Abstracting Remote Object Interaction in a Peer-to-Peer Environment*

Patrick Thomas Eugster
Distributed Computing and Systems Research Group
Chalmers University of Technology
Göteborg, S-412 96 Sweden
peugster@cs.chalmers.se

Sébastien Baehni
Distributed Programming Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne, Switzerland
sebastien.baehni@epfl.ch

ABSTRACT

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 been devoted substantial consideration recently.

This paper presents an abstraction for remote object interaction in a P2P environment, called borrow/lend (BL). We present the principles underlying our BL abstraction, and its implementation 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 predating abstractions, unifies flavors of these, but also how it captures the constraints specific to P2P environments.

Categories and Subject Descriptors

C.2 [**Computer Systems Organization**]: Computer Communication; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications*; D.2 [**Software**]: Software Engineering; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.3 [**Software**]: Programming Languages; D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*

General Terms

Distributed Programming

Keywords

Borrow/lend, peer-to-peer, abstraction, type, Java

*This research was supported by the Swiss National Science Foundation.

1. INTRODUCTION

Through the overnight success stories of many non-profit programs enabling the collaboration of users, by *lending* data resources and inversely *borrowing* resources of interest, in the goal of distributing media throughout the Internet (e.g., Gnutella [27], Freenet [14]), the paradigm of *peer-to-peer* (P2P) [22] 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 plays the same role.

While the moral promoted by some of the applications which stand behind the recent success of P2P computing is sometimes dubious and has been the source of much polemics, the practical value of the P2P paradigm is unquestionable. This observation has led to many important contributions in the field, and is emphasized by the very fact that the striving for decentralizing applications, with the avoiding 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 already.

Ever since, P2P computing has been the subject of more profound studies, also benefitting from previous research in that community. For instance, many predating application-level protocols for data routing or membership management, especially such with 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 led also to novel protocols (e.g., [24, 25]), 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), the *tuple space* (emerged from parallel computing based on distributed shared memory) or the *publish/subscribe* (for mass dissemination of events) abstractions, do indeed also make sense in P2P settings, and have already been used successfully in such 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., [12]).

In this paper we present an abstraction called *borrow/lend* (BL)[1] we have implemented in the Distributed

[1]The term *query/share* (QS) appeared in a previous version

Asynchronous Computing Environment (DACE) together with protocols [11, 9] *specifically* for P2P object programming. We present the implementation of the BL abstraction in Java, which has been chosen as implementation language mainly because it includes many basic mechanisms and specifications for distributed programming already, and provides an all to rare combination of *genericity*[2] and *reflection*. As we illustrate in this paper through the BL abstraction, these concepts are very useful for the implementation of abstractions for distributed programming in statically typed object-oriented programming languages.

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 [10]. Nevertheless, we believe 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 illustrate by contrasting it with such "classic" abstractions. For instance, (1) it embraces a form of (asynchronous) RMI but at the same time incorporates the functionality of a distributed lookup service, (2) offers borrowers a fine-grained encapsulation-preserving means of expressing the kind of resources of interest (based on the methods of application-defined resource types), (3) makes transmission protocols and parameters governing *Qualities of Service* (QoS) explicit, and since lended resources are not perpetual, (4) gives the possibility of explicitly recalling (in the sense of canceling) or replacing such resources, aiding garbage collection and hence improving scalability.

The remainder of this paper is structured as follows. Section 2 illustrates the notions of resource borrowing and lending. Section 3 characterizes the nature of resources. In Section 4 we discuss implementation issues. Section 5 compares our BL abstraction with related abstractions and specifications for distributed programming. Section 6 concludes this paper.

2. BASIC ABSTRACTION

With the BL abstraction, peers, which are in the following also viewed as remote components,[3] communicate anonymously and indirectly by making objects available to each other.

2.1 Model

An indirect interaction of two peers through such *resource objects*, or simply *resources* (see 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 peers, and (2) a peer willing to import such resources (acting as *borrower*) must express the desire to *borrow* that "kind" of resources.

of this report.

[2]Genericity is foreseen for Java 1.5. [26]. Our implementation relies on the compiler prototype available from Sun.

