grant sponsor: Swiss National Science Foundation NCCR-MICS; contract/grant number: 5005-76322 (Terminodes)
Contract/grant sponsor: European IST FET; contract/grant number: IST-2001-33234 (PEPITO)

Received 15 January 2003
Revised 28 August 2003
Accepted 14 October 2003

by the very fact that the striving for decentralized applications, with the avoidance of bottlenecks and single points of failure in mind, has been a major concern in the distributed and dependable systems community for a long time.

Ever since, P2P computing has been the subject of more profound studies, also benefitting from previous research in the above-mentioned community. For instance, many previous application-level protocols for data routing or membership management, especially those with a focus on scalability, have been adapted to P2P environments. However, the P2P paradigm has some characteristics of its own, leading to new challenges. The addressing of these challenges by the research community has also led to novel protocols (e.g. [4–6]), which capture precisely the characteristics of P2P environments.

Similarly to the underlying protocols, abstractions established in distributed object programming, such as the *remote method invocation* (RMI, typical for client/server interaction [7]), the *tuple space* [8] (emerged from parallel computing based on distributed shared memory) or the *publish/subscribe* [9] (for mass dissemination of events) abstractions, also indeed make sense in P2P settings, and have already been used successfully in those contexts. In particular, the publish/subscribe abstraction captures many characteristics of P2P environments, and has been widely employed to model and implement remote interaction in P2P settings (e.g. [10]).

In this paper we present a high-level abstraction called *borrow/lend* (BL) we have implemented in the Distributed Asynchronous Computing Environment (DACE) [11] together with protocols [12,13] *specifically* for P2P object programming. Our BL abstraction has been initially implemented in Java, while more recently efforts have also been invested in the implementation of a prototype for Microsoft's .NET platform [14]. In this paper, we present the concepts of the BL abstraction in Java, since *genericity*[‡] is currently in a more advanced state in Java [15] than comparable efforts in the .NET platform [16]. As we illustrate in this paper through the BL abstraction, the combination of genericity with *reflection* is very useful for the implementation of abstractions for distributed programming in statically typed object-oriented programming languages. We present measurements obtained with our .NET prototype focusing on the overhead of our use of reflection (including features also present in Java).

Emerging from DACE, a general framework for distributed object programming, it is not surprising that our BL abstraction combines, and unifies, flavors of many other previous abstractions, its main contributor being a variant of the publish/subscribe abstraction implemented in the DACE framework [17]. Nevertheless, we believe that our BL abstraction, which can be pictured as representing a general service for interchanging *resource objects* (or simply *resources*), has some considerable differences to previous abstractions, which we introduce by pointing out the drawbacks of 'classic' abstractions in dynamic P2P settings. In short, BL: (1) makes transmission protocols and parameters governing *qualities of service* (QoS) explicit, yet provides high-level guarantees such as type safety and encapsulation; (2) embraces a form of (asynchronous) RMI, but at the same time incorporates the functionality of a distributed lookup service; (3) supports application-defined concurrency policies; and (4) provides fine-grained control of resource sharing (e.g. cancelling or replacing resources), aiding garbage collection and hence improving scalability. These characteristics make our BL abstraction an ideal paradigm for P2P programming, without however limiting its applicability to such contexts.

[‡]Genericity is foreseen for Java 1.5 [15]. Our implementation relies on the compiler prototype available from Sun.

The remainder of this paper is structured as follows. Section 2 introduces the distinctive features of our BL abstraction by discussing 'classic' abstractions and specifications for distributed programming, focusing on P2P computing, and Java. Section 3 illustrates the notions of resource borrowing and lending. Section 4 characterizes the nature of resources. Section 5 presents advanced features, such as more sophisticated resource types and concurrency control. In Section 6 we discuss implementation issues, and present simple prototype measurements. Section 7 concludes this paper.

2. RELATED WORK

In this section, we first present an overview of four prominent abstractions for distributed programming, namely the *message passing*, *remote method invocation (RMI)*, *tuple space*, and *publish/subscribe* abstractions. We focus on what we believe to be the core[§] abstractions, pointing out which properties and concepts of these respective abstractions are interesting for P2P settings, and which are less adequate for such environments. Thereafter, we discuss two prominent specifications for remote interaction (in Java) often mentioned in the context of P2P computing, namely the Java Message Service (JMS), and JXTA.

2.1. Message passing

Message passing is probably the abstraction providing the most explicit form of distributed interaction. Operating system-level concepts for network communication such as *socket*s are usually reflected up to 'higher' levels, as also exemplified by Java (`java.net.Socket`). The explicit use of these *low-level* abstractions is often bypassed by using specific libraries, for instance based on variants of the *message passing interface* (MPI).

As a rather general and low-level abstraction, message passing can be used to implement other abstractions, including our BL abstraction. The fact that message passing is, however, too low-level for many (P2P) applications is also illustrated by recent trends observed in work around the MPI in Java, consisting of emphasizing multi-party interaction more than strict pairwise object interaction (e.g. [22]), and viewing conveyed data as objects (e.g. [23]) rather than as simple bytes.

2.2. RMI

Originally introduced as the *remote procedure call* (RPC) abstraction for procedural programming models (e.g. *Sun RPC* [24], *DCE RPC* [25]), remote invocations have been quickly applied to object settings. Most respective approaches promote some form of entities remotely invoked through *proxies* [26]. Such an object mimics the remote object, i.e. it offers an interface which derives from that of the object it represents, and its *methods* are invoked locally as if the invocations were made on the original remote object. This approach blends well into a global model of objects interacting through

[§]Comparing abstractions is in fact not trivial, since the borders between them are not exactly clear. For instance, many recent *tuple space* variants (e.g. [18]) provide a *publish/subscribe*-like call-back scheme (e.g. [18,19]), and publish/subscribe is sometimes implemented with a variant of the proxy abstraction (e.g. [20,21]) known from asynchronous RMIs.

method invocations, as remote invocations hide the underlying message passing (although inversely message passing can very well be built on top of remote invocations [22]).

Early representatives of this model are given by the *guardians* in Argus [27] (with its follow-up CLU [28]) or *network objects* in Modula-3 [29] (followed by Obliq [30]). In the same philosophy, Java introduces the RMI [31] paradigm. Several variations of the RPC/RMI paradigm have appeared reducing the strict binding of an invoking object with an invoked object, by adding an asynchronous flavor (e.g. [32]), or by offering the possibility to atomically invoke several objects (e.g. [33]).

RMIs are not *a priori* harmful for dynamic settings such as P2P environments. Especially 'large', location-bound resources are best used via RMIs. As we will elucidate shortly, the BL abstraction actually embraces RMI by offering the possibility of lending asynchronous proxies. What is usually provided as a separate 'lookup service', i.e. a *name service* (cf. white pages) or a *trade service* (cf. yellow pages) with RPC, is even an inherent part of the BL abstraction. In a true P2P environment, one can namely not presuppose the knowledge of which peer is hosting what resources. This contradicts models such as that proposed by Java RMI, where objects are registered with their respective local lookup services (*registries*), and finding a remote object requires the identity of the host in order to connect to the corresponding registry. By integrating the lookup service with the BL abstraction, it is inherently distributed to suit the nature of P2P applications (cf. [5]).

2.3. Tuple space

The *tuple space* underlying the *generative communication* style originally advocated by Linda [8] provides a simple, yet powerful, distributed shared memory abstraction. A tuple space is composed of a collection of ordered *tuples*, equally accessible to all hosts of a distributed system.

The principle of tuple spaces has since undergone considerable evolution. There have been a series of attempts to transform the structured form of tuples into an object form, mainly by moving from the type equivalence for tuple elements of primitive types in Linda to a more general type conformance of object types. In contrast to early approaches to integrating the tuple space with objects (e.g. Smalltalk [34]), which promoted tuples as sets of objects, later approaches, such as [35] (C++), [36] (Objective Linda) or [18,37,38] (Java), consider tuples as single objects, however often 'degrading' their fields to tuple elements.