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

```
public final class Lender<R implements Resource>
  java.io.Serializable
{
  public Lender(R lent, String[] key)
    throws InvalidResourceTypeException {...}
  public void activate()
    throws ActiveException, RemoteException {...}
  public void deactivate()
    throws InactiveException, RemoteException {...}
  public void replace(R by)
    throws RemoteException {...}
  ...
}

public final class Borrower<R implements Resource>
  java.io.Serializable
{
  public Borrower(Inbox<R> in, String[] key)
    throws InvalidResourceTypeException {...}
  public void activate()
    throws ActiveException, RemoteException {...}
  public void deactivate()
    throws InactiveException, RemoteException {...}
  public R constrain() throws
    InvalidConstraintException {...}
  ...
}

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

Figure 1: Borrowers and lenders (excerpt)

2.1.1 Lenders

More precisely, when a resource is lent, it is made available to *any* party which expresses interest in it. There is no inherent exclusive access to resources.

For a P2P application to scale properly, it is important for individual peers to be aware of, and to indicate, the expiration of resources. Lenders, in the sense of the interaction, are hence activated as well as deactivated, and lended resources can also be replaced by new resources. This is expressed in the `Lender` type in Figure 1 through corresponding methods, and assists the underlying protocols in performing efficient distributed garbage collection (see Section 4.2).

Note that lenders are not resources themselves, meaning that they can not be lent to others. However, lends (as well as borrowers) can be made persistent, e.g., by saving them to stable storage and reactivating them later. The identity of a lender depends on the hosting peer and a unique identifier for that peer.

2.1.2 Borrowers

Through the desire of borrowing resources, one expresses interest in particular objects. Borrowers (can) 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 3.1.3).

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 usually 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: When lending a resource, one can explicitly attach a key in the form of an array of strings to it. Though this could easily be put inside the resources themselves (e.g., as field with access methods involved in predicates), we have preferred to isolate this criterion, because programmers of distributed applications are used to explicit names, the resulting string matching can be performed extremely quickly, and, last but not least, this key can be used also to enforce security barriers, an important aspect in P2P computing. Note that type safety is hereby not compromised.

While the type of resource to borrow is indicated by a *type parameter* provided upon creation of an instance of `Borrower`, and the key is provided as an array of strings to the constructor, a predicate is expressed in a type-safe manner through a *dynamic proxy* acting as formal argument, obtained through the `constrain()` method (see Section 4.1).

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 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.

2.2 Concurrency and Synchronization

When a previously unavailable resource is made available (or vice versa) through a corresponding instance of class `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 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.

In a large scale and completely decentralized setting, it is inherently difficult to achieve strong consistency guarantees, when considering the possibility of partial failures. Since the hiding of distribution beyond a level where transparency can be absolutely ensured (e.g., location transparency of resource exporters/importers, or transparent marshaling/unmarshaling of resources) has never been a goal when designing the BL abstraction, neither has the addition of an inherent mechanism for concurrency control such as provided by the `in()` primitive known from tuple spaces (see Section 5.1.3).

However, a concurrency control mechanism can be indeed very useful in scenarios such as the replication of a resource. In such a case, some support from the developer of the resource is required, and this issue is addressed by viewing such a type of resource as a specific one (see Section 6).

2.3 Qualities of Service and Protocols

While different levels of consistency are imaginable for such resources, different levels of reliability are advisable for different scenarios.

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. 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 Figure 1. Due to space restrictions, these are however not further discussed in this paper.

3. RESOURCES

Borrowers and lenders are created with respect to resources, which are instances of application-defined types. We overview the guidelines for the design of such types, and illustrate these through examples.

3.1 Common Characteristics

Resource types are subtypes of the `Resource` interface presented in Figure 2, and are further divided into different kinds of resources (see Section 3.2).

3.1.1 Abstract vs 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 namely 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 4.1.1). Interfaces, whose declaration can not contain any of the above, but mainly `public` methods, are easily supported.

For the same reason Java RMI is also restricted to interfaces. The associated `rmic` pre-compiler generates namely proxies (and skeletons) according to a similar scheme than the above-described.

Hence, our abstraction complies with Java's specification for *pass-by-reference* remote object interaction, i.e., Java RMI. By furthermore making remotely invocable resource types implement the very `java.rmi.Remote` interface (Figure 2), our abstraction seemlessly integrates with Java RMI,

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

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

Figure 2: Basic resource types (excerpt)

though 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`.

3.1.2 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 the return type, such methods can have empty bodies, or return specific values to indicate the "absence" of an implementation.

The basic resource type `Resource` for instance contains methods allowing borrowing peers the setting of preferences, such as QoS parameters, transmission protocols to be used, or the policy of replacement of resources (transparent replacement vs re-delivering, see Section 4.2.2). 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 own methods. Note however that though one would expect many of these parameters to be expressed on a per-type base, they are set through instance methods. This is mainly a consequence of the fact that `static` methods can not be declared in Java interfaces.

3.1.3 Type Conformance

A contract method of primary importance is the `setConformance()` method. Through that 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 static type safety of individual components. The goal is to be able to add/remove new components at run-time, 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 type safety 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, by supporting different "levels" of type conformance between these components. 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) implements `I` (or an interface explicitly subtyping `I`).

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

3.1.4 Exceptions

When defining methods for custom resource types, application developers are *encouraged* to follow the Java RMI "philosophy" for exceptions, consisting in 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 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 [21], which use 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 run-time, since, unlike Java RMI, the BL abstraction is not implemented with a pre-compiler.

3.2 Variations

The identification of the different natures of resources and their translation to abstract types is an ongoing issue. We present here a preliminary set of basic resource types outlined in Figure 2.

3.2.1 Value Resources

`ValueResource`s 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 are informations 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 talk advertisement could look like the following (`OutputStream` refers to the class defined in `java.io`):

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

Advertising such a talk requires the implementation of a corresponding class, and its instantiation:

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

VTalk t = new VTalkImpl("BL", ...);
Lender<VTalk> tLender =
  new Lender<VTalk>(t, new StringImpl[]{"Chalmers",
                                         "CS"});
tLender.activate();
```

Interest in any talks could then be expressed like the following (by omitting the catching of exceptions for simplicity):

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

Borrower<VTalk> talks =
  new Borrower<VTalk>(new VTalkInbox(),
                      new StringImpl[]{"Chalmers",
                                       "CS"});
VTalk t = talks.constrain();
t.setConformance(0);
t.getSpeaker().equals("Patrick Eugster");
talks.activate();
```

The third-last line illustrates the use of contract methods. In this case, the borrowing peer indicates that it is only interested in receiving instances of classes which explicitly implement the `VTalk` interface. This invocation is "trapped", and not performed on any effective resource.[4]

With this talk example, the practical benefit of the `replace()` method for "overriding" lent resources can also be demonstrated. Indeed, who has never had one of her/his 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 can not 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.

3.2.2 Remote Resources

If the abstract of a talk is very long, or the speaker has written an article which corresponds exactly to the contents of that talk and could be added to the advertisement, it might be of interest 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 though remote invocations.

In general, as resources gain in size, and maybe become location-bound, it might be more adequate 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:

[4]The example illustrates a logical *and* of two constraints. A logical *or* of two (or more) constraints requires these to be expressed on individual proxies obtained by several calls to `constrain()`.

```
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 in remote interactions.

3.2.3 Downloadable 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 whether serializable nor remotely accessible.

A better solution than implementing a remotely accessible output stream consists in making talk advertisements downloadable *on demand*. Indeed, the `prettyPrint()` method will most probably print all 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 even 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 `DownloadResource` 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).

```
interface DTalk extends DownloadResource<DTalk>,
  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 is not necessarily the same than the type of the resource itself. Though in general this applies, like in the case of the `DTalk` type shown below, 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 downloadable resources appears when the instances of a resource type such as `DType` are both serializable *and* remotely accessible. One could, like 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, we want the flexibility of being able to specify *if and when* to download a remotely accessible resource, and *how*, i.e., which protocol to exploit for the downloading (e.g., `ftp`, `http`, or own ones), which is in fact one of the characteristics of P2P programming. Also, in case the resource type is parameterized by itself, such as the `DTalk` type above, a borrowing peer can decide that resources are to be automatically downloaded upon their first invocation.

3.2.4 Dynamic Resources

In certain cases, more *late binding* is required than provided by implicit type conformance, in the sense that even at compilation of a component, the types of imported and/or exported resources are not known. A dynamic form of interaction, already supported to some extent by Java through *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* [20]) but also pass-by-reference (e.g., *dynamic invocation interface* (DII) and *dynamic skeleton interface* (DSI) [21]) 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 resource at run-time, and is required for the communication infrastructure to be able to match such resources against borrower's criteria. 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 run-time 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});
  }
}

String[] s = new StringImpl[]{"Chalmers", "CS"};
Inbox<DynamicInbox> inbox = new DynamicInbox();
Borrower<DynamicResource> talks =
  new Borrower<DynamicResource>(inbox, s);
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, though 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).

4. IMPLEMENTATION ISSUES

This section elucidates implementation issues of the current prototype of the BL abstraction, focusing on mechanisms of the Java language and P2P protocols.

4.1 Java

The static type safety provided by the BL abstraction is a merit of two very "recent" Java mechanisms, namely *dynamic proxies* and *genericity*.

4.1.1 Dynamic Proxies