The indirect interaction between components obtained with the tuple space abstraction is a first step towards programming in dynamic distributed settings. However, the (original) tuple space paradigm has several drawbacks when being used for large-scale peer-based settings. First, the use of a blocking primitive for obtaining a copy of a tuple (non-exclusive consumption, `read()`) hampers decoupling of components. This is nicely illustrated by a recent trend observed in tuple spaces, consisting of supplementing the original blocking primitives used by consumers with asynchronous notification of these consumers upon appearance of new matching tuples (e.g. [18,37,38]). Second, the notion of tuple is very limited. Tuples invariably represent pass-by-value semantics. Furthermore, and as already mentioned above, virtually all tuple space implementations (including object-oriented ones), lack *encapsulation* of tuples by viewing these as sets of public fields and supporting the expression of consumer interests based on these fields only. Also, concurrency control is provided with respect to an entire tuple, through a specific primitive (exclusive consumption, `in()`). This does not match general, object-oriented and dynamic distributed settings. There, concurrency control involving an object of

interest depends on the semantics of that resource object (which would be the tuple) and cannot simply be dealt with outside the object, but will most likely affect the implementation of that object.

As we will illustrate, the BL abstraction offers more flexibility, by generalizing the notion of tuple to objects, which can be passed with various semantics.

2.4. Publish/subscribe

With the *publish/subscribe* [9] abstraction underlying *anonymous communication*, producers publish data and subscribers subscribe to data. This indirect form of interaction between remote components, inherited from its ancestor, the tuple space abstraction, is passed along to our BL abstraction. Indeed, the BL abstraction has been strongly influenced by our own work on *type-based publish/subscribe* (TPS) [17], a recent application of publish/subscribe to object settings emphasizing static type safety and encapsulation. In order to compare things at the same level of abstraction, we discuss here TPS as representative of publish/subscribe interaction.

TPS, like any publish/subscribe (i.e. multicast) abstraction, focuses on one-to-many interaction by exchanging objects between components with pass-by-value semantics. Hence, TPS is communication-centric, and concurrency control has to be built on top. In addition, objects shared with TPS have to be rather fine-grained, as conveying large objects can quickly lead to network congestion. This is particularly true, as once a subscriber has expressed interest in a type of objects, any published instance matching the subscriber's criteria is *immediately* sent to that subscriber.

This can become prohibitively expensive in a large-scale (typically, tens of thousands of users) P2P setting. As we will show, the BL abstraction can be viewed as a variant of the TPS abstraction generalized in order to remedy the above-mentioned drawbacks. The BL abstraction namely abstracts from the nature of conveyed objects (TPS focuses on pass-by-value semantics), and separates these objects from the QoS parameters (in the implementation of TPS with a specific compiler described in [17], QoS are specified on a per-type base). Furthermore, the BL abstraction supports borrower activation and deactivation (similarly to subscriptions in publish/subscribe), and symmetrically provides lender *activation and deactivation* (while a published object can neither be recalled nor replaced [39]).

2.5. JMS

As already mentioned, the publish/subscribe abstraction has often been used to model interaction in P2P environments. In particular Sun's own Java API for publish/subscribe interaction, the JMS [40], has received much attention (e.g. Narada Brokering [10]).

The BL abstraction and the JMS API are not fundamentally opposed. We believe that the JMS could even be extended to achieve BL-like interaction, however without providing the same high-level guarantees regarding type safety and encapsulation. These are namely distinctive features of the BL abstraction, which, by emerging from research efforts, builds on 'recent' concepts of Java to implement those features.

2.6. JXTA

Sun also provides a P2P-specific API, called JXTA [41], along with an implementation. JXTA represents the most popular attempt of rigorously specifying a set of constituents for a P2P service, and

its coexistence with JMS confirms the existence of a gap between the publish/subscribe abstraction and the P2P paradigm.

With respect to the BL abstraction, which represents a simplified and higher-level abstraction exploiting recent features of the Java language to enforce static type safety, JXTA can be viewed as far more complex and low-level, which, like most P2P specifications/systems, deals primarily with data as XML structures. In short, the BL abstraction is to the JXTA specification what the RPC/RMI abstraction is to the MPI specification.

3. BORROWERS AND LENDERS

With the BL abstraction, peers, which are in the following also viewed as remote components[¶], communicate anonymously and indirectly by making objects available to each other.

3.1. Model

An indirect interaction of two peers through such *resource objects*, or simply *resources* (see the next section) can be seen as a contract with distinct roles and actions for the respective peers. (1) A source peer (playing the role of *lender*) exports a resource by indicating that it is willing to *lend* that resource to other peers, and (2) a peer willing to import such resources (acting as *borrower*) must express the desire to *borrow* that 'kind' of resource.

Any participation in an interaction, whether lending or borrowing, is limited in time. Both lenders and borrowers can hence be *activated* and *deactivated*. This is conveyed by corresponding methods in the `Participant` interface, an abstract supertype for both borrowers and lenders, in Figure 1. The `constrain()` method offers the possibility of expressing constraints on resources, and will hence be discussed in the next section focusing on resources.

3.2. Lenders

In order to lend a resource, the `Lender` class is instantiated, by passing it the corresponding resource and a key. Although this key could easily be put inside the resources themselves (e.g. as a field with access methods involved in predicates), we have preferred to make it first class for several reasons. First of all, programmers of distributed applications are used to making use of explicit 'names', and such names can be represented by keys. Second, the resulting string matching can be performed extremely quickly when matching lent resources with borrowers. A third advantage of explicitly associating keys with resources lies in the enforcing of access control through these keys, an important aspect in P2P computing.

To enforce type safety, the `Lender` class (just like the `Borrower` class) makes use of genericity (Section 6.2). An instance of the `Lender` class has a fixed type value for the `R` parameter.

[¶]For presentation simplicity we view one peer as corresponding to exactly one application component. This, however, is not a necessity.


```
public interface Participant<R extends Resource> {
  public void activate()
    throws ActiveException, RemoteException;
  public void deactivate()
    throws InactiveException, RemoteException;
  public R constrain()
    throws InvalidConstraintException;
  ...
}

public final class Lender<R> implements Participant<R> {
  public Lender(R lent, byte[] key) {...}
  ...
}

public final class Borrower<R> implements Participant<R> {
  public Borrower(Inbox<R> in, byte[] key) {...}
  ...
}

public interface Inbox<R> {
  public void deliver(R r);
}
```

Figure 1. Borrowers and lenders (excerpt).

To ensure that borrowers and lenders are only instantiated with objects which are resources, the type parameter of the class `Participant` is bound by `Resource` (this bound is inherited by subclasses).

Note that lenders are not resources themselves, meaning that they cannot be lent to others. The identity of a lender depends on the hosting peer and a unique lender identifier for that very peer.

3.3. Borrowers

Through the desire to borrow resources, one expresses interest in particular objects. Borrowers express *which* objects they are precisely interested in through the following criteria.

- Type: the type of a resource, in the sense of its static description as a set of public members, can be used to express what resources are of interest. The criteria for type conformance can range from explicit type conformance to less strict *structural* (*implicit*) conformance (see Section 4.1.3).
- Predicates: borrowers can also describe predicates expressed in a statically type-safe manner based on the public members of the type of the resources of interest. The best example in Java, in which methods have a single return value, and the `equals()` method is used to verify value equality of objects, are nested method invocations ending by a call to that method, e.g. `resource.m1().m2().equals(expectedValue)`.

- Key: as outlined above, one 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.

The class `Borrower`, which is instantiated to express interest in certain resources, is outlined in Figure 1. Examples of its use are given in the next section.

A resource qualifies for becoming accessible to a borrower if all above-mentioned criteria are fulfilled, i.e. the resource's type matches the queried type, the predicate is satisfied by the resource, and the key specified by the borrower justifies access to the resource.

While the type of resource to borrow is indicated by a *type parameter* provided upon creation of an instance of `Borrower` (just as with the `Lender` class), the predicate is expressed in a type-safe manner through a *dynamic proxy* acting as formal argument, obtained through the `constrain()` method. Examples illustrating this follow in the next section.

A matching resource lent to a borrower is passed asynchronously to that borrower. To that end, the corresponding resource is passed as a parameter upon invocation of the `deliver()` method on the call-back object of type `Inbox` associated with that borrower. Just as in the case of the `Borrower` and `Lender` types, the type parameter of the `Inbox` type does not have to be explicitly bound to the `Resource` type. Through the use of objects implementing `Inbox` with instances of type `Borrower`, the bound introduced for the type parameter of `Participant` is inferred.

3.4. Synchronization

A strong *synchronization* between components with respect to concurrent insertions/extractions of tuples in a tuple space, such as obtained when considering (1) the real time order of the invocation of primitives by consumers (*strict consistency*), or even only (2) the respective local orders (*sequential consistency* [42]), cannot be implemented in the presence of peer and transmission failures [43], in particular when striving for scalability. This observation is backed by a number of fundamental impossibility results in asynchronous distributed systems (e.g. [44]), which make it impossible to solve problems such as *membership* and *mutual exclusion* in the presence of peer failures, even at a small scale.

Hence, no 'inherent' synchronization is built into the basic BL abstraction. When a previously unavailable resource is made available (or vice versa) through a corresponding instance of `Lender`, there is no direct synchronization with other peers. For instance the execution of `activate()` through a lender returns 'immediately', and the unavailability of a borrowing peer at that very moment does not necessarily result in an exception. An exception might however be thrown if the exporting peer experiences communication problems of a more general nature (indicated through an instance of a subclass of the Java RMI `RemoteException`), e.g. when trying to communicate with immediate neighbor peers.

The lack of specific synchronization primitives in the basic BL abstraction might seem surprising to users acquainted with the tuple space abstraction and its synchronized `in()` primitive (see Section 2.3), which is often used to implement a form of mutual exclusion (prone to node failures and with limited scalability). However, as we illustrate in Section 5.3, the BL abstraction indeed supports the implementation of *concurrency control*. As resources are a generalization of tuples, these can be used to synchronize components, with even more flexibility than tuples, since these resources


```
interface Resource {
  void setConformance(int depth) throws NotSupportedException;
  void setProtocol(Protocol p) throws NotSupportedException;
  void setQoS(QoS qos) throws NotSupportedException;
  void setReplacement(boolean transparent) throws NotSupportedException;
  ...
}

interface ValueResource extends Resource, Serializable {}

interface ReferenceResource extends Resource, Remote {
  void setSynchronization(boolean lazy) throws NotSupportedException;
}

interface LazyResource<R extends ValueResource> extends RemoteResource {
  void setDownload(boolean automatic) throws NotSupportedException;
  R download(Protocol p) throws NotSupportedException;
}
```

Figure 2. Basic resource types.

involve methods whose implementations are provided by the application. By replicating such resources, partial failures can be tolerated.

4. RESOURCES: BASIC CONCEPTS

Borrowers and lenders are created with respect to resources, which are instances of application-defined types. We present an overview of the guidelines for the design of such types, before presenting the basic resource types and illustrating these through concrete examples.

4.1. Common traits of resource types

Resource types are subtypes of the `Resource` interface presented in Figure 2, and are further divided into different kinds of resources (see also the next section).

4.1.1. Abstract versus concrete types

All resource types, including subtypes defined by P2P applications, have in common that they have to be defined as abstract types, i.e. interfaces. Similarly, return types of methods should be abstract types, etc. In particular, the examples introduced later on make use of our own ‘primitive object types’, all defined with corresponding interfaces (e.g. `String` and `StringImpl`).

This restriction is not a consequence of our abstraction, but rather of its implementation in Java. The dynamic proxies used to 'access' resources from remote peers, whether these resources are passed by value or by reference, are not available for classes. In short, a dynamic proxy class `CP` for a class `C` would be implemented as a subclass of `C`, making the overriding (in the goal of intercepting) of `final` or `private` methods, and any fields, impossible (see Section 6.1). Interfaces, whose declaration cannot contain any of the above, but are mainly `public` methods, are easily supported.

Note that for the same reason Java RMI is also restricted to interfaces. The associated `rmic` pre-compiler namely also generates proxies (and skeletons) as classes for the respective interfaces.

Our abstraction hence complies with Java's specification for *pass-by-reference* remote object interaction, i.e. Java RMI. Furthermore, by making remotely invocable resource types implement the very `java.rmi.Remote` interface (Figure 2), our abstraction seemlessly integrates with Java RMI, although not using it underneath. Similarly, our abstraction complies with Java's specification for *pass-by-value* remote interaction (serialization), by making resources passed by value subtype `java.io.Serializable` (see Figure 2).

4.1.2. Contract methods

The basic resource types have predefined methods, somewhat reflecting the 'contracts' introduced by the use of such resources. These methods, called *contract methods* in the following, hence differ between resource types, and sometimes *can be* implemented by a resource class, but *do not have to be*. Depending on its return type, such a method can have an empty body, or return specific values.

As an example, the basic resource type `Resource` declares methods giving borrowing peers the possibility of setting preferences, such as QoS parameters, transmission protocols to be used, or the policy of replacement of resources (transparent replacement versus re-delivering, see Section 5.5). These methods are used when expressing borrower criteria through the `constrain()` method of an instance of `Borrower`, and have been put into the resource types rather than into the `Borrower` type itself since every resource type potentially defines its own methods. Note, however, that, although one would expect many of these parameters to be expressed on a per-type base, they are set through instance methods. This is a consequence of the fact that `static` methods cannot be declared in Java interfaces and cannot be intercepted with dynamic proxies.

4.1.3. Type conformance

A contract method of primary importance is the `setConformance()` method. Through this method, the conformance strictness between the types of resources queried by a peer and the effectively corresponding, and hence assigned, resources can be set.

In fact, in our context of resource lending, we are mostly concerned with the static type safety of individual components. The goal is to be able to add/remove new components at runtime, each of these components being able of incarnating multiple resource importers (borrowers) and/or exporters (lenders), and to provide developers with static type safety with respect to the resources lent and borrowed by individual components as a tool to safely devise those components. Providing a form of global or distributed static typing would require an *a priori* agreement on types, i.e. an explicit global type hierarchy, offering only little flexibility. The BL abstraction aims at providing

support for static type safety of individual components, yet avoids an explicit global type hierarchy. To that end, it promotes a novel, flexible, notion of conformance 'levels' for the type conformance between these components, which in our .NET prototype is paired with language interoperability [45]. This is achieved through the `setConformance()` contract method. When invoked by a component describing a borrower, that component provides an integer value representing the minimum *depth* of conformance expected. A depth of 0 represents *explicit* type conformance, meaning that a resource which is an instance of a class `C` is only accessible through a borrower parameterized by an interface `I` if `C` explicitly (in the sense of Java; directly or recursively) implements `I`.

What could be called structural type conformance is further divided. With a depth of 1, class `C` above would not have to implement `I`, but would only have to provide a method for each method of `I`, with a corresponding signature. A depth of 2 would further relax restrictions on the parameter types of methods in `C`, such that those parameter types would themselves only have to implicitly conform, i.e. with a depth of 1, to those of the respective methods in `I`, etc.

4.1.4. Exceptions

When defining methods for custom resource types, application developers are *encouraged* to follow the Java RMI 'philosophy' for exceptions, consisting of reflecting possible failures occasioned by the distributed nature of interaction through the `RemoteException` type (which we borrow from Java RMI) in corresponding method signatures.