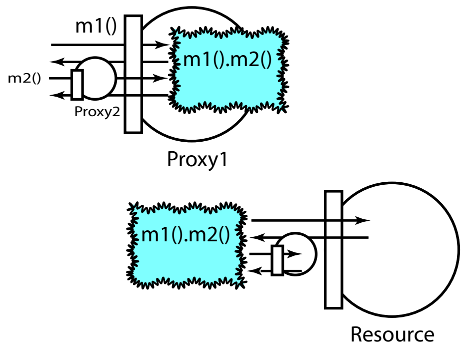
With version 1.3 of Java, *dynamic proxies* have been added as part of the Java core reflection API. Dynamic proxies provide a limited form of *behavioral reflection* (a.k.a. *computational reflection*) in combination with static type safety as "library", that is, without specific support from the Java compiler or virtual machine. A dynamic proxy object created for an interface `I` can be 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 one 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 3).

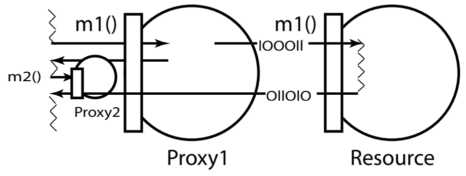
Borrower criteria: Through the `constrain()` method of the `Borrower` class, a dynamic proxy object can be obtained for a queried type. Such an object acts as formal argument for expressing queries (Figure 3(a)). Furthermore, the interception of contract methods is performed by these proxies. The thereby 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.

Lent resources: Copies of resources delivered to consumers are dynamic proxies, in the case of value resources "hiding" the effective collocated resources. This has several benefits. First, a form of weak synchronization can be implemented when invoking resources through proxies (Figure 3(b)), but also when dealing with automatically downloaded resources. Second, when replacing a lent resource by a new one, the reference to a local copy of such an object can be kept valid (Figure 3(c)). Without addition of hooks into the virtual machine, this is in fact the only way of "changing" transparently the instance of a user type pointed to by a variable. 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 *3.1.3*) pointing to it and giving access to it (Figure 3(d)).

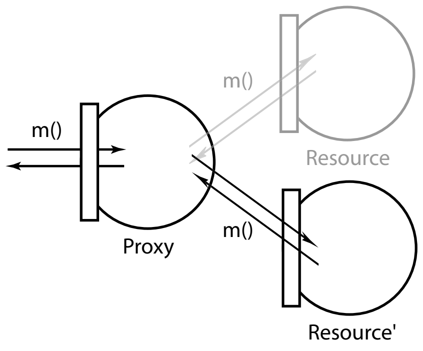
As mentioned in Section *3.1.1* however, dynamic proxies are only available for interfaces, which restricts the use of the BL abstraction in Java.



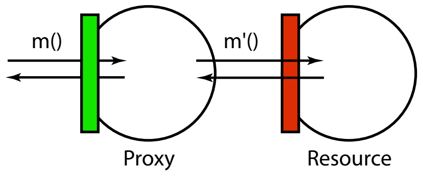
(a) Borrower criteria



(b) Lazy remote synchronization



(c) Replacing a resource



(d) Structural conformance

Figure 3: Dynamic proxies in the BL abstraction

4.1.2 Genericity

Besides 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. Sun's own efforts to finally integrate genericity into the Java language [26] for version 1.5 is based on the solution described in [3], which provides *F-bound polymorphism*, enabling the parameterization of a type by itself (e.g., `DTalk` in Section 3.2). Its implementation originally relies on a *homogenous translation* [3], 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 inserted wherever necessary. The drawback of such an approach is however that run-time type information is usually lost, meaning that an instance of a parameterized type does not know the value(s) of its type parameter(s). This, fortunately by now recognized and repaired lack [26], 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 run-time. An instance of 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.

4.2 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.

4.2.1 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:

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 [11] offering *probabilistic guarantees*, meaning that with very high probability a peer would acquire some of the above-mentioned knowledge, though 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 [9]. With that second protocol, interacton between peers increases as they become "closer", in terms of both physical but also interest distance, i.e., overlappings between their borrower criteria.

4.2.2 Replacing and Deactivating Resources

Latter protocol furthermore applies the intuitive concept of *message obsolenscence* first studied and formalized in detail in [23]. 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 [7], efficient garbage collection of remotely accessed objects is not straightforward, and the original implementation found in Java RMI an illustration of this observation. Especially in a highly dynamic P2P setting, where resources are available in general only temporarily, it is increasingly important for a resource creator to indicate the end of a lend, rather than inversely keeping such an object alive waiting for the last remote peer to release its references to it.

5. RELATED WORK

In this section, we first contrast the BL abstraction with respect to prominent abstractions for distributed programming. Then, we compare it with similar P2P-specific abstractions and APIs. For space restrictions and fairness, we focus here mainly on efforts related to Java.

5.1 Abstractions