Java's strong support for exceptions in fact allows two ways of dealing with such distribution-related issues, namely explicitly, such as explained above for RMI, or by 'hiding' resulting exceptions: the possible throwing of exceptions of the type `java.lang.RuntimeException` (or any subtypes) does not have to be reflected by a method's signature, and hence the invocation of such a method does not require an enclosing `try...catch` statement. This approach has been chosen by many authors of libraries for distributed programming in Java, including the authors of the Java mapping for CORBA [46], who used such exceptions to improve transparency.

We provide the programmer with the choice between the two ways of handling exceptions. A failure occurring upon invocation of a resource method declaring the possible throwing of a `RemoteException` is signalled as an instance of that type, otherwise as a `RuntimeException`. The examples given in the following illustrate the use of both kinds of exceptions.

Note that when *forcing* resources to make use of `RemoteException`, which would definitely be a better practice, violations could only be detected at runtime, since, unlike Java RMI, the BL abstraction is not implemented with a pre-compiler.

4.2. Value resources

`ValueResources` in Figure 2 represent generally fine-grained, statefull objects, which, as their name suggests, are passed by value to components with corresponding borrowers.

A simple example for such resources is information concerning upcoming talks (presentations or also meetings), shared between researchers working in an industrial research laboratory or a university. In both contexts, researchers can be invited and asked to give talks. A typical type in Java whose instances would be used to incarnate the talk advertisement could look as follows:

```
interface VTalk extends ValueResource {
  String getTitle();
  String getSpeaker();
  String[] getAbstract();
  Long getStartTime();
  Long getExpectedDuration();
  void prettyPrint(OutputStream os);
}
```

While the first five methods give access to various attributes of talk advertisements, the `prettyPrint()` method offers the possibility of printing the entire talk advertisement in a formatted style. `OutputStream` refers to the class defined in `java.io`, and the corresponding parameter hence makes it possible to print the advertisement to a terminal, or a text box in an arbitrary window. Alternatively, one could imagine a method for taking a graphical window as a parameter, which, when invoked, would output the talk to that window.

Advertising such a talk requires the implementation of a corresponding class and its instantiation (the catching of exceptions is omitted in the following for simplicity):

```
class VTalkImpl implements VTalk {
  public VTalkImpl(String title, String speaker, ...) {...}
  ...
}

byte[] key =
  new StringImpl("Swiss Federal Institute of Technology").getBytes();
VTalk t = new VTalkImpl("Borrow/Lend", ...);
Lender<VTalk> tLender = new Lender<VTalk>(t, key);
tLender.activate();
```

Interest in any talks given by a particular speaker could then be expressed as follows[‖]:

```
class VTalkInbox implements Inbox<VTalk> {
  public void deliver(VTalk t)
    { t.prettyPrint(System.out); }
}

byte[] key =
  new StringImpl("Swiss Federal Institute of Technology").getBytes();
Borrower<VTalk> talks =
  new Borrower<VTalk>(new VTalkInbox(), key);
VTalk t = talks.constrain();
t.setConformance(0);
t.getSpeaker().equals("Patrick Eugster");
talks.activate();
```

[‖]The example illustrates a logical *and* two constraints expressed through `t`. Constraints expressed through different proxies obtained by several calls to `constrain()` are logically *or*-ed.

The third-last line nicely illustrates the use of contract methods. In this case, the borrowing peer indicates that it is only interested in receiving instances of classes that explicitly implement the `VTalk` interface. This invocation, just like the previous one, is 'registered' by a proxy referenced by `t` (see Section 6.1), and not performed immediately on any resource.

4.3. Remote resources

In certain cases it might be better to leave such a talk resource on the exporting peer, and instead advertise it as a remote reference, through which desired information can be obtained through remote invocations. Typical motivations are overly long abstracts that might not even be read by most users receiving the advertisement, or even attached articles written by the speaker and corresponding to the contents of that talk.

In general, as resources gain in size, and maybe become location-bound, it might be more appropriate to view them as services and access them from a distance. In that case, a peer offering such a service would rather lend it as a `RemoteResource`, providing interested peers access to it through a proxy:

```
interface RTalk extends RemoteResource {
  String getTitle() throws RemoteException;
  String getSpeaker() throws RemoteException;
  String[] getAbstract() throws RemoteException;
  Long getStartTime() throws RemoteException;
  Long getExpectedDuration() throws RemoteException;
  void prettyPrint(OutputStream os)
    throws RemoteException;
}
```

Instances of the `RemoteException` type declared in the `throws` clause of all methods of the `RTalk` type are used here to indicate problems occurring at runtime in remote interactions.

4.4. Lazy resources

When implementing the above talk example with remote resources, however, one problem becomes apparent. The `prettyPrint()` method does not make any sense anymore, since its input argument of type `OutputStream` represents an object local to the borrower and is neither serializable nor remotely accessible.

A better solution than implementing a remotely accessible output stream consists of making talk advertisements downloadable *on demand*. Indeed, the `prettyPrint()` method will most probably print all the information related to a talk, i.e. all fields of such an instance. Transferring a copy of a talk advertisement to be printed from the exporting peer to a borrowing peer will be more efficient than having the exporting peer make several remote calls back to an output stream on a borrowing peer to pass the individual fields.

To support lazy pass-by-value semantics for resources, we introduce the `LazyResource` type, which combines flavors of both pass-by-value and pass-by-reference semantics. Resources implementing that type usually represent larger objects than pure value resources and are hence only downloaded when really required (*lazy pass-by-value* semantics).

```
interface LTalk extends LazyResource<LTalk>,
  ValueResource
{
  String getTitle() throws RemoteException;
  String getSpeaker() throws RemoteException;
  String[] getAbstract() throws RemoteException;
  Long getStartTime() throws RemoteException;
  Long getExpectedDuration()
    throws RemoteException;
  void prettyPrint(OutputStream os);
}
```

The type parameter representing the resource that can be downloaded through the `download()` method of type `LazyResource` is not necessarily the same as the type of the resource itself. Although this applies in general, as in the case of the `LTalk` type shown above, one could indeed imagine a resource acting as a service through which other, possibly smaller, resources could be downloaded on demand.

A similar effect could be achieved without explicitly introducing such a resource type. For instance, one could equip the `RTalk` with a method returning an array of strings, which, invoked from a remote peer on a proxy to such a talk resource, would automatically transfer a formatted representation of the talk by value. The advantage of introducing a type for lazy resources appears when the instances of a resource type are both serializable *and* remotely accessible. One could, as in Java RMI, give priority to the remote nature of objects, which we also advocate if a resource type explicitly subtypes both `RemoteResource` and `ValueResource`. However, much more flexibility is provided if the user is given the possibility of specifying *if and when* to download a remotely accessible resource, and *how*, i.e. which protocol to exploit for the transfer (e.g. `ftp`, `http` or their own), which in fact reflects the *ad hoc* and self-organizing character of P2P computing. Also, in the case where the resource type is parameterized by itself, such as the `LTalk` type above, a borrowing peer can decide that resources are to be automatically downloaded upon their first invocation.

5. RESOURCES: ADVANCED FEATURES

In this section, we discuss features we regard as advanced with respect to the basic model underlying our BL abstraction presented so far. These features include, for instance, two additional resource types (*dynamic resources* and *replicated resources*) and how to achieve concurrency control with the BL abstraction.

5.1. Dynamic resources

In certain cases, more *late binding* is required than provided by structural type conformance, in the sense that even at the compilation of a component, the types of imported and/or exported resources are not known. A dynamic form of interaction, even exceeding that of Java *introspection*, is then advisable. Similar forms of dynamic interaction have already proven useful in distributed contexts, in combination

with both pass-by-value (e.g. *self-describing messages* [9]) but also pass-by-reference (e.g. *dynamic invocation interface* (DII) and *dynamic skeleton interface* (DSI) [46]) semantics.

To that end, we introduce the `DynamicResource` type. Lending such a resource means mainly implementing a general method `invoke()` through which the resource can be invoked. A method `getDescription()` returns a description of the functionalities implemented by a such resource at runtime and is required for the communication infrastructure to be able to match these resources against the criteria of borrowers. Similarly, that method can be used for borrowers to express interest in resources, without compile-time knowledge of the interfaces of those resources. Single constraints on the interfaces offered at runtime by resources are also intrinsically specified by expressing predicates through the `invoke()` method of a proxy obtained via the `constrain()` method.

Consider for example a borrower relying on such a dynamic interaction for the talks presented above (again omitting exceptions):

```
class DynInbox implements Inbox<DynamicResource> {
  public void deliver(DynamicResource d) {
    d.invoke("prettyPrint",
             new Object[]{System.out});
  }
}

byte[] key =
  new StringImpl("Swiss Federal Institute of Technology").getBytes();
VTalkInbox<DynamicInbox> inbox = new DynamicInbox();
Borrower<DynamicResource> talks =
  new Borrower<DynamicResource>(inbox, key);
DynamicResource d = talks.constrain();
Object[] args = new Object[]{"Patrick Eugster"};
d.invoke("getSpeaker", null).invoke("equals", args);
talks.activate();
```

Note that the `invoke()` and `getDescription()` methods, although seeming somewhat redundant to the methods defined by Java's introspection classes, are necessary. Latter methods would only reflect methods that are defined 'statically' by a resource class, while `DynamicResources` will in the general case only implement the `invoke()` method (through which further dispatching is done explicitly).

Any kind of resource, i.e. value resource, reference resource or lazy resource, can be a dynamic resource.

5.2. Replicated resources

As already outlined, we have kept peer and transmission failures in mind when designing the BL abstraction. In order to tolerate a certain number of such failures, the BL abstraction provides a means of replicating resources. When lending a resource, a lender can define a default value `create` for the number of replicas required for that resource. Depending on the protocols used underneath, which reflect both consistency and fault-tolerance requirements for (replicas of) a resource and are chosen by the application, typically `create`-1 (stateless objects, peer crashes), $\lfloor \frac{\texttt{create}-1}{2} \rfloor$

(stateful objects with consistent states, peer crashes [47]), $\lfloor \frac{\texttt{create}-1}{3} \rfloor$ (stateful objects, Byzantine failures [48]) peer failures can be tolerated. A value of 1 for `create` means that no replicas are to be created automatically. This value is used if several components explicitly start a replica representing the same resource (with identical keys). (A value of 0 indicates that the BL engine is free to decide on a replication scheme and degree.)

Methods `getRelevantState()` and `setRelevantState()` are used, respectively, for obtaining the state of a resource and to make sure an automatically created replica of that resource has the same state.

Note that in the case of a value resource, automatically created *replicas* are clearly separated from *copies* created for borrowers with matching interests. Former copies represent the resource at its initial state, i.e. with identical immutable states, and are sealed from borrowers. Latter copies are those used by borrowers and as a consequence of the value semantics are allowed to manifest diverging states.

Replicas of reference resources, however, always present the same resource, and their states have to remain consistent according to certain criteria with respect to actions performed by borrowers with these resources (e.g. invocations of resource methods). For such resources, the `getSemantics()` method must return a description of the semantics associated with the individual methods in order to trigger the appropriate replication protocols.

5.3. Concurrency

Reference resources represent a powerful means for implementing concurrency control. The form of mutual exclusion obtained with the `in()` primitive of the tuple space (see Section 2.3) can be easily emulated by implementing a class as follows:

```
class Token implements ReferenceResource {
  private boolean taken = false;
  synchronized public void take() {
    if(taken) wait();
    taken = true;
  }
  public void cede() {
    taken = false;
    notify();
  }
}
```

Any borrower expressing an interest in such a resource then obtains a proxy to it, and all concurrent invocations of the `take()` method of the lent resource finally go through Java's own `synchronized` primitive. This built-in mechanism guarantees mutual exclusion between all accesses to members of an object marked with that key word, and can hence also be used directly to implement any form of concurrency control as required by the semantics of the resource, i.e. as defined by the application.

Note, however, that a remote resource (just like any remote object in the sense of Java RMI) cannot be locked from the outside, meaning that the following example will not work:


```
class RTalkInbox implements Inbox<RTalk> {
  public void deliver(Rtalk t)
    {
      synchronized(t) {... action requiring mutual access to t ...}
    }
}
```

The object `t` namely represents a local object, i.e. a proxy. Exclusive access to the proxy is guaranteed, but not to the original, remote resource.

To avoid hampering *liveness* in the case of a crash failure of a peer hosting a critical resource, such resources can be replicated (just like tuple spaces implementations should replicate tuples, especially those acting as tokens for mutual exclusion, on several hosts to tolerate crash failures), by following simple guidelines. In order to achieve consistency between replicas (and hence *safety*), all replicas must let peers enter into the critical section in the same order. This requires concurrent invocations to synchronized methods to be made in a *total order* (ensured by choosing the appropriate `MembersSemantics()`), and `synchronized` to be replaced by a primitive with FIFO guarantees (e.g. a common FIFO mutex).

5.4. QoS and protocols

While different levels of consistency are imaginable for resources (and for their members), different levels of reliability are advisable for any interaction through the BL abstraction.

These different degrees can be seen as QoS, which are intrinsically tied to the protocols used underneath for communication. In P2P environments, which manifest a flavor of self-organization, it has proven interesting to provide different protocols for the underlying remote communication, and to make their choice explicit to the application. Since sometimes the choice is restrained by the nature of a single interaction, it is necessary to handle protocols and QoS expressions in isolation from the resource types.

Protocols and QoS are hence reflected as first-class constructs, as shown in Figures 1 and 2. These will not, however, be further discussed in this paper.

5.5. Replacing resources

Lent resources can also be replaced by new resources. This is expressed in the `Lender` type in Figure 3 through a corresponding method, and assists the underlying protocols in performing efficient distributed garbage collection (see Section 6.3).

With the talk example introduced in Section 4, the practical benefit of the `replace()` method for 'overriding' lent resources can also be demonstrated. Indeed, who has never had one of their talks pre- or postponed? By calling the `replace()` method with a new `VTalk` instance, a previously issued talk notification can be replaced. This does not mean that a previously issued talk cannot be delivered first to a borrower's `Inbox` anymore. However, if the order of the two talks is permuted during routing, the previous one is dropped.

In other terms, when objects, i.e. resources, correspond to the criteria expressed through a borrower, they can become accessible on that corresponding peer in two ways. In the most common

```
interface DynamicResource extends Resource {
  DynamicResource invoke(String methodName, Object args[])
    throws InvalidMethodException;
  ResourceDescription getDescription();
}

interface ReplicatedResource extends Resource {
  void setReplication(int create) throws NotSupportedException;
  byte[] getRelevantState();
  void setRelevantState(byte[] state);
  MemberSemantics getSemantics();
}

public final class Lender<R> implements Participant<R> {
  ...
  public void replace(R by) throws RemoteException {...}
  ...
}
```

Figure 3. Advanced resource types.

case, they are delivered by a call-back to an object of type `Inbox` registered by that peer upon description of the borrower. If the new resource was made available by the exporting peer through the `replace()` method, that resource can also become (somewhat invisibly) accessible through variables pointing to the replaced resource on peers which have already received the now obsolete resources (see Section 6.3.2).

6. IMPLEMENTATION ISSUES

This section first elucidates two 'recent' mechanisms of the Java language to which the BL abstraction owes its type safety and, then, of P2P protocols. Finally, we discuss the impact of the overhead of the above-mentioned mechanisms.

6.1. Dynamic proxies

With version 1.3 of Java, *dynamic proxies* [49] have been added as part of the Java core reflection API [50].