The idea here is not to claim that the BL abstraction somehow replaces predating abstractions, but rather to show how it has been influenced by those abstractions. Comparing abstractions is in fact not trivial, since the borders between them are not exactly clear. For instance, many recent *tuple space* variants provide a *publish/subscribe*-like call-back scheme (e.g., [13]), publish/subscribe is sometimes implemented with a variant of the proxy abstraction (e.g., [19]) known from asynchronous *remote method invocations*.

5.1.1 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 exemplified also by Java (`java.net.Socket`), but are often bypassed by the use of libraries, for instance based on variants of the *message passing interface* (MPI).

As a rather fundamental abstraction, message passing could be used to implement the BL abstraction, especially since much recent work around the MPI in Java has emphasized multi-party interaction more than strict pairwise object interaction (e.g., [18]), and conveyed data is increasingly viewed as objects (e.g., [6]) rather than as simple bytes.

5.1.2 Remote Method Invocation

Originally introduced as the *remote procedure call* (RPC) abstraction for procedural programming models, remote invocations have been quickly applied to object settings, promoting some form of entities remotely invoked through proxies while hiding the underlying message passing (though inversely message passing can very well be built on top of remote invocations [18]). In the same philosophy, Java introduces the *remote method invocation* (RMI) 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., [5]), or by invoking atomically several objects (e.g., [17]).

The BL abstraction embraces RMI by offering the possibility of lending asynchronous proxies. What is usually provided as separate "lookup service", i.e., a *name service* (cf. white pages) or a *trade service* (cf. yellow pages) with RPC, is an inherent part of the BL abstraction. In a true P2P environment, one can not presuppose the knowledge of which peer is hosting what resources. This contradicts models such as the one 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. [25]).

5.1.3 Tuple Space

The *tuple space* underlying the *generative communication* style originally advocated by Linda 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.

There have been a series of attempts to transform the structured form of tuples to 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, which promoted tuples as sets of objects, later approaches, like Sun's own JavaSpaces [13] considered tuples as single objects, however often "degrading" their fields to tuple elements.

The BL abstraction is close to the tuple space, in the sense that components do not directly interact (e.g., by invoking each other's methods), but via a form of "object pool". In particular, the BL abstraction comes close to recent tuple space variants such as JavaSpaces, by delivering objects of interest through a call-back, and through the possibility of limiting the time during which objects are available to others. With the BL abstraction, QoS and transmission protocols are made explicit, and the distributed application interacts with the abstraction for the purpose of enhancing garbage collection. Additionally, the BL abstraction enables the expression of borrower criteria with application-defined resource types based on their methods rather than on their fields, preserving encapsulation of resources.

Note however that tuple spaces offer strong support for the synchronization of distributed components through `in()`-like primitives, while the BL abstraction currently does not implicitly provide a similar primitive.

5.1.4 Publish/Subscribe

With the *publish/subscribe* abstraction underlying *anonymous communication*, producers publish data and subscribers subcribe 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) [10], a recent application of publish/subscribe to object settings emphasizing static type safety and encapsulation.

The BL abstraction can be viewed as a generalization of the TPS abstraction, in that it 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 [10], 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 [1]).

5.2 Specifications

As already mentioned, the publish/subscribe abstraction has been often used to model interaction in P2P environments. In particular Sun's own Java API for publish/subscribe interaction, the Java Message Service (JMS) [15], has been devoted much attention (e.g, [12]). Sun has however defined a P2P-specific API, called JXTA [4], 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 somehow confirms the sensible gap between the publish/subscribe abstraction and the P2P paradigm.

With respect to the BL abstraction, which represents a simplified and more high-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.