6.1.1. Overview

Dynamic proxies provide a limited form of *behavioral reflection* [51] (also known as *computational reflection* [52]) in combination with static type safety as a 'library'; that is, without specific support from the Java compiler or virtual machine. A dynamic proxy object created for an interface `I` can be

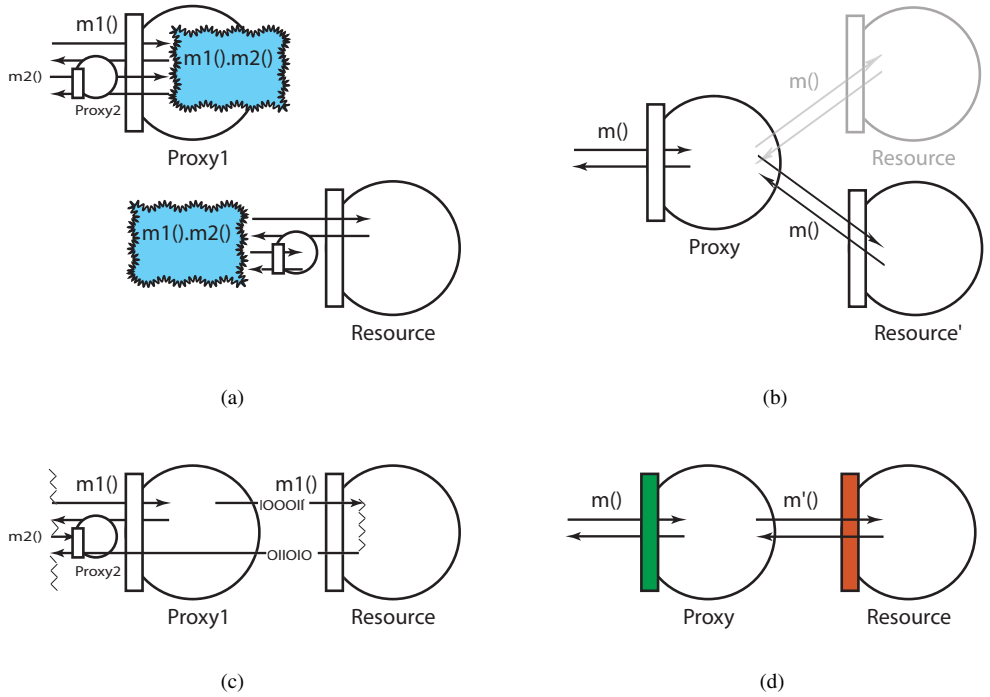


Figure 4. Dynamic proxies in the BL abstraction: (a) borrower criteria; (b) replacing a resource; (c) lazy remote synchronization; (d) structural conformance.

used in a consistent manner wherever an object of that type `I` or any supertype is expected, except that a method invocation performed on such a dynamic proxy object is in a first step *reified*, somehow enabling the passing from a typed context to an untyped context where *any* action can be performed in the confines of such a method invocation.

The implementation of our BL abstraction relies heavily on this concept of dynamic proxies (see Figure 4), on both the borrower and lender sides.

6.1.2. Borrower criteria

Through the `constrain()` method of the `Participant` class, a dynamic proxy object can be obtained for a queried type. Such an object acts as a formal argument for expressing queries, such as that expressed in the first example in Section 4, where `t.getSpeaker().equals("Patrick Eugster")` delimited the talks of interest for a given component. Figure 4(a) outlines how such

a query (think of `m1()` and `m2()` as the two nested method invocations above) expressed by a borrower is 'registered' and then replayed on a lent resource to verify whether that resource is indeed of interest to the borrower. The interception of contract methods is similarly performed by these proxies. The thereby implemented decorator pattern, in contrast to the regrouping of implementations of such contract methods in abstract resource classes to be subclassed by application-defined resource classes, has the advantage of not polluting the single inheritance resource class hierarchy.

6.1.3. Lent resources

When a resource is 'delivered' to a borrower, i.e. a borrower gains access to a resource through an invocation of `deliver` on a callback object of type `Inbox`, the passed object is a dynamic proxy. A 'real' resource is accessed indirectly through such a proxy, which seems normal in the case of a remote resource, and which also hides the effective collocated resource in the case of a value resource. The benefit of such a scheme is threefold:

- *Lazy synchronization*. A form of lazy synchronization can be implemented when invoking resources through proxies. This is outlined in Figure 4(c), where an invocation of a delivered resource by a borrower takes place through a (first) proxy, allowing the invocation to return a 'result', i.e. a second proxy, while the effective invocation has not yet been completed. An invocation of the returned proxy hence blocks, until the invocation result of the initial invocation of the first proxy is finally obtained (cf. *wait-by-necessity* [53]). A special case of this are automatically downloaded resources.
- *Resource replacement.* When replacing a lent resource by a new one, the reference to a borrower's local copy of such a resource can be kept valid if it is accessed through a proxy. This is illustrated by Figure 4(b), where a first resource is transparently to the invoker swapped against a new resource, which then performs the invocation. Without the addition of hooks into the virtual machine, this is in fact the only way of transparently 'changing' the instance of a user type pointed to by a variable.
- *Structural conformance.* Third, the wrapper pattern implemented by dynamic proxies can be used to implement the structural conformance provided as part of borrowers. Indeed, a class `C` can be instantiated in the virtual machine of a consumer, possibly after transferring it from its exporting peer, with a dynamic proxy of a non-explicitly related type `I` (see Section 4.1.3) pointing to it and giving access to it. This is illustrated by Figure 4(d), where a borrower can access a resource of a type distinct from what they had queried, by interposing a proxy which 'adapts' the invocations made on it by the borrower in order to perform them on the resource.

As mentioned in Section 4.1.1, however, dynamic proxies are only available for interfaces, which restricts the use of the BL abstraction in Java.

6.2. Genericity

In addition to dynamic proxies, genericity has been heavily used in the BL abstraction to provide static type safety without pre-compilation. In fact, the addition of genericity to the Java language

has represented a very vivid research field for several years, leading to a wide variety of approaches (e.g. [54,55]). Sun's own efforts to finally integrate genericity into the Java language [15] for version 1.5 are based on the solution described in [55], which provides *F-bound polymorphism*, enabling the parameterization of a type by itself (e.g. `LTalk` in Section 5.1). Its implementation originally relies on a *homogeneous translation* [55], meaning that type parameters are 'erased' at compilation, variables of such types are changed to the corresponding bounds (possibly the very root type `java.lang.Object`), and type casts are inserted wherever necessary. The drawback of such an approach is, however, that runtime type information is usually lost, meaning that an instance of a parameterized type does not know the value(s) of its type parameter(s) (unlike for instance with *virtual types* [54]). This fortunately by now recognized and addressed lack [15] at first represented an important drawback in the implementation of the BL implementation, as queries are expressed through dynamic proxies, which are created for specific, reified, types at runtime. An instance of the class `Borrower` parameterized by type `VTalk`, in order to create a dynamic proxy for that type `VTalk`, must pass a reification of that type in the form of an instance of `java.lang.Class` to the `java.lang.reflect.Proxy` class.

6.3. Protocols

One of the basic building blocks of a P2P communication infrastructure is made up of *multicast protocols*. Basically, an *event* such as the activation or deactivation of a borrower or a lender has to be disseminated among those peers for which that event is of importance.

6.3.1. Wanted: reliability and scalability

To offer a compromise between *reliability* and *scalability*, such protocols must ensure that relevant knowledge is received and stored by sufficient peers, but not by unnecessarily many. There are several faces of this tradeoff as follows.

- *Peer knowledge*: not every peer should know every other peer; however, a peer should be known by at least a minimum number of other peers.
- *Borrower knowledge*: the action and deactivation of borrowers should be notified to those peers concerned, but should not flood the network. They should be stored at a reasonable number of peers.
- *Resource knowledge*: similarly, the activation, deactivation and modification of a resource should be notified to, and stored by, a subset of peers only.