6. 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 distributed object settings.

The BL abstraction achieves its scalability by providing peers 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" aspects related to distribution, yet crucial for P2P computing, such as protocols and QoS, explicit.

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) [2]. 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 sufficiently many replicas).

In that context, we are exploring new resource types, such as resources which are *automatically replicated* (cf. [8]), possibly also implemented with a *caching mechanism*, or resources which are passed by value along *with computation* (cf. [16]).

7. ACKNOWLEDGEMENTS

We would like to thank the anonymous referees for many valuable suggestions which helped improve this paper.

8. REFERENCES

[1] S. Baehni, P.Th. Eugster, and R. Guerraoui. OS Support for Peer-2-Peer Programming. In *Proceedings of the 22th IEEE International Conference on Distributed Computing Systems (ICDCS '02)*, pages 355–362, July 2002.

[2] S. Benford, C. Greenhalgh, T. Rodden, and J. Pycock. Collaborative Virtual Environments. *Communications of the ACM*, 44(7):79–89, July 2001.

[3] G. Bracha, M. Odersky, D. Stoutamire, and Ph. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '98)*, pages 183–200, October 1998.

[4] D. Brookshier, D. Govoni, and N. Krishman. *JXTA: Java P2P Programming.* Sams Publishing, 2002.

[5] D. Caromel, W. Klauser, and J. Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, September 1998.

[6] B. Carpenter, G. Fox, S.H. Ko, and S. Lim. Object Serialization for Marshaling Data in a Java Interface to MPI. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 66–71, June 1999.

[7] M. Philippsen Ch. Nester and B. Haumacher. A More Efficient RMI for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 152–159, June 1999.

[8] J.R Douceur and R.P. Wattenhofer. Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System. In *Proceedings of the 15th International Conference on Distributed Computing (DISC 2001)*, pages 48–62, October 2001.

[9] P.Th. Eugster and R. Guerraoui. Probabilistic Multicast. In *Proceedings of the 2002 IEEE International Conference on Dependable Systems and Networks (DSN 2002)*, pages 313–323, June 2002.

[10] P.Th. Eugster, R. Guerraoui, and C.H. Damm. On Objects and Events. In *Proceedings of the 16th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, pages 131–146, October 2001.

[11] P.Th. Eugster, R. Guerraoui, S.B. Handurukande, A.-M. Kermarrec, and P. Kouznetsov. Lightweight Probabilistic Broadcast. In *Proceedings of the 2001 IEEE International Conference on Dependable Systems and Networks (DSN 2001)*, pages 443–452, June 2001.

[12] G.C. Fox and S. Pallickara. The Narada Event Brokering System: Overview and Extensions. In *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, June 2002.

[13] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice.* Addison-Wesley, June 1999.

[14] *Freenet: A Distributed Anonymous Information Storage and Retrieval System.* http://www.freenetproject.org/, 2000.

[15] M. Happner, R. Burridge, and R. Sharma. Java Message Service. Technical report, Sun Microsystems Inc., October 1998.

[16] M. Izatt, P. Chan, and T. Brecht. Ajents: Towards an Environment for Parallel, Distributed and Mobile Java Applications. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 15–24, June 1999.

[17] J. Maassen, T. Kielmann, and H.E. Bal. Efficient Replicated Method Invocation in Java. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 88–96, June 2000.

[18] A. Nelisse, T. Kielmann, H.E. Bal, and J. Maassen. Object-Based Collective Communication in Java. In *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 11–20, June 2001.

[19] R.J. Oberg. *Understanding & Programming COM+.* Prentice Hall, 2000.

[20] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 58–68, December 1993.

[21] OMG. *The Common Object Request Broker: Architecture and Specification.* OMG, February 2001.

[22] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies.* O'Reilly and Associates, Inc., March 2001.

[23] J. Pereira, L. Rodrigues, and R. Oliveira. Semantically Reliable Multicast Protocols. In *Proceedings of the 19th IEEE Symposium On Reliable Distributed Systems (SRDS'00)*, pages 60–73, October 2000.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the 4th IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2001)*, pages 329–350, November 2001.

[25] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, August 2001.

[26] Sun. *Adding Generics to the Java Programming Language. Java Specification Request (JSR) 000014.*

[27] Wego.com Inc., *What Is Gnutella?* http://gnutella.wego.com/, 2000.