A seminal protocol we have implemented which addressed these issues was based on a *broadcast* protocol [12] offering *probabilistic guarantees*, meaning that with very high probability a peer would acquire some of the above-mentioned knowledge, although every peer would only know a subset of the other peers. To further limit the amount of acquired knowledge for a given peer, a second probabilistic protocol for *multicasting* knowledge has been proposed [13]. With that second protocol, interaction between peers increases as they become 'closer', in terms of both physical but also interest distance, i.e. overlappings between their borrower criteria.

6.3.2. *Replacing and deactivating resources*

The latter protocol is furthermore inspired by the intuitive concept of *message obsolescence* first studied and formalized in detail in [56]. The idea is that in most applications certain events, or in our case resources, make previously created ones obsolete. With some, even limited, support from an application in indicating such relationships (e.g. the `replace()` method in the `Lender` class), scalability of protocols can be significantly improved.

As pointed out in [57], efficient garbage collection of remotely accessed objects is not straightforward, and the original implementation found in Java RMI is an illustration of this observation. In a highly dynamic P2P setting, where resources are available in general only temporarily, it is particularly important for a resource creator to indicate the termination of a lender, rather than inversely keeping such an object alive waiting for the last remote peer to release its references to it.

6.4. Measurements

We focus here on illustrating the overhead of the BL abstraction with respect to its use of reflection mechanisms. More precisely, we measure the costs of testing structural type conformance at runtime, and the overhead of using dynamic proxies to invoke local resources as instances of such implicitly conformant types.

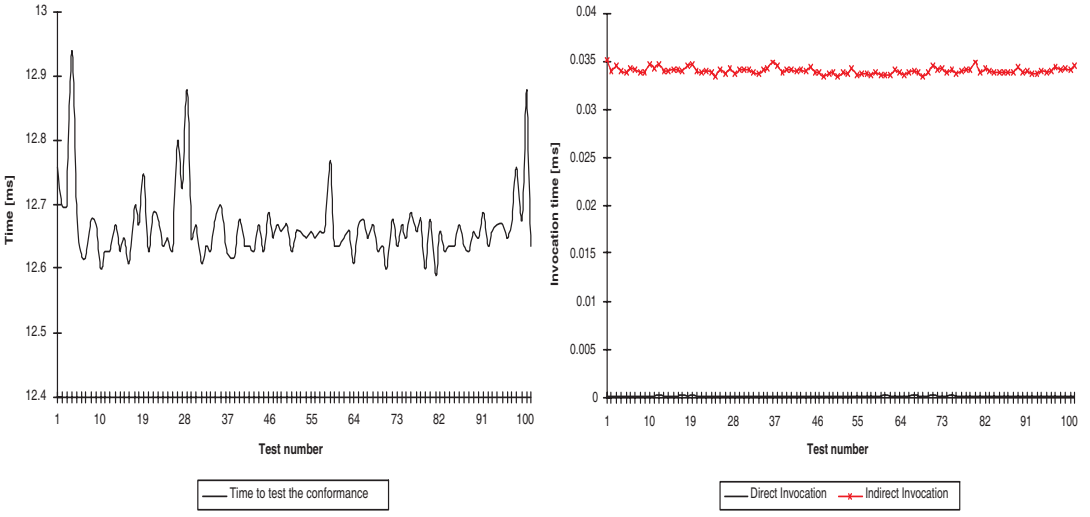
6.4.1. *Setting*

These measurements were obtained with a HP Omnibook XT6050, with the following configuration: Pentium III 1 GHz, 256 MB RAM, 30 GB HDD, Windows 2000 SP2. Note that these tests were performed with our .NET prototype version, i.e. Visual Studio .NET Enterprise Architect 2002 version 7.0.9466. The results obtained are comparable with our Java prototype, although genericity is not included, since the current solution for genericity in Java deals with genericity at compilation, and thus does not incur runtime overhead.

6.4.2. *Type conformance verification*

Figure 5(a) reflects the cost of testing conformance between a type `VTalk` (Section 4.1.3) and a type `VTalk2` which contains the exact same methods as `VTalk`, yet is not explicitly related to that type (i.e. a conformance depth of 1). One-thousand consecutive conformance tests have been run 100 times. In Figure 5(a), one can see that the average time measured is 12.66 ms. This seems relatively expensive, but only has to be performed seldom: since our underlying P2P multicast protocols constructs *global type conformance tables*, such a test only has to be performed once every time a lender or a borrower is expressed for a completely new type, i.e. whenever a completely new resource (sub)type is introduced.

Inside the entire update procedure of the global type conformance and routing tables in the multicast protocols, along with the dissemination of the code pertaining to the new type, this operation is negligible. Moreover, routing table updating and class dissemination would have to take place with any abstraction implemented on top of a P2P overlay network, and do not depend on the implicit conformance promoted by our BL abstraction.



(a) (b)

Figure 5. Overhead of structural conformance: (a) conformance verification; (b) dynamic proxy invocations.

6.4.3. Dynamic proxy invocations

In Figure 5(b) we represent the time for invoking method `getSpeaker()` of class `VTalkImpl` outlined in Section 4.2. The scenario consisted in performing 100×10^6 invocations (1) directly, and (2) indirectly using a dynamic proxy. The average direct invocation time is around 0.000 142 ms, while the average indirect time is of the order of 0.03 ms. This difference might be sensible 'locally', but its impact on global performance is negligible, given the strong asynchrony introduced with the BL abstraction. This is, however, difficult to illustrate, as it depends strongly on the target application (mainly on the proportion between remote interaction and local computation with value resources), and because of the absence of a 'reference': the indirect interaction with resources through dynamic proxies is a core concept of the BL abstraction, which cannot be separated from it. We believe that the overhead is in any case negligible (without even considering the unquantifiable gains in type safety achieved in return), given that applications benefitting from our BL abstraction are strongly distributed, and are hence strongly subject to delays incurred by network communication [45].

7. CONCLUSIONS AND FUTURE WORK

As illustrated in this paper, the BL abstraction abides well to P2P object environments, which can be described as completely decentralized and potentially large-scale dynamic distributed object settings.

The BL abstraction achieves its scalability by providing peers with the possibility of *asynchronously* lending and borrowing *resource objects*, reducing the coupling between these peers. This notion of resources, on the one hand, provides the BL abstraction with flavors of a 'high-level' abstraction in the sense that distribution-related issues such as serialization and location of resources are concealed, and encapsulation and static type safety are ensured. On the other hand, this model allows the BL abstraction to make 'low-level', yet in the context of P2P computing crucial, aspects related to distribution such as protocols and QoS explicit.

Let us summarize the characteristics of our BL abstraction with respect to those of other abstractions.

- BL combines an awareness of the underlying system and protocols (as with message passing) with a unified abstraction and higher-level guarantees.
- Pass-by-reference semantics (as with RMI) are provided for dealing with 'large', location-bound resources, yet are provided with an inherent and distributed lookup service. To support asynchrony which is crucial in large-scale settings, reference resources can be invoked in a lazy-reply manner.
- Concurrency control (as with the tuple space) takes into account application-defined policies and fault tolerance.
- Pass-by-value semantics (as with publish/subscribe) are provided for 'small' resources. A lazy flavor of pass-by-value semantics is provided for 'larger' resources, as well as support for garbage collection.

We are currently investigating the application of our BL abstraction to the design and implementation of a scalable general communication substrate for *collaborative virtual environments* (CVEs) [58]. In such environments, distributed users interact through a virtual shared world, and resources such as the constituents of the world have to be shared among those peers hosting 'interested' users (e.g. users to whom these constituents are visible), in a way which ensures scalability (by avoiding unnecessarily many replicas of data structures), but also reliability (by ensuring that there are sufficiently many replicas).

ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for many valuable suggestions which helped improve this paper. This research was financially supported by the Swiss National Science Foundation NCCR-MICS project under grant 5005-76322 (*Terminodes*; http://www.terminodes.com), as well as the European IST FET research project on Global Computing under grant IST-2001-33234 (*PEer-to-Peer Implementation and TheOry*, *PEPITO*; http://www.sics.se/pepito).

REFERENCES

1. Wego.com Inc. What is Gnutella? http://gnutella.wego.com/ [2000].
2. Freenet: A distributed anonymous information storage and retrieval system. http://www.freenetproject.org/ [2000].
3. Oram A. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.
4. Rowstron A, Druschel P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. *Proceedings of the 4th IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2001)*, November 2001; 329–350.

5. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H. Chord: A scalable peer-to-peer lookup service for Internet applications, August 2001; 149–160.
6. Ganesh AJ, Kermarrec A-M, Massoulié L. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers* 2003; **52**(2):139–149.
7. Birrell AD, Nelson BJ. Implementing remote procedure calls. *ACM Transactions on Computer Systems* 1984; **2**(1):39–59.
8. Gelernter D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems* 1985; **7**(1):80–112.
9. Oki B, Pfluegl M, Siegel A, Skeen D. The information bus—An architecture for extensible distributed systems. *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, December 1993; 58–68.
10. Fox GC, Pallickara S. The Narada event brokering system: Overview and extensions. *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, June 2002.
11. DACE Distributed Asynchronous Computing Environment. http://www.d-a-c-e.com [2004].
12. Eugster PTh, Guerraoui R, Handurukande SB, Kermarrec A-M, Kouznetsov P. Lightweight probabilistic broadcast. *Proceedings of the 2001 IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, June 2001; 443–452.
13. Eugster PTh, Guerraoui R. Probabilistic multicast. *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, June 2002; 313–323.
14. Thai T, Lam H. *.NET Framework Essentials*. O'Reilly, 2001.
15. Sun Microsystems, Inc. Adding generics to the Java programming language. *Java Specification Request (JSR) 000014*.
16. Kennedy A, Syme D. Design and implementation of generics for the .NET common language runtime. *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'01)*, June 2001.
17. Eugster PTh, Guerraoui R, Damm CH. On objects and events. *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, October 2001; 131–146.
18. Freeman E, Hupfer S, Arnold K. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley: Reading, MA, 1999.
19. IBM. *MQSeries Publish/Subscribe Users Guide*. IBM, 2000.
20. Oberg RJ. *Understanding & Programming COM+*. Prentice-Hall: Englewood Cliffs, NJ, 2000.
21. OMG. *CORBAservices: Common Object Services Specification, Chapter 4: Event Service*. OMG, 2001.
22. Nelisse A, Kielmann T, Bal HE, Maassen J. Object-based collective communication in Java. *Proceedings of the Joint ACM Java Grande–ISCOPE 2002 Conference*, June 2001; 11–20.
23. Carpenter B, Fox G, Ko SH, Lim S. Object serialization for marshaling data in a Java interface to MPI. *Proceedings of the ACM 1999 Java Grande Conference*, June 1999; 66–71.
24. Srinivasan R. RFC 1831: Remote procedure call protocol specification version 2. *Technical Report*, Sun Microsystems, Inc., August 1995.
25. Rosenberry W, Kenney D, Fisher G. *OSF Distributed Computing Environment: Understanding DCE*. O'Reilly, 1993.
26. Shapiro M. Structure and encapsulation in distributed systems: The proxy principle. *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS '86)*, May 1986; 198–204.
27. Liskov B. Distributed programming in Argus. *Communications of the ACM* 1988; **31**(3):300–312.
28. Liskov B. A history of CLU. *ACM SIGPLAN Notices* 1993; **28**(3):133–147.
29. Cardelli L, Donahue J, Jordan M, Kalsow B, Nelson G. The Modula-3 type system. *Conference Record of the 16th ACM Symposium on Principles of Programming Languages (POPL '89)*, January 1989; 202–212.
30. Cardelli L. A language with distributed scope. *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, August 1995; 286–297.
31. Sun Microsystems, Inc. *Java Remote Method Invocation—Distributed Computing for Java (White Paper)*, 1999.
32. Caromel D, Klauser W, Vayssière J. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience* 1998; **10**(11–13):1043–1061.
33. Maassen J, Kielmann T, Bal HE. Efficient replicated method invocation in Java. *Proceedings of the ACM 2000 Java Grande Conference*, June 2000; 88–96.
34. Matsuoka S, Kawai S. Using Tuple space communication in distributed object-oriented languages. *Proceedings of the 3rd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, November 1988; 276–284.
35. Polze A. Using the object space: A distributed parallel make. *Proceedings of the 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, September 1993; 234–239.
36. Kielmann Th. Objective Linda: A coordination model for object oriented parallel programming. *PhD Thesis*, Department of Electrical Engineering and Computer Science, University of Siegen, Germany, September 1997.
37. Lehman TJ, Mac Laughry SW, Wyckoff P. TSpaces: The next wave. *Proceedings of the 32nd IEEE Hawaii International Conference on System Sciences (HICSS-32)*, January 1999.
38. Rowstron A. Optimising the Linda in primitive: Understanding tuple-space runtimes. *Proceedings of the 2000 ACM Symposium on Applied Computing*, 2000; 227–232.

39. Baehni S, Eugster PTh, Guerraoui R. OS support for peer-to-peer programming. *Proceedings of the 22th IEEE International Conference on Distributed Computing Systems (ICDCS '02)*, July 2002; 355–362.
40. Happner M, Burridge R, Sharma R. Java message service. *Technical Report*, Sun Microsystems Inc., October 1998.
41. Brookshier D, Govoni D, Krishman N. *JXTA: Java P2P Programming*. Sams Publishing, 2002.
42. Lamport L. How to make a multiprocessor computer that correctly executes multiprocess programs. *ACM Transactions on Computer Systems* 1979; **28**(9):690–691.
43. Bakken DE, Schlichting RD. Supporting fault-tolerant parallel programming in Linda. *IEEE Transactions on Parallel and Distributed Systems* 1995; **6**(3):287–302.
44. Fischer MJ, Lynch NA, Paterson MS. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 1985; **32**(2):217–246.
45. Baehni S, Eugster PTh, Guerraoui R, Altherr P. Pragmatic type interoperability. *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS '03)*, May 2003.
46. OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, 2001.
47. Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 1996; **43**(2):225–267.
48. Cristian F, Aghili H, Strong R, Dolev D. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation* 1995; **118**(1):158–179.
49. Sun Microsystems, Inc. *Dynamic Proxy Classes*, 1999.
50. Sun Microsystems, Inc. *Java Core Reflection API and Specification*, 1999.
51. Kiczales G, des Rivières J, Bobrow DG. *The Art of the Metaobject Protocol*. MIT Press: Cambridge, MA, 1991.
52. Maes P. Concepts and experiments in computational reflection. *Proceedings of the 2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, October 1987; 147–155.
53. Caromel D. Towards a method of object-oriented concurrent programming. *Communications of the ACM* 1993; **36**:90–102.
54. Thorup KK. Genericity in Java with virtual types. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, June 1997; 444–471.
55. Bracha G, Odersky M, Stoutamire D, Wadler Ph. Making the future safe for the past: Adding genericity to the Java programming language. *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, October 1998; 183–200.
56. Pereira J, Rodrigues L, Oliveira R. Semantically reliable multicast protocols. *Proceedings of the 19th IEEE Symposium On Reliable Distributed Systems (SRDS'00)*, October 2000; 60–73.
57. Philippsen M, Nester Ch, Haumacher B. A more efficient RMI for Java. *Proceedings of the ACM 1999 Java Grande Conference*, June 1999; 152–159.
58. Benford S, Greenhalgh C, Rodden T, Pycock J. Collaborative virtual environments. *Communications of the ACM* 2001; **44**(7):79–89.